Self-Evolving AI Agents: A Survey of Feedback-Driven Generation, Evaluation, Memory, and Self-Modification

aha team

May 2026

Abstract

Self-evolving AI agents modify persistent components of their own operation–including prompts, memory, tools, code, workflows, data curricula, and sometimes model behavior–through feedback-driven improvement loops. This survey develops a unified framework for the field across language-model self-improvement, evolutionary computation, AutoML, agent architecture search, code repair, benchmark harnesses, and production agent frameworks. We organize the literature into five evolution loops, compare representative papers and open-source systems, summarize benchmark evidence, analyze user and production pain points, and identify safety and governance requirements for agents that learn to improve the machinery by which they act. We further introduce the Evolve-AGI Index as an evidence-to-index instrument for field maturity, combining benchmark performance, loop strength, evidence credibility, transfer verification, implementation access, field momentum, and governance readiness. The central thesis is that self-evolution should be evaluated as a controlled systems process: every claimed improvement must specify what changed, what feedback justified the change, what evaluator approved it, what it cost, what risks it introduced, and how it can be audited or rolled back.

Keywords: self-evolving AI; LLM agents; recursive self-improvement; agent evaluation; memory; code evolution; automated agent design; AI safety

Contents

1 Introduction

Artificial intelligence has historically been engineered as a largely static artifact. A model is trained, a policy is selected, a prompt is written, an architecture is deployed, and the resulting system is then evaluated as if its intelligence were a property fixed at release time. This view is convenient for benchmarking and for product deployment, yet it is increasingly misaligned with the systems now emerging around large language models, tool-using agents, automated program synthesis, and evolutionary search. Contemporary AI systems do not merely map inputs to outputs. They can inspect execution traces, write and execute code, critique their own reasoning, revise their prompts, call external tools, store episodic memories, generate new training tasks, and propose modifications to the very software that implements them. These capabilities suggest a shift from static AI to *self-evolving AI*: systems that improve through feedback-driven loops over their own components.

This survey studies that shift. We use the term *AI Self-Evolution* to denote a family of methods in which an AI system modifies some part of itself—its prompts, memory, tools, agent code, architecture, data curriculum, or neural weights—as a result of feedback collected from interaction with tasks, users, environments, verifiers, or other agents. The field is not a single algorithm. It is a convergence zone where language-agent reasoning, AutoML and neural architecture search, evolutionary computation, reinforcement learning from verifiable feedback, program synthesis, and open-ended science automation are beginning to share concepts. A prompt-refinement loop such as Self-Refine [36], a memory-based verbal reinforcement learner such as Reflexion [49], a symbolic-backpropagation agent optimizer [1], a meta-agent that writes new agent architectures [21], a Darwinian archive of self-modifying coding agents [76], a runtime monkey-patching agent inspired by the Gödel machine [48, 72], an evolutionary coding system such as AlphaEvolve [40], and a zero-data self-play reasoner [77] all differ in mechanism. Yet they can be compared under the same abstraction: a system observes performance, converts feedback into a modification proposal, validates the proposal, and conditionally retains the modified system.

The timing of this convergence is not accidental. Classical AI already contained the ingredients: Turing's behavioral framing of machine intelligence [54], recursive self-improvement thought experiments, Gödel-machine theory [48], meta-learning, evolutionary algorithms, and AutoML. What changed in the 2020s was the availability of foundation models that can operate over the representations in which modern AI systems are built. Prompts are text, tool descriptions are text, code is text, benchmark reports are text, stack traces are text, scientific papers are text, and even many architectural decisions can be serialized as text. Large language models therefore act as general-purpose variation operators: they can propose a new prompt, edit a Python class, design a test, summarize a failure, synthesize a training example, or explain why an earlier attempt failed. When such proposals are coupled to automated evaluators—unit tests, code executors, human preference models, software-engineering benchmarks, mathematical verifiers, web task success criteria, or peer-review rubrics—the result is a practical loop of self-modification and selection.

The research challenge is to understand when these loops constitute genuine self-evolution rather than ordinary adaptation or offline optimization. A system that is merely retrained by engineers after deployment is not self-evolving in the sense used here. A model that passively receives more data under a fixed update rule may be online learning, but not necessarily self-evolution. Conversely, a language agent that keeps its weights fixed but revises its memory and prompts after every failure can be self-evolving at the representational level at which its behavior is actually controlled. The key issue is not whether the modified component is neural, symbolic, or procedural. The key issue is whether the AI system participates in the generation, evaluation, and retention of modifications to its own operational machinery.

This chapter introduces the background, definition, boundaries, and contributions of the survey.

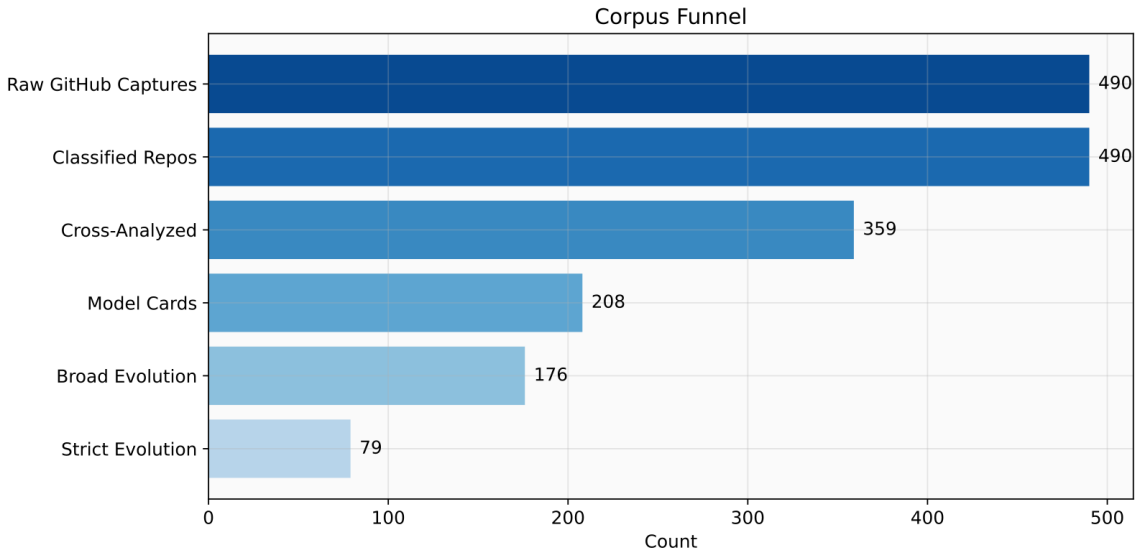

Figure 1: Survey corpus funnel. The figure links the public resource library to the paper evidence base: raw captures are filtered into classified repositories, model-card projects, strict self-evolution cases, and broader adjacent evidence.

Section 1.1 traces the path from static AI and the Turing test to Gödel machines, AutoML, evolutionary architecture search, large language models, and modern self-improving agents. Section 1.2 provides a formal definition of AI Self-Evolution as a feedback-driven modification process over system components. Section 1.3 distinguishes self-evolution from continual learning, online learning, self-supervised learning, meta-learning, AutoML, reinforcement learning, and conventional agent memory. Section 1.4 summarizes the survey's contributions: a Five Evolution Loops framework, analysis of more than nineteen open-source and research systems, a cross-domain unification of AutoML/NAS/LLM/evolutionary-computation/agent research, a researcher-network map of the field, and the Evolve-AGI Index as a field-maturity instrument. Section 1.7 outlines the rest of the paper.

1.1 Background: From Static AI to Self-Evolving AI

1.1.1 The static-system assumption in early artificial intelligence

The earliest public imagination of artificial intelligence was already entangled with questions of evaluation and change. Turing's 1950 proposal replaced the metaphysical question "Can machines think?" with an operational imitation game in which a machine's intelligence is judged through interaction [54]. The test did not prescribe a specific learning mechanism, but it established a durable pattern: intelligence would be inferred from behavior under a task protocol. Most subsequent benchmarks inherited this framing. A system is trained or programmed, placed behind an interface, and judged by its outputs. The protocol may be conversational, mathematical, perceptual, strategic, or embodied, but the object under test is normally treated as fixed during evaluation.

This static-system assumption shaped the organization of modern machine learning. A training phase produces parameters θ; a validation phase chooses hyperparameters; a test phase estimates generalization. The deployed model is then represented as a function $f_\theta : \mathcal{X} \to \mathcal{Y}$, and the central research question is how to learn θ from a dataset D. Even reinforcement learning, which emphasizes

interaction, usually freezes the learned policy when reporting benchmark scores. This division is scientifically useful because it separates search from assessment, reduces leakage, and makes comparisons reproducible. Yet it also narrows the idea of an AI system to the final artifact rather than the process that produced and continues to alter that artifact.

Symbolic AI and expert systems were also mostly static after design time. Engineers encoded rules, taxonomies, production systems, or search strategies. A knowledge base could be updated, but the update was typically performed by humans or by a narrow acquisition routine. The expert system did not redesign its inference engine, rewrite its knowledge-acquisition workflow, or choose a new representational scheme in response to failures. The system's intelligence was in its manually specified structure. Machine learning later replaced hand-written rules with learned functions, but the deployed artifact remained a snapshot.

The static assumption was reinforced by software-engineering practice. A model release is versioned, audited, and monitored. If performance degrades, developers collect data, retrain the model, patch code, or adjust prompts. The system does not usually hold authority to alter those components itself. This separation between the object being improved and the human process that improves it is the main boundary that AI Self-Evolution begins to erode. In a self-evolving system, some part of the improvement process becomes internal to the system's own operation.

The historical importance of static evaluation should not be understated. Without frozen benchmarks, it would be impossible to measure progress in speech recognition, computer vision, machine translation, game playing, or code generation. However, the same practice can obscure a new class of systems whose competence resides not only in their current policy but also in their ability to generate better future policies. A static snapshot of an evolving agent is analogous to a photograph of a learning organism: useful, but incomplete. For self-evolving AI, we must evaluate both the current behavior and the evolution operator that changes that behavior over time.

1.1.2 Self-reference and the Gödel-machine ideal

Long before current language agents, researchers considered whether an intelligent machine could improve its own learning algorithm. Schmidhuber's Gödel machine is the canonical formal proposal [48]. It describes a self-referential system that contains a representation of its own software and a proof searcher. The machine rewrites itself only after proving that the rewrite will improve expected utility according to a formal objective. The theoretical beauty of the Gödel machine lies in its treatment of self-modification as a rational action. It does not merely mutate itself; it requires a proof that the mutation is beneficial.

The original Gödel-machine ideal is far stronger than most modern systems. Current agents rarely possess complete formal models of their environments, their utility functions, or their own implementation. They cannot generally prove that a code patch will improve long-run utility. Yet the idea introduced several principles that reappear in contemporary self-evolution. First, the system must have some access to a description of itself. Second, self-modification must be evaluated relative to an objective. Third, the system must decide whether to accept or reject candidate modifications. Fourth, the improvement process is recursive: the mechanism that modifies the system may itself be modified.

Modern systems relax formal proof into empirical validation. Instead of proving that a modification is globally utility-improving, they run tests, benchmarks, simulations, or external evaluations. Gödel Agent, for example, explicitly invokes the self-referential inspiration of the Gödel machine but implements runtime self-modification through Python monkey patching and validation on tasks [72]. Darwin Gödel Machine combines the self-referential ambition with Darwinian search: a foundation model proposes code-level changes to agents, and an archive retains variants that improve on

benchmarks [76]. These systems do not deliver the mathematical guarantees of Schmidhuber's construction. They instead trade proof for empirical selection, scalability, and compatibility with messy software environments.

This shift from proof to evaluation is one of the defining compromises of practical self-evolution. Formal proof is attractive but brittle in open domains. Empirical evaluation is noisy but scalable. The self-evolving systems surveyed in this paper almost always use the latter. Reflexion stores verbal feedback after failed episodes [49]; Self-Refine iteratively critiques and revises outputs [36]; Agent Symbolic Learning produces language gradients over prompts, tools, and pipelines [1]; ADAS evaluates newly generated agent programs [21]; AlphaEvolve uses automated evaluators to select program variants [40]; Absolute Zero uses a code executor as a verifier for self-generated tasks [77]. The unifying pattern is not proof-carrying self-rewrite but feedback-carrying self-modification.

Gödel-machine theory also clarifies why self-evolution is potentially unstable. If a system can modify its own modification procedure, it may improve faster than a system limited to a fixed update rule. But the same recursion can amplify errors, reward hacking, unsafe objectives, or distributional drift. A self-modification that improves a short benchmark may degrade robustness; a prompt revision that increases success on one task may remove safety constraints; a code patch that passes tests may exploit the evaluator. Thus self-evolution inherits both the promise and the danger of self-reference. The survey's emphasis on loops, validation, rollback, and governance follows directly from this tension.

1.1.3 From hand-designed learning to AutoML and neural architecture search

A second lineage comes from automating the design of learning systems. AutoML asks whether the pipeline that turns data into a model can itself be optimized: data preprocessing, feature engineering, architecture selection, hyperparameter tuning, training schedules, and ensembling. Neural architecture search (NAS) narrows the question to architectures, often using reinforcement learning, evolutionary algorithms, differentiable relaxations, or weight sharing. Methods such as NASNet [81], Efficient NAS [42], DARTS [34], and regularized evolution for image classifiers [45] showed that search can discover architectures competitive with human designs.

AutoML and NAS matter for self-evolution because they shift intelligence from a learned model to a search process over models. The artifact is no longer only a set of weights; it is also the procedure that designs the architecture and training configuration. Classical NAS, however, is typically an external process. A controller proposes architectures, a training loop evaluates them, and engineers deploy the winner. The resulting model does not normally continue to search over its own architecture after deployment. In this respect, AutoML is an ancestor of self-evolution but not always an instance of it.

The analogy becomes tighter when the target of search is an agentic system rather than a neural network. ADAS frames automated design of agentic systems as a search over programs that implement agents [21]. The meta-agent writes code for new agent architectures, evaluates them on tasks, and stores successful designs. The search space is Turing-complete because agents are expressed as Python programs. This is NAS for agent software: instead of choosing convolutional cells or transformer blocks, the system searches over prompts, tool calls, memory strategies, decomposition routines, reflection procedures, and control flow. Agent Symbolic Learning makes a related move by treating agents as symbolic networks whose learnable elements are prompts, tools, and pipelines rather than differentiable weights [1]. The method then defines language losses, language gradients, and textual updates analogous to backpropagation.

Evolutionary computation deepens this lineage. NEAT evolves both topology and weights [52]; novelty search argues that divergent exploration can be more productive than direct objective

optimization [31]; MAP-Elites and quality-diversity methods maintain archives of high-performing diverse solutions [39]; POET co-evolves agents and environments in open-ended processes [58]; AI-GAs imagine systems that generate architectures, learning algorithms, and environments [13]. Modern self-evolving agents reuse these ideas but replace random or hand-coded mutation operators with LLM-guided semantic variation. The mutation operator now reads code, understands test failures, edits prompts, and proposes structured improvements.

Darwin Gödel Machine is the clearest synthesis. It maintains an archive of agent implementations, samples parent agents, asks a foundation model to modify their code, evaluates the child, and adds improved variants to the archive [76]. Its reported SWE-bench improvement from 20.0% to 50.0% and Polyglot improvement from 14.2% to 30.7% indicate that agent architecture can itself be a cumulative evolutionary object. AlphaEvolve provides an industrial-scale counterpart: a Gemini-powered evolutionary coding agent that combines broad exploration with deeper reasoning and MAP-Elites-style quality diversity, reporting improvements in matrix multiplication, data-center scheduling, chip design, and LLM-training kernels [40]. These systems make explicit what AutoML implied: the design process can become a computational object that is searched, evaluated, and improved.

1.1.4 The large language model revolution as an enabler of self-evolution

The rise of large language models changed the practical feasibility of self-evolution. Earlier systems could optimize numeric parameters, discrete architectures, or symbolic rules, but they struggled to operate over the heterogeneous artifacts that make up modern AI systems: prompts, tool APIs, logs, source code, test results, scientific rationales, and natural-language feedback. GPT-4, Claude, Gemini, and related frontier models are not merely stronger predictors of text; they are broadly competent manipulators of software and language-mediated procedures. They can read a stack trace, infer a bug, generate a patch, produce a unit test, critique a solution, rewrite a prompt, summarize a benchmark failure, and propose a new experiment.

This capacity turns language into a universal interface for self-modification. In Self-Refine, the model generates an output, critiques it, and refines it using its own feedback [36]. No weights change, but the output process evolves within an episode. Reflexion adds memory: failed attempts produce verbal reflections that condition later attempts [49]. Agent Symbolic Learning pushes the analogy further by defining language gradients over the nodes of an agentic computation graph [1]. RISE trains recursive introspection as a learned multi-turn behavior using reinforcement learning [43]. RAGEN decomposes trajectories into state, thinking, actions, and rewards, while identifying the "Echo Trap" in which agents appear to improve by repeating themselves rather than reasoning [59]. ReVeal treats self-verification as a first-class learned capability in code agents, optimizing generation and verification together [27].

The LLM revolution also enabled systems that act on code as an evolutionary substrate. Self-Evolve, an early code-evolution framework, used self-generated API knowledge and execution feedback to iteratively repair code, improving DS-1000 and HumanEval performance [25]. ADAS asks an LLM meta-agent to write new agent code [21]. Gödel Agent modifies its own runtime behavior through monkey patches [72]. Darwin Gödel Machine and AlphaEvolve use foundation models as semantic mutation operators over agent implementations and algorithmic code [40,76]. AI Scientist uses LLM agents to traverse a broader research loop: idea generation, experimental design, code execution, manuscript writing, and peer review [35]. These systems would be hard to imagine without models that can manipulate both natural language and executable artifacts.

Importantly, LLMs do not remove the need for evaluation. They make variation cheap and semantically informed, but they also hallucinate, overfit to rubrics, exploit loopholes, and produce

plausible but invalid rationales. The strongest systems therefore pair LLM generation with external feedback. Code executors, test suites, benchmark harnesses, environment rewards, and human or model-based judges provide the selection pressure. Absolute Zero's zero-data self-play depends on a code executor that verifies proposed tasks and solutions [77]. AlphaEvolve depends on automated evaluators [40]. ReVeal emphasizes reliable self-verification [27]. DGM evaluates modified agents on software-engineering benchmarks [76]. The lesson is that LLMs make self-evolution possible, but verifiers make it cumulative.

The field's empirical record reflects this verifier dependence. Prompt-only self-feedback can improve many tasks, but it can also stagnate or reinforce errors. Memory-based reflection can increase pass rates when feedback is specific and evaluators are reliable, but memory can grow unbounded. Code-level self-modification can discover novel strategies, but it requires sandboxing, rollback, and careful acceptance criteria. RL-based self-improvement can internalize revision behaviors, but it risks reward hacking, reasoning collapse, or echoing. Open-ended archives can escape local optima, but they are expensive and difficult to govern. These are not isolated problems; they are the core design trade-offs of self-evolving AI.

1.1.5 From isolated tricks to self-improvement systems

The most visible early LLM self-improvement methods were local loops. A single answer is critiqued and revised. A failed code sample is debugged. A reflection is stored in memory. These loops are important because they demonstrated that models could use their own outputs as material for improvement. But they are not the end state of the field. Recent work increasingly integrates multiple loops into systems that improve across episodes, across components, or across populations.

A useful progression begins with output-level refinement. Self-Refine improves a candidate output through generate–critique–refine iterations [36]. Reflexion improves future attempts by storing verbal lessons from prior failures [49]. SelfEvolve improves code by combining self-generated knowledge with interpreter feedback [25]. These systems modify the context or artifact of a task, not necessarily the agent's source code or weights. They show that feedback can be represented linguistically and reused.

The next step is component-level optimization. Agent Symbolic Learning treats prompts, tools, and pipelines as learnable symbolic weights, allowing a deployed agent to adjust internal components after observing trajectories [1]. DSPy-style prompt compilation, textual backpropagation, and multi-agent prompt evolution belong to the same family. Here the object being improved is not just an answer but the agent's procedure for producing answers. The boundary between inference and learning becomes less clear: the agent can execute tasks while updating symbolic control parameters.

A third step is architecture- and code-level evolution. ADAS uses a meta-agent to write new agents [21]. Gödel Agent modifies runtime logic [72]. DGM evolves agent implementations through an archive [76]. AlphaEvolve evolves algorithmic code at scale [40]. These systems treat software as the genome of intelligence. The genome is readable and editable by a foundation model; evaluation determines which variants survive. This perspective makes agent design cumulative and potentially open-ended.

A fourth step is weight- and curriculum-level self-improvement. RISE trains a model to introspect and revise its reasoning through multi-turn reinforcement learning [43]. RAGEN adapts RL to multi-turn agents while warning about echo traps [59]. Absolute Zero removes external training data by letting the model propose tasks and solve them under verifiable rewards [77]. ReVeal co-optimizes code generation and self-verification [27]. These methods modify neural weights, task curricula, or verification behaviors. They suggest that self-evolution can happen not only at the

prompt or code layer but also inside the model's learned policy.

A fifth step is lifecycle-level automation. AI Scientist attempts to automate the scientific research lifecycle, from idea generation to experiment execution and paper writing [35]. Production agent frameworks such as AutoGen, LangGraph, Letta/MemGPT, OpenHands, SWE-Agent, CrewAI, MetaGPT, DSPy, OpenEvolve, and related projects vary widely in their degree of self-evolution, but collectively they show how research loops are moving into usable engineering ecosystems. Some frameworks remain mostly static; others support memory evolution, graph modification, programmatic agent control, or evolutionary code search. The open-source ecosystem is therefore both a deployment substrate and a measurement challenge: popularity, stars, and hype do not necessarily correlate with genuine self-evolving capability.

The background thus motivates a new survey category. AI Self-Evolution is not equivalent to LLM prompting, not reducible to AutoML, and not identical to reinforcement learning. It is a cross-layer design pattern in which systems modify themselves through feedback. The rest of this chapter defines that pattern precisely.

1.2 Core Definition: Formalizing AI Self-Evolution

We define **AI Self-Evolution** as follows:

> *An AI system S is self-evolving if it can modify one or more of its own operational components—including prompts, memories, tools, code, architecture, data curriculum, or weights—through feedback-driven loops in order to improve performance on target tasks, subject to an acceptance mechanism that determines whether candidate modifications are retained.*

This definition emphasizes five elements: a system boundary, mutable components, feedback, a modification operator, and retention. The system boundary specifies what counts as "self." Mutable components specify what may change. Feedback supplies evidence about success or failure. The modification operator proposes changes. Retention decides whether the change becomes part of the future system. Without feedback, change is drift rather than evolution. Without retained modification, the system may adapt within an episode but does not accumulate improvement. Without a system boundary, every external human update would count as self-evolution, making the concept meaningless.

Formally, let an AI system be represented as

$$S_t = (\mathcal{C}_t, \mathcal{M}_t, \mathcal{E}, \mathcal{U}, \mathcal{G}), \tag{1}$$

where $\mathcal{C}_t$ denotes mutable components at time t, $\mathcal{M}_t$ denotes memory or state, $\mathcal{E}$ denotes an environment or task distribution, $\mathcal{U}$ denotes an objective or utility criterion, and $\mathcal{G}$ denotes governance constraints such as safety policies, sandbox limits, or rollback rules. The mutable component set may include

$$\mathcal{C}_t = \{P_t, R_t, T_t, A_t, K_t, D_t, W_t\}, \tag{2}$$

where P_t are prompts or instructions, R_t are reasoning routines, T_t are tools and tool descriptions, A_t is agent architecture or control flow, K_t is source code or executable implementation, D_t is data or curriculum, and W_t are neural weights.

During an evolution episode, the system interacts with tasks $x \sim \mathcal{E}$, produces trajectories τ_t, receives feedback F_t, proposes a modification Δ_t, and updates its components if an acceptance rule

approves:

$$\tau_t = \text{Run}(S_t, x), \tag{3}$$
$$F_t = \text{Feedback}(\tau_t, x, \mathcal{U}), \tag{4}$$
$$\Delta_t = \text{Modify}(S_t, F_t, \mathcal{M}_t), \tag{5}$$
$$S_{t+1} = \text{Accept}(S_t, \Delta_t, F_t, \mathcal{G}). \tag{6}$$

The modification may be a prompt edit, a memory entry, a tool update, a code patch, an architecture change, a generated training task, or a weight update. The acceptance function may be deterministic, such as keeping a patch only if tests pass, or probabilistic, such as selection in an evolutionary population. It may also include human approval, but the system must contribute materially to proposing or evaluating the change.

A performance-improvement objective can be written as

$$\mathbb{E}_{x\sim\mathcal{E}_{\text{target}}}\left[U(S_{t+1}, x)\right] \;>\; \mathbb{E}_{x\sim\mathcal{E}_{\text{target}}}\left[U(S_t, x)\right] - \epsilon, \tag{7}$$

where ϵ allows for noisy evaluation and exploration. In strict exploitation settings, we require expected improvement. In open-ended settings, however, a system may retain diverse variants that do not immediately dominate the parent but improve coverage, novelty, or future evolvability. For archive-based systems, the objective becomes multi-dimensional:

$$\max_{S\in\mathcal{A}} \left(f(S),\ n(S),\ r(S),\ c(S)^{-1}\right), \tag{8}$$

where $\mathcal{A}$ is an archive, f is task fitness, n is novelty or diversity, r is robustness, and c is computational cost. This form captures quality-diversity methods such as MAP-Elites [39] and their modern use in AlphaEvolve [40].

The phrase "its own components" should be interpreted operationally, not metaphysically. A self-evolving agent need not be conscious, autonomous in a legal sense, or free from human-designed constraints. It must merely have a loop by which the system that performs tasks is modified by a process in which that system, or a tightly coupled meta-system, participates. ADAS, for instance, uses a meta-agent to design object-level agents [21]. Whether this is one system or two depends on the boundary. We count it as self-evolution when the meta-agent, archive, evaluator, and generated agents form a persistent system whose future designs are conditioned on its own search history. Similarly, Reflexion's actor, evaluator, and reflection model can be viewed as a composite system whose memory updates alter future behavior [49].

This boundary-based view avoids two extremes. On one extreme, every training procedure could be called self-evolution because it improves a model. That is too broad. Standard supervised learning updates weights by an externally specified optimizer over a fixed dataset; the model does not propose changes to its own learning process or components. On the other extreme, only systems that rewrite their own source code would qualify. That is too narrow. Modern AI behavior is often controlled by prompts, retrieval state, memories, tools, and workflow graphs. If those components are modified by feedback-driven loops, the system has evolved at the level that matters for behavior.

We therefore distinguish levels of self-evolution:

1. **Output-level evolution**: the system revises a candidate output within a task episode, as in generate–critique–refine loops.

2. **Memory-level evolution**: the system stores feedback, reflections, or experiences that alter future behavior.

3. **Prompt/tool-level evolution**: the system edits prompts, tool descriptions, or symbolic parameters.
4. **Architecture-level evolution**: the system changes control flow, modular composition, or agent topology.
5. **Code-level evolution**: the system writes or patches executable source code implementing itself or its descendants.
6. **Data/curriculum-level evolution**: the system generates or selects tasks that shape future learning.
7. **Weight-level evolution**: the system updates neural parameters using feedback from self-generated or self-mediated experience.
8. **Population-level evolution**: the system maintains multiple variants and selects among them over time.

These levels are not mutually exclusive. ReVeal combines code generation, self-verification, tool feedback, and reinforcement learning [27]. Absolute Zero combines task generation, solving, executable verification, and weight updates [77]. DGM combines code-level modification with population-level archiving [76]. AI Scientist combines idea generation, experiment code, paper writing, and review loops [35]. A mature self-evolving system will likely contain several levels simultaneously.

Feedback is the central resource. It can be internal, external, symbolic, scalar, textual, executable, or social. Internal feedback includes self-critique and introspection. External feedback includes unit tests, environment rewards, benchmark scores, human judgments, peer-review scores, and market signals. Symbolic feedback includes language gradients and reflection notes. Scalar feedback includes rewards, pass/fail signals, and performance metrics. Executable feedback includes stack traces, assertions, and verifier outputs. Social feedback includes user corrections and community adoption. The reliability of self-evolution depends heavily on the reliability and alignment of this feedback. A weak evaluator produces weak evolution; a manipulable evaluator selects for manipulation.

The modification operator is equally important. In classical evolution, mutation may be random; in gradient descent, the update direction is computed from differentiable loss; in AutoML, search operators are hand-designed or controller-generated. In modern self-evolving AI, LLMs often implement the modification operator:

$$\Delta_t = \mathrm{LLM}_\phi(\mathrm{serialize}(S_t), F_t, H_t), \tag{9}$$

where H_t is history, archive context, or instructions. This operator is powerful because it can make semantically coherent edits across heterogeneous artifacts. It is dangerous because it is stochastic, prompt-sensitive, and capable of introducing hidden failures. The same LLM that patches a bug can remove a guardrail; the same agent that improves a benchmark can overfit to its public tests.

Retention mechanisms define whether improvements accumulate. A naive system may overwrite itself after every self-critique, producing drift. A more robust system evaluates candidate changes on validation tasks, compares against baselines, stores rollback points, and preserves diversity. Reflexion's memory buffer retains reflections but must manage context limits [49]. Gödel Agent accepts modifications only after validation and can roll back failed patches [72]. DGM stores variants in an archive rather than a single mutable lineage [76]. AlphaEvolve uses quality-diversity selection to retain elites in behavioral niches [40]. Retention is where learning becomes evolution.

Finally, self-evolution must be evaluated over time. A static benchmark score after one update is insufficient. We must ask whether the system improves monotonically or cyclically, whether gains transfer across tasks, whether it maintains safety constraints, whether it collapses into repetitive behavior, whether it overfits to evaluators, and whether the cost of evolution is justified by the benefit. For an evolution trace $S_0, S_1, \ldots, S_T$, relevant metrics include cumulative improvement,

$$\Delta_T = U(S_T) - U(S_0), \tag{10}$$

per-step efficiency,

$$\eta_T = \frac{U(S_T) - U(S_0)}{\sum_{t=0}^{T-1} c_t}, \tag{11}$$

and robustness under held-out distributions,

$$\rho_T = \mathbb{E}_{x \sim \mathcal{E}_{\text{heldout}}}[U(S_T, x)] - \mathbb{E}_{x \sim \mathcal{E}_{\text{train}}}[U(S_T, x)]. \tag{12}$$

These metrics foreshadow the evaluation chapter of the survey, where we argue that self-evolving systems require benchmarks that measure trajectories, not just endpoints.

1.3 Related Concepts and Boundaries

AI Self-Evolution overlaps with several established fields, but it is not identical to any of them. This subsection draws boundaries because the term "self-improvement" is often used loosely. A system may improve after deployment, learn online, adapt to a user's preferences, tune hyperparameters, or train on self-supervised losses without satisfying the stronger criteria of self-evolution. Conversely, a system may be self-evolving even if its neural weights remain fixed, provided that feedback-driven changes to prompts, memory, tools, code, or architecture alter future behavior.

Continual learning. Continual learning studies models that learn from a sequence of tasks or data distributions without catastrophic forgetting. Its central questions include stability-plasticity trade-offs, replay, regularization, task inference, and lifelong adaptation. A continual learner updates parameters over time, but the update rule is usually fixed by the designer. The model does not necessarily inspect failures, propose new tools, rewrite its architecture, or choose its own curriculum. Continual learning becomes self-evolution when the system participates in modifying its own learning machinery or task interface, not merely when more data are streamed through a fixed optimizer.

Online learning. Online learning updates predictions as examples arrive. It is often formalized with regret bounds against a comparator. The learner receives an input, predicts, observes loss, and updates. This is feedback-driven, but the feedback is typically used by a fixed algorithm such as stochastic gradient descent, multiplicative weights, or bandit updates. Self-evolution requires an additional layer: the system may change the representation, prompt, code, architecture, or update rule itself. RISE and RAGEN are closer to self-evolution than generic online learning because they learn multi-turn introspection or agent policies, but even they must be analyzed by whether the learned behavior changes future self-modification capacity [43, 59].

Self-supervised learning. Self-supervised learning constructs labels from data itself, as in masked language modeling, contrastive learning, or next-token prediction. It is "self" in the sense that supervision is derived automatically, not manually annotated. It is not necessarily self-evolving

because the model does not modify its own operational components through feedback about task performance. Absolute Zero is an instructive contrast: it generates tasks and solutions, verifies them with a code executor, and updates via reinforcement learning [77]. This is more than self-supervision because the model shapes its own curriculum under verifiable rewards.

Meta-learning. Meta-learning trains systems that learn how to learn across tasks. It is closely related to self-evolution because it concerns adaptation mechanisms. However, much meta-learning remains offline: a meta-learner is trained on task distributions and then adapts using a fixed inner-loop procedure. Self-evolution emphasizes post-deployment or within-system modification of components, often including symbolic and code artifacts not captured by differentiable meta-learning. ADAS can be interpreted as meta-learning over agent designs, but it is also self-evolutionary because the search history and generated agents condition future designs [21].

AutoML and NAS. AutoML automates model design, training, and selection. NAS searches architecture spaces. These fields are direct predecessors, but most AutoML pipelines are external optimizers. A human invokes the optimizer, waits for a candidate model, and deploys it. AI Self-Evolution internalizes part of that optimizer into the system lifecycle. When a meta-agent designs new agents, when an archive of agent implementations accumulates, or when a coding system mutates its own algorithms under evaluators, AutoML becomes agentic and self-referential.

Reinforcement learning. Reinforcement learning optimizes policies through reward signals. It is foundational for self-evolution, especially in RISE, RAGEN, Absolute Zero, and ReVeal [27, 43, 59, 77]. Yet RL alone does not imply self-evolution. A policy trained by PPO on environment rewards may become more competent, but it does not necessarily modify its prompts, tools, code, architecture, or curriculum. Self-evolution appears when the policy controls or participates in the modification process, such as proposing tasks, generating tests, revising reasoning, or altering agent software.

RLHF and preference optimization. Reinforcement learning from human feedback aligns models to preferences [12, 80]. It uses feedback, but the process is usually externally orchestrated: humans label data, engineers train reward models, and training pipelines update weights. Constitutional AI and self-critique methods move closer to self-evolution by using model-generated critiques and revisions under a constitution [5]. Still, the distinction depends on whether the deployed system can continue modifying its own components under feedback rather than merely being trained once with synthetic preferences.

Agent memory and personalization. Many agent frameworks store memories, user preferences, or conversation summaries. Memory can be self-evolutionary if it is generated from feedback and retained to improve future behavior. Letta/MemGPT-like memory systems are therefore partial self-evolution mechanisms. But memory alone is not enough. A logging system that records transcripts without using them for future modification is not self-evolving. A recommender that stores user preferences may be adaptive, but if its update rule and task interface are fixed, it occupies a narrower category.

Program synthesis and automated repair. Program synthesis generates code from specifications; automated program repair patches bugs according to tests. Modern LLM code agents blur this boundary. SelfEvolve, ReVeal, AlphaEvolve, and DGM all use program synthesis or repair as

Concept	Primary object of change	Typical feedback	Difference from AI Self-Evolution
Continual learning	Neural parameters over task streams	New data, task labels, replay losses	Usually uses a fixed update rule; may not modify prompts, tools, code, architecture, or curriculum.
Online learning	Predictor state after each example	Immediate loss or reward	Focuses on regret under fixed algorithms; self-evolution can change the algorithmic machinery itself.
Self-supervised learning	Representations and weights	Automatically generated labels	Supervision is automatic, but the system does not necessarily evaluate and modify its own operational components.
Meta-learning	Adaptation procedure or initialization	Distribution of training tasks	Often learned offline; self-evolution emphasizes runtime feedback loops and retained system modifications.
AutoML/NAS	Model pipelines or architectures	Validation performance	Usually an external design optimizer; becomes self-evolutionary when integrated into the agent/system lifecycle.
Reinforcement learning	Policy parameters	Environment rewards	RL is an optimization substrate; self-evolution additionally concerns self-modification of components and procedures.
RLHF/preference tuning	Model behavior and alignment	Human or AI preference labels	Usually externally orchestrated; self-evolution requires ongoing system-mediated modification.
Agent memory	Episodic or semantic context	User interactions, reflections	Partial self-evolution if memory is feedback-derived and behaviorally active; otherwise only logging/personalization.
Program synthesis/repair	Task-specific code	Specifications, tests, errors	Self-evolution when generated or repaired code changes the future agent/system, not just a one-off output.

Table 1: AI Self-Evolution in relation to neighboring concepts. The boundaries are porous: many modern systems combine several rows, but self-evolution specifically requires feedback-driven modification of the system's own operational components.

part of self-evolution [25, 27, 40, 76]. The distinguishing feature is recursive use: code generation is not only used to solve an external programming task but also to improve the system's own future problem-solving procedure.

The comparison in Table 1 suggests a general rule. If the system merely changes its output, it is adaptation. If it changes an internal state that affects future outputs, it may be learning. If it changes the components or procedures by which it learns, reasons, acts, or designs successors, under feedback and retention, it is self-evolution. This rule captures both lightweight prompt/memory methods and heavyweight code/weight/population methods.

Several borderline cases deserve attention. First, an LLM using chain-of-thought prompting does not self-evolve if each response is independent. If the chain is critiqued, revised, stored, and reused, it may become output- or memory-level self-evolution. Second, a human editing an agent's prompt after observing a failure is not AI self-evolution, though it may be part of a human-in-the-loop evolution system. If the agent proposes the edit and the human only approves it, the system is partially self-evolving. Third, a benchmark-tuned model release is not self-evolving after deployment, but the training organization may implement an external evolution pipeline. Our survey focuses on systems whose loop is technically represented and can be analyzed as part of the AI system.

Another boundary concerns autonomy. A self-evolving system need not be fully autonomous. Human approval, safety filters, sandboxing, and governance are often desirable. In fact, unrestricted autonomy is a safety risk. We therefore define self-evolution by participation in modification, not by absence of human oversight. A system that drafts code patches for itself, evaluates them in a sandbox, and requests approval before deployment still exhibits self-evolutionary structure. The acceptance function includes humans as part of governance $\mathcal{G}$.

A final boundary concerns improvement. Evolutionary systems can regress. A single failed modification does not disqualify a system, just as a biological lineage can accumulate deleterious mutations. What matters is that the system contains a feedback-driven selection process intended to improve performance or diversity over time. Evaluation should measure the trajectory, including failures, rather than assume monotonic progress.

1.4 Contributions of This Survey

This survey makes five main contributions. First, we introduce a **Five Evolution Loops** framework that organizes the field by the mechanism through which feedback becomes retained change. Second, we analyze more than nineteen research and open-source systems, separating genuine self-evolution mechanisms from static agent orchestration and hype-driven adoption. Third, we unify work across AutoML/NAS, LLM self-improvement, evolutionary computation, reinforcement learning, program synthesis, and agent frameworks under a common formal abstraction. Fourth, we map the researcher and institution network that produced the field, highlighting the convergence of the evolutionary-computation lineage, the language-agent reasoning lineage, and the LLM self-improvement lineage. Fifth, we introduce the **Evolve-AGI Index**, an evidence-to-index instrument that measures field maturity through benchmark performance, loop strength, evidence-chain credibility, transfer verification, implementation access, field momentum, and governance readiness.

1.4.1 Contribution 1: The Five Evolution Loops framework

The first contribution is a framework that classifies self-evolving AI by the loop that performs modification and selection. We use five loops because they capture the dominant mechanisms in the literature while remaining general enough to compare hybrid systems.

Loop I: Reflexive output and memory evolution. The system improves by critiquing outputs, storing feedback, and conditioning future attempts. Self-Refine and Reflexion are representative [36, 49]. The modified component is usually the current output, an episodic memory, or a reflection buffer. The evaluator may be the same model, an external reward model, tests, or an environment. This loop is low-cost and easy to deploy, but it is limited by context length, self-critique quality, and evaluator reliability.

Loop II: Symbolic component evolution. The system treats prompts, tools, workflows, and pipelines as learnable symbolic parameters. Agent Symbolic Learning is the central example: it defines language losses, language gradients, and optimizers for prompt, tool, and pipeline nodes [1]. DSPy-like prompt compilation and textual backpropagation also fit here. This loop bridges gradient-based learning and agent engineering. It makes the agent's design surface optimizable without requiring neural weight updates.

Loop III: Verification-driven code evolution. The system generates, tests, repairs, and retains code. SelfEvolve uses self-generated knowledge plus interpreter errors [25]; ReVeal co-

Loop	Mutable component	Representative systems	Typical failure modes
Reflexive output and memory	Outputs, reflections, memory buffers	Self-Refine, Reflexion, ReAct-style agents	Generic critiques, memory bloat, context overfitting, local optima.
Symbolic component	Prompts, tools, pipelines, workflow nodes	Agent Symbolic Learning, DSPy, EvoMAC-like textual optimization	Prompt overfitting, unstable language gradients, high LLM-call cost.
Verification-driven code	Candidate code, tests, repair patches	SelfEvolve, ReVeal, AlphaEvolve, code agents	Test overfitting, sandbox escape risk, verifier incompleteness.
Architecture and agent design	Agent code, control flow, tool topology	ADAS, Gödel Agent, DGM	Unsafe self-modification, irreproducible search, expensive evaluation.
Curriculum, weight, and population	Training tasks, policy weights, archives	RISE, RAGEN, Absolute Zero, DGM, AlphaEvolve	Reward hacking, echo trap, mode collapse, compute cost.

Table 2: The Five Evolution Loops framework used throughout this survey. The loops classify mechanisms rather than mutually exclusive systems.

optimizes generation and self-verification [27]; AlphaEvolve uses automated evaluators to select algorithmic programs [40]. This loop is powerful because code can be mechanically checked. Its limitation is domain dependence: reliable verifiers are easier for programming, mathematics, and simulation than for open-ended social tasks.

Loop IV: Architecture and agent-design evolution. The system searches over agent architectures, control flow, tool use, and self-modifying code. ADAS, Gödel Agent, and DGM are representative [21, 72, 76]. The modified component is the agent itself. This loop is closest to the intuitive idea of self-redesign. It can discover strategies not anticipated by humans, but it raises cost, safety, and reproducibility challenges.

Loop V: Curriculum, weight, and population evolution. The system changes the data distribution, training curriculum, neural weights, or population archive. RISE, RAGEN, Absolute Zero, DGM, and AlphaEvolve all instantiate parts of this loop [40, 43, 59, 76, 77]. The loop can produce deeper behavioral changes than prompt-only methods, but it requires careful reward design and protection against collapse. Absolute Zero's dual role of task proposer and solver is especially important because it removes the assumption of human-curated training data.

The loops can be summarized as

$$\text{SelfEvolution} = \text{Observe} \rightarrow \text{Interpret} \rightarrow \text{Modify} \rightarrow \text{Verify} \rightarrow \text{Retain}, \tag{13}$$

with each loop differing in the representation modified and the verifier used. The framework is not a taxonomy of papers; it is a taxonomy of mechanisms. A single paper may instantiate multiple loops. For example, DGM combines architecture/code evolution with population archives; ReVeal combines verification-driven code evolution with weight-level RL; AI Scientist combines idea, code, manuscript, and review loops.

1.4.2 Contribution 2: Analysis of research systems and open-source projects

The second contribution is an evidence-oriented analysis of more than nineteen systems. The research core includes Self-Refine, Reflexion, Agent Symbolic Learning, ADAS, Gödel Agent, Darwin Gödel Machine, SelfEvolve, RISE, RAGEN, Absolute Zero, ReVeal, AlphaEvolve, FunSearch,

AI Scientist, STaR, Constitutional AI, Tree of Thoughts, ReAct, SWE-agent [67], and related software-engineering agents. The open-source and framework ecosystem includes DSPy, AutoGen, LangGraph, Letta/MemGPT, CrewAI, MetaGPT, OpenHands, OpenEvolve, AutoGPT, smolagents, and project-specific implementations of the papers above.

Our analysis emphasizes mechanism rather than popularity. The star-analysis report used for this survey shows that GitHub stars can be a poor proxy for technical quality. AutoGPT accumulated massive attention, but its star-quality score was estimated far below smaller research-oriented projects. OpenEvolve, DSPy, SWE-Agent, OpenHands, LangGraph, AutoGen, CrewAI, and MetaGPT exhibit different propagation patterns: academic, steady, hype-driven, or community-driven. This matters because self-evolution is a technical property, not a marketing label. A project may be popular because it is easy to demo, while a quieter project may contain the more important self-evolution mechanism.

We therefore examine each system along dimensions such as mutable component, feedback source, retention mechanism, evaluation benchmark, openness, safety controls, and transfer. Reflexion is strong evidence for memory-level verbal feedback because it reports large HumanEval and interactive-task gains under external evaluation [49]. Self-Refine demonstrates that same-model critique can improve diverse tasks by roughly 20% on average, but it also illustrates the limits of self-feedback without external verification [36]. Agent Symbolic Learning reports improvements on HotPotQA, MATH, HumanEval, software-development executability, and creative writing by optimizing symbolic agent components [1]. DGM reports striking software-engineering improvements through open-ended code-level evolution [76]. AlphaEvolve reports algorithmic and infrastructure improvements under automated evaluation [40]. Absolute Zero demonstrates that self-generated curricula can support zero-data reasoning improvements [77].

The open-source ecosystem is equally important for understanding adoption. AutoGen supports multi-agent conversation and human-in-the-loop workflows; LangGraph offers state graphs that can be modified or orchestrated programmatically; Letta/MemGPT emphasizes long-term memory; DSPy compiles prompts and demonstrations; SWE-Agent and OpenHands operationalize software-engineering agents; OpenEvolve adapts evolutionary coding ideas for the community. CrewAI and MetaGPT popularize role-based agent orchestration, but many deployments remain static unless combined with feedback-driven modification. Distinguishing static orchestration from self-evolution is one of the survey's practical aims.

1.4.3 Contribution 3: Cross-domain unification

The third contribution is a cross-domain unification. The same abstract loop appears under different names in different fields. In AutoML it is architecture search. In evolutionary computation it is variation and selection. In LLM prompting it is critique and refinement. In reinforcement learning it is policy improvement. In program synthesis it is generate-and-test. In software engineering it is automated repair. In agent frameworks it is memory and workflow adaptation. In AI-for-science it is hypothesis generation, experimentation, and review.

We formalize these domains using a shared tuple: representation, variation operator, feedback channel, retention rule, and governance constraint. For NAS, the representation is a neural architecture, variation is architecture mutation or controller sampling, feedback is validation accuracy, retention is selection, and governance is compute budget. For Reflexion, the representation is memory, variation is verbal reflection, feedback is evaluator reward, retention is memory update, and governance is context management. For DGM, the representation is agent source code, variation is LLM patching, feedback is benchmark performance, retention is archive insertion, and governance is sandboxing and selection. For Absolute Zero, the representation includes task curriculum and

weights, variation is self-generated tasks and solutions, feedback is code execution, retention is RL update, and governance is verifier design.

This unification reveals shared bottlenecks. Search cost appears in NAS, ADAS, DGM, and AlphaEvolve. Evaluator reliability appears in Self-Refine, Reflexion, ReVeal, AI Scientist, and RLHF. Diversity maintenance appears in evolutionary computation, DGM, AlphaEvolve, and open-ended science. Reward hacking appears in RL, code generation, and benchmark-driven agent design. Safety constraints appear wherever code or tools can be modified. The terminology differs, but the engineering problems rhyme.

The unification also reveals transfer opportunities. NAS developed weight sharing and benchmark suites such as NAS-Bench; self-evolving agents need analogous reusable evaluation landscapes. Quality-diversity algorithms developed archives and novelty metrics; code-evolving agents can use them to avoid local optima. Program repair developed test-generation and patch-validation methods; self-modifying agents need them for safe code evolution. RL developed off-policy evaluation and reward-model diagnostics; self-play and self-verification methods need them to detect collapse. Software engineering developed CI/CD, sandboxing, and rollback; self-evolving AI needs these as governance primitives.

1.4.4 Contribution 4: Researcher network mapping

The fourth contribution is a map of the researcher network. The field is young, but the sources reveal three converging lineages. The evolutionary-computation lineage runs from Risto Miikkulainen, Kenneth Stanley, Joel Lehman, and Jeff Clune through NEAT, novelty search, quality diversity, POET, AI-GAs, ADAS, DGM, and AlphaEvolve-related work [13, 21, 31, 52, 76]. The language-agent reasoning lineage runs through Karthik Narasimhan, Shunyu Yao, Noah Shinn, ReAct, Tree of Thoughts, Reflexion, SWE-bench, and SWE-agent [49,70,71]. The LLM self-improvement lineage includes Aman Madaan, Peter Clark, Sean Welleck, Eric Zelikman, and work on Self-Refine, STaR, self-rewarding models [74], and constitutional self-critique [5, 36, 75].

These lineages now overlap institutionally and methodologically. Jeff Clune's group appears in ADAS, DGM, AI Scientist, and open-ended evolutionary AI. Shengran Hu and Cong Lu connect ADAS and DGM; Cong Lu also appears in AI Scientist-related work. Shunyu Yao and Karthik Narasimhan connect ReAct, Reflexion, Tree of Thoughts, SWE-agent, and broader language-agent evaluation. AIWaves and Zhejiang University contribute Agent Symbolic Learning, which explicitly bridges agent frameworks and textual optimization. Google DeepMind contributes FunSearch and AlphaEvolve, scaling LLM-guided evolution to mathematics and infrastructure. Microsoft Research contributes ReVeal's verification-centered code-agent RL. Tsinghua and LeapLabTHU contribute Absolute Zero's zero-data self-play reasoning.

Mapping the network is not merely sociological. It explains why ideas appear together. DGM's archive design makes sense against Clune's open-endedness background. Reflexion's memory-based verbal RL fits the Princeton/Northeastern language-agent lineage. Agent Symbolic Learning reflects industry-academic interest in deployable agent frameworks. AlphaEvolve reflects DeepMind's AI-for-science and systems-optimization agenda. AI Scientist reflects Sakana AI's nature-inspired, open-ended automation focus. Understanding these lineages helps researchers locate missing bridges, such as stronger connections between formal program repair and agent self-modification, or between continual-learning theory and prompt/tool evolution.

1.4.5 Contribution 5: Evidence graph and Evolve-AGI Index

The fifth contribution is to treat the survey itself as an evolving evidence graph rather than a static bibliography. Each paper, repository, benchmark, project report, user pain point, and production signal should be indexed by the same questions used throughout the survey: what object evolves, what feedback drives the update, what operator proposes change, what evaluator approves it, what evidence shows transfer, what it costs, and what governance boundary prevents harm. This evidence graph is the bridge between a qualitative survey and a reusable field instrument.

The Evolve-AGI Index operationalizes that bridge. It is not an AGI capability score and should not be read as a claim that any single system has reached general intelligence. Instead, it measures whether the self-evolving-agent field is becoming more benchmark-visible, loop-complete, evidence-grounded, transferable, runnable, active, and governable. The index combines seven signals: benchmark performance, core loop strength, evidence-chain credibility, transfer and verification, implementation access, field momentum, and governance readiness. This makes the paper's central thesis measurable: self-evolution matures when controlled improvement loops become independently verifiable and reusable, not when a project accumulates stars or a demo looks autonomous.

This contribution also changes how the repository supporting the survey is organized. Raw materials remain immutable evidence. Processed analyses convert them into classifications, model cards, rankings, and benchmark tables. The README becomes a survey-result entry point rather than a full link dump. The website and graph expose the same evidence as navigable public artifacts. In this sense, the survey is not only about self-evolving agents; it is maintained as a small self-evolving knowledge system whose claims can be refreshed, audited, and revised as the field changes.

1.4.6 Running examples and evidence used throughout the survey

To make the survey concrete, we use several running examples that span the five loops. These examples are not chosen merely because they are well known; they are chosen because each exposes a different design decision in self-evolving AI. Self-Refine demonstrates the minimal loop: no external training, no explicit memory across tasks, and no code modification, only iterative critique and revision inside a task episode [36]. Reflexion adds an explicit memory buffer and an evaluator, turning feedback from one attempt into context for the next [49]. Agent Symbolic Learning moves from episodic improvement to component optimization by representing an agent as a symbolic network with prompts, tools, and pipeline nodes [1]. ADAS then makes the architecture itself the search object, asking a meta-agent to write new agent programs [21]. DGM extends that search into an open-ended archive of self-improving agent implementations [76]. AlphaEvolve scales the same broad idea to industrial algorithm discovery and systems optimization [40]. Absolute Zero and ReVeal show how self-evolution can also alter the training distribution and policy weights through verifiable rewards [27, 77].

These examples reveal that self-evolution is not a single point on a capability ladder. A prompt-level loop may be more reliable than a code-level loop in settings where feedback is subjective and safety constraints are tight. A code-level loop may be more powerful in programming tasks because unit tests provide crisp feedback. A weight-level loop may produce deeper generalization, but it is slower, more expensive, and more vulnerable to reward misspecification. An archive-based loop may be the best way to avoid local optima, yet it complicates deployment because the system no longer has one canonical state. The survey therefore avoids ranking methods by apparent sophistication alone. Instead, it asks which loop is appropriate for which task, verifier, cost regime, and governance requirement.

Consider Self-Refine as a first running example. Its generate–critique–refine structure is simple enough to be implemented with a single model and a prompt template. The method reported improvements across code optimization, mathematical reasoning, sentiment reversal, dialogue, poetry, readability, and constrained generation [36]. Its importance lies less in any one benchmark number than in the demonstration that a foundation model can use its own natural-language feedback as an optimization signal. But Self-Refine also exposes a core weakness: the same model that produced an error may fail to diagnose it. Without an external evaluator, self-critique can become rationalization. In our framework, Self-Refine is therefore a high-accessibility, low-assurance instance of Loop I.

Reflexion strengthens the loop by separating actor, evaluator, and self-reflection roles [49]. The actor attempts a task; the evaluator assigns success or failure; the reflection model writes a verbal lesson; memory carries that lesson forward. On code and interactive tasks, this architecture can produce large gains because failures are turned into reusable advice. Reflexion's reported HumanEval result, along with improvements on AlfWorld and WebShop, made verbal reinforcement learning a central paradigm. Yet Reflexion also makes visible the memory-management problem. If every failure becomes a memory, context grows; if memories are summarized too aggressively, useful detail is lost; if reflections are generic, they add tokens without adding guidance. Later systems inherit this trade-off whenever they store textual feedback.

Agent Symbolic Learning is our main running example for Loop II because it changes the object of optimization from answers or memories to the symbolic components of an agent [1]. The method's language loss and language gradients are analogies to differentiable learning, but the analogy is operationally useful. A failing trajectory can be converted into a textual diagnosis; the diagnosis can be backpropagated through prompt, tool, and pipeline nodes; each node can then be updated by an LLM optimizer. The reported gains on HotPotQA, MATH, HumanEval, software-development executability, and creative writing suggest that agent components can be optimized in a structured way. At the same time, the method raises new questions: What is a learning rate in language space? How stable are language gradients? How should conflicting textual gradients be aggregated? How do we prevent a prompt optimizer from overfitting to a small development set? These questions connect symbolic agent optimization to the older theory of optimization, but they do not reduce to it.

ADAS is our running example for automated agent-design search [21]. It treats Python programs implementing agents as the search space. This matters because the space is not just large; it is expressive enough to encode arbitrary computation. A meta-agent can invent new decomposition procedures, tool-use policies, memory layouts, voting schemes, verification routines, or recursive calls. The promise is clear: human designers no longer need to enumerate all plausible agent architectures. The risk is equally clear: generated agents may be brittle, unsafe, or hard to interpret. ADAS reported that automatically discovered agents can outperform hand-designed baselines and transfer across domains or model backbones. The survey uses ADAS to discuss when search over agent code is better than manual engineering and when the evaluation cost becomes prohibitive.

Gödel Agent is a running example for runtime self-reference [72]. Instead of maintaining a population of separate descendants, it lets an agent inspect and modify its own logic through monkey patching. This design is close to the intuitive notion of a self-rewriting agent. It also brings software-engineering hazards to the foreground. Runtime patching can introduce nondeterminism, hidden state dependencies, portability problems, and rollback complexity. The method uses validation and rollback, but the general lesson is broader: as self-evolution moves from prompts to executable code, the system must adopt practices from secure software deployment. Patches need provenance, tests, sandboxing, audit logs, and clear acceptance criteria.

DGM is our running example for open-ended agent evolution [76]. Its archive converts self-

improvement from a single lineage into a population process. This is crucial because one agent variant may improve tool use while another improves planning, and a third may improve error recovery. A single greedy lineage could discard variants that are not immediately best but contain useful building blocks. The archive preserves diversity and supports cumulative innovation. DGM's reported improvement on SWE-bench and Polyglot makes it an important empirical milestone. The survey uses DGM to analyze archive sampling, diversity bonuses, cross-domain transfer, and the difference between improving an agent's score and improving the evolvability of the agent population.

AlphaEvolve is our running example for quality-diversity search with strong industrial evaluation [40]. It combines multiple Gemini models, automated evaluators, and MAP-Elites-style retention. Its reported results on matrix multiplication, data-center scheduling, chip design, and attention kernels show why code and algorithms are attractive domains for self-evolution: candidates can be executed, measured, compared, and retained. Yet AlphaEvolve also illustrates the infrastructural barrier. Many domains lack reliable, cheap, automated evaluators. A system that can improve a kernel by 23% under a precise performance benchmark may not be able to improve a policy argument, a medical decision, or a social interaction with the same confidence. The availability of verifiers is thus a central axis of the field.

Absolute Zero is our running example for self-generated curricula [77]. The model proposes tasks, solves them, and receives verification from a code executor. This dual role resembles self-play in games, but with programming and reasoning tasks replacing board positions. The paradigm is important because human-curated data may become a bottleneck for further improvement. If a model can generate challenging, solvable, diverse tasks, it can create its own training signal. But the same setup risks triviality, unsolvable tasks, mode collapse, or curriculum myopia. The proposer and solver can collude in a degenerate region of task space unless rewards encourage both difficulty and solvability. Absolute Zero therefore highlights the governance problem inside the training loop itself.

ReVeal is our running example for reliable self-verification in code agents [27]. It treats verification not as a post-hoc tool call but as a capability that should be optimized along with generation. This is an important shift. Many agents can write code; fewer can design effective tests, interpret failures, and decide when to stop. ReVeal's turn-level credit assignment and generation-verification co-evolution point toward agents that learn how to use tools for self-correction. The broader lesson is that self-evolution depends not only on better generators but also on better critics, verifiers, and credit assignment.

AI Scientist is a running example for lifecycle-level self-evolution [35]. It attempts to automate a full research cycle: idea generation, novelty checking, experiment design, code execution, manuscript writing, and review. It broadens the survey beyond coding agents and prompt optimizers. In scientific discovery, the object being improved may be a hypothesis, experiment, paper, or research agenda. The feedback may be experimental results, novelty scores, or peer-review rubrics. AI Scientist also demonstrates the limits of current systems: generated papers may lack theoretical depth, reviewers may be unreliable, and domains may be narrow. These limitations are not incidental; they show that long-horizon self-evolution requires reliable intermediate evaluations at every stage of a complex workflow.

Together, these running examples create a ladder from local revision to open-ended system redesign. However, the ladder should not be read as a chronology in which later methods simply replace earlier ones. In practice, strong systems will compose loops. A DGM-style agent may use Reflexion-like memory during evaluation. An ADAS-generated architecture may include Self-Refine subroutines. A ReVeal-trained code model may be embedded inside AlphaEvolve-like population search. An AI Scientist may use Agent Symbolic Learning to optimize its review prompts and

DGM-like archives to preserve successful experimental strategies. The survey's central claim is that the future field will be compositional: self-evolution will be an architecture of interacting loops.

1.4.7 Why the field needs a survey now

The need for a survey arises from conceptual fragmentation. The same mechanism is being rediscovered under different labels: self-refinement, verbal reinforcement learning, recursive introspection, agent symbolic learning, automated design of agentic systems, code evolution, self-play reasoning, reliable self-verification, open-ended discovery, and AI scientist automation. Without a common vocabulary, it is difficult to compare claims. A paper may report that an agent "self-improves" when it only revises outputs within a single context window. Another may claim "self-evolution" because it trains on self-generated data, even though the deployed system cannot modify its own components. A third may evolve code in a population but not address safety or governance. The survey provides a vocabulary for saying precisely what changed, who changed it, using which feedback, retained by which rule, and evaluated on which distribution.

The field also needs a survey because empirical results are difficult to compare. Self-Refine reports average improvements across diverse tasks; Reflexion reports code and interactive-task gains; Agent Symbolic Learning reports improvements across QA, math, code, software development, and creative writing; DGM reports SWE-bench and Polyglot improvements; AlphaEvolve reports scientific and infrastructure discoveries; Absolute Zero reports zero-data reasoning gains; ReVeal reports multi-turn code-agent improvements. These results do not share a benchmark suite, cost accounting method, safety protocol, or standard definition of an evolution step. A static model leaderboard cannot capture the difference between a one-time prompt refinement and a 1,000-iteration archive search. Survey-level synthesis is necessary before the community can build fair benchmarks.

A third reason is the gap between research and open-source adoption. Popular agent frameworks often advertise autonomy, memory, or multi-agent collaboration, but many deployments remain static. A role-based CrewAI or MetaGPT workflow may coordinate agents without modifying itself. AutoGen conversations may adapt through dialogue but not necessarily retain structural changes. LangGraph can represent dynamic workflows, yet the graph usually changes because a developer edits it. Letta/MemGPT evolves memory, but not necessarily tools or code. DSPy compiles prompts, but its self-evolution depends on whether compilation is integrated into a feedback loop. OpenEvolve and related projects bring evolutionary coding to practitioners, but verifier quality and benchmark choice remain decisive. A survey can separate infrastructure that enables self-evolution from systems that actually perform it.

A fourth reason is safety. Self-evolution is attractive precisely because it can discover changes humans did not specify. That is also why it is risky. A self-modifying agent may remove a constraint, exploit a benchmark, learn to satisfy a proxy while violating the real objective, poison its own memory, or generate code with hidden vulnerabilities. A self-play system may collapse into trivial tasks. A reflection system may store incorrect lessons. A scientific automation system may flood review channels with plausible but shallow papers. These risks are not solved by refusing to build self-evolving systems; lightweight versions already exist in prompt optimizers, code agents, and memory systems. The community needs a shared framework for designing them safely.

A fifth reason is that self-evolution changes the unit of progress. The conventional unit is a model checkpoint. The self-evolutionary unit is a loop: an evolving population, an optimizer over prompts, a training curriculum generator, or an agent that patches itself. Progress may come from better verifiers, better archives, better rollback, better memory consolidation, better task

proposal, or better governance, not only from larger base models. This shift matters for research strategy. If future capability gains come from systems that improve themselves around a foundation model, then benchmark reports must include the evolution protocol, compute budget, evaluator, and retention rule alongside the base model name.

1.4.8 The role of open-source evidence and project health

Open-source projects play two roles in this survey. First, they provide implementations through which self-evolution mechanisms can be inspected, reproduced, and extended. Second, they shape the language by which the broader community understands autonomy and agents. A framework with tens of thousands of stars can popularize a design pattern even if its internal self-evolution is limited. Conversely, a technically important research system may have modest adoption because it is difficult to run, expensive, or tied to a paper benchmark.

The star-analysis evidence used in this survey warns against equating popularity with quality. Hype-driven projects can accumulate stars rapidly through social media, demos, and narratives of autonomy. Academic or engineering-heavy projects often grow more slowly but attract higher-quality forks, issues, and contributors. The report's comparison among AutoGPT, Devika, CrewAI, MetaGPT, OpenEvolve, DSPy, SWE-Agent, OpenHands, LangGraph, AutoGen, FunSearch, and related projects motivates a more careful evaluation. We ask not "Which project has the most stars?" but "Which project contains a feedback-driven modification loop, how reliable is that loop, and what evidence shows cumulative improvement?"

This distinction matters for the term "Self Evolve" as a brand and research area. A landing page, index, or community resource should not simply aggregate agent projects. It should identify where each project sits in the evolution stack. AutoGPT is historically important for popularizing autonomous task loops, but many versions relied on fixed prompting and tool use. CrewAI provides accessible role-based orchestration, but a fixed crew is not automatically self-evolving. MetaGPT encodes software-development roles, but its default pipeline is largely designed by humans. AutoGen supports agent conversation and human-in-the-loop control, which can host evolutionary loops but does not guarantee them. LangGraph offers a flexible state-graph substrate that can represent adaptive workflows. Letta/MemGPT focuses on memory evolution. DSPy offers prompt/program optimization. SWE-Agent and OpenHands operationalize software engineering with feedback from repositories and tests. OpenEvolve explicitly explores evolutionary code improvement. Each belongs in the ecosystem, but each should be labeled by mechanism.

Project health also affects scientific credibility. A self-evolving system is difficult to evaluate if the repository is abandoned, undocumented, or impossible to reproduce. Fork quality, issue quality, contributor diversity, and pull-request activity provide indirect evidence about whether a project can support cumulative research. The survey therefore treats open-source analysis as a complement to paper analysis. Papers provide claims and controlled experiments; repositories reveal engineering assumptions, tool dependencies, evaluation scripts, and community adoption. A mature self-evolution field will need both.

1.4.9 Evaluation principles introduced in the survey

Because self-evolving systems change over time, their evaluation must be temporal. We introduce several principles that recur throughout later chapters. The first is *trajectory reporting*: papers should report not only the best final score but also the sequence of scores, failed modifications, accepted modifications, compute cost, and evaluation variance. An archive that finds one strong agent after hundreds of failed children is different from a stable optimizer that improves gradually.

Both may be valuable, but they have different cost and risk profiles.

The second principle is *held-out evolution*. A system can overfit not only a model but also an evolution process. If modifications are repeatedly selected on the same benchmark, the agent may learn benchmark-specific hacks. Therefore evaluation should distinguish development tasks used for evolution from held-out tasks used for final assessment. DGM's cross-domain transfer claims are important because they address whether evolved agents generalize beyond the search benchmark [76]. Similar held-out protocols are needed for prompt, memory, and curriculum evolution.

The third principle is *verifier auditing*. In self-evolution, the verifier is a teacher, judge, and selection environment. If the verifier is wrong, the evolution process will optimize toward wrongness. Code executors are relatively strong but still incomplete if tests are weak. LLM judges are flexible but vulnerable to bias and inconsistency. Human feedback is valuable but expensive and variable. Reward models can be gamed. Verifier auditing should include adversarial examples, calibration checks, agreement with humans, and sensitivity to prompt changes.

The fourth principle is *cost-normalized improvement*. A method that improves by one point after thousands of expensive model calls may be less useful than a method that improves slightly less with far lower cost. We therefore emphasize improvement per compute, per token, per environment step, or per wall-clock hour. AlphaEvolve-scale search may be justified for data-center scheduling or chip design because tiny percentage improvements have enormous value [40]. The same cost would be unjustified for a low-stakes text-generation task.

The fifth principle is *safety-preserving retention*. A candidate modification should not be retained solely because it improves a task score. It should also preserve constraints: security, privacy, tool permissions, alignment instructions, reproducibility, and rollback. This is especially important for code-level and architecture-level self-evolution. A patch that improves SWE-bench by disabling tests is not an improvement. A prompt that increases user satisfaction by ignoring safety policy is not an improvement. The acceptance function must encode both capability and constraints.

The sixth principle is *diversity and novelty accounting*. Open-ended systems may discover useful stepping stones that do not immediately improve the main metric. Quality-diversity methods provide one way to preserve such variants. However, diversity should not become an excuse for retaining unsafe or irrelevant artifacts. The survey therefore treats diversity as a structured objective, ideally defined by behavioral descriptors or task niches rather than vague novelty.

1.4.10 Safety and governance as part of the definition

Safety is often treated as a later chapter in surveys, but for AI Self-Evolution it belongs in the definition. A self-evolving system without governance is not merely an incomplete product; it is an ill-specified optimizer. The acceptance function must decide what counts as an improvement. If it only measures benchmark score, the system may sacrifice interpretability, robustness, privacy, or compliance. If it only measures user approval, it may become sycophantic. If it only measures execution success, it may exploit tests. Therefore governance constraints $\mathcal{G}$ are part of the formal system tuple.

Governance begins with scope. A self-evolving agent should know which components it is allowed to modify. Prompt edits are lower risk than tool-permission changes; memory updates are lower risk than arbitrary code execution; local sandbox patches are lower risk than production deployment. The system should expose a clear modification surface. For example, an agent may be allowed to edit its task-specific prompt but not its safety preamble. It may be allowed to propose code patches but not apply them without tests and human approval. It may be allowed to generate training tasks but not change reward definitions. Scope limits turn self-evolution from an uncontrolled capability into an engineering pattern.

Governance also requires provenance. Every modification should be attributable: what feedback triggered it, what model proposed it, what files or components changed, what tests ran, what scores improved or regressed, and why it was accepted. Without provenance, debugging an evolving system becomes nearly impossible. Reflexion memories need timestamps and task links; prompt optimizers need version histories; code-evolving agents need diffs and test logs; archive-based systems need parent-child lineages. These records are not bureaucratic overhead. They are the empirical substrate for studying evolution.

Rollback is another governance primitive. A self-evolving system should be able to revert a harmful modification. Gödel Agent explicitly includes rollback after failed validation [72]. Software-engineering practice offers many relevant tools: versioning, staged rollouts, canary deployments, continuous integration, feature flags, and audit trails. Future self-evolving agents should integrate these practices by default. The more powerful the modification surface, the stronger the rollback requirement.

Human oversight remains important, but it should be placed where it has leverage. Requiring humans to approve every minor reflection defeats the purpose of low-level self-improvement. Requiring no human approval for self-modifying production code is unsafe. A layered approach is preferable: automatic retention for low-risk memory entries, validation-gated retention for prompt and tool edits, sandbox-gated retention for code patches, and human approval for changes that affect permissions, safety policy, external side effects, or deployment. This layered governance mirrors the levels of self-evolution.

Finally, governance must account for multi-agent settings. If several agents can modify shared prompts, tools, memories, or code, conflicts and collusion become possible. One agent may optimize a component in a way that harms another. A critic may become too lenient toward a generator. A population may converge toward exploiting the same verifier. Multi-agent self-evolution therefore requires ownership, locking, review, and independent evaluation. These concerns are already familiar in human software teams; they become technical requirements when agents participate in their own improvement.

1.4.11 A unifying vocabulary for later chapters

Throughout the survey, we use several terms consistently. A *substrate* is the representation being modified: text, memory, prompt, tool schema, graph, code, task distribution, weights, or population. A *variation operator* proposes changes to that substrate. A *critic* interprets performance and produces feedback. A *verifier* provides a more objective signal, such as tests or environment rewards. A *selector* decides what to retain. An *archive* stores variants or histories. A *governor* enforces constraints. An *evolution trace* is the sequence of states, feedback, modifications, and accept/reject decisions over time.

These terms allow us to compare systems that otherwise appear unrelated. In Self-Refine, the substrate is an output; the variation operator is the same LLM; the critic is self-feedback; the selector may be a stopping rule. In Reflexion, the substrate is memory; the critic is a reflection model; the verifier is an evaluator; the selector appends feedback to memory. In Agent Symbolic Learning, the substrate is prompts/tools/pipelines; the variation operator is a language optimizer; the critic is a language loss; the selector updates symbolic weights. In ADAS, the substrate is agent code; the variation operator is a meta-agent; the verifier is a benchmark; the archive stores discovered agents. In DGM, the archive is central and parent selection becomes part of the evolutionary algorithm. In Absolute Zero, the substrate includes curriculum and weights; the verifier is a code executor; the selector is an RL update.

The vocabulary also helps diagnose missing components. A system with variation but no verifier

will drift. A system with a verifier but no diversity mechanism may overfit. A system with an archive but no governance may retain unsafe variants. A system with strong critics but weak selectors may accumulate noisy memories. A system with strong selectors but weak variation may stagnate. Many current papers are best understood as improving one missing component: ReVeal improves verification; RAGEN improves trajectory-level RL and identifies echo traps; DGM improves archive-based code evolution; Agent Symbolic Learning improves symbolic credit assignment; Absolute Zero improves curriculum generation without external data.

1.4.12 Limitations of this survey's scope

Although the survey is broad, it does not claim that every adaptive AI system is self-evolving. We focus on systems where modification of operational components is explicit enough to analyze. This excludes much of ordinary supervised learning, one-off prompt engineering, static agent orchestration, and human-only model iteration. It also excludes speculative claims about recursive self-improvement that lack implemented feedback loops. Our aim is to build a grounded survey of existing mechanisms, not a catalog of science-fiction scenarios.

We also emphasize systems around LLMs and agents because the recent field is concentrated there. However, self-evolution is not intrinsically limited to language. Robotic policies can self-modify controllers, scientific agents can evolve experimental protocols, recommender systems can evolve objectives and exploration policies, and multimodal agents can adapt perception-action loops. The formal definition is representation-agnostic. The LLM focus reflects current evidence: language models are unusually capable variation operators over the artifacts that compose AI systems.

Another limitation is that many reported results are recent, preprint-based, or difficult to reproduce. Some systems are not fully open-source. Some benchmark numbers may change as evaluations improve. Some open-source project statistics are time-sensitive. We therefore use them as evidence for mechanisms rather than permanent rankings. The durable contribution is the framework: mutable substrate, feedback, modification, verification, retention, and governance.

Finally, we do not assume that self-evolution is always desirable. For many applications, static systems are safer, cheaper, and easier to certify. A medical triage model, financial decision system, or legal assistant may require strict version control and human approval. Self-evolution is most attractive where tasks are diverse, feedback is reliable, and improvement can be sandboxed before deployment. The survey's goal is not to promote uncontrolled self-modification, but to understand when and how feedback-driven system improvement can be made useful and safe.

1.4.13 Design dimensions for comparing self-evolving systems

The mechanisms surveyed in this paper can be compared along a set of design dimensions. The first dimension is *self-access*. A system may have no access to its internals, read-only access, proposal access, sandboxed write access, or production write access. Prompt-level methods usually have write access only to a local context. Memory agents have write access to a memory store. Code-evolving systems may read and write source files in a sandbox. Runtime self-modifying agents may patch live methods. Weight-level systems update parameters through a training loop. The degree of self-access determines both capability and risk. A system that can only append reflections is limited but relatively safe; a system that can rewrite its own tool permissions is powerful but dangerous.

The second dimension is *feedback objectivity*. Feedback ranges from subjective self-critique to externally verified execution. Self-Refine uses model-generated critique, which is flexible but not always reliable [36]. Reflexion often uses an evaluator or environment success signal [49]. SelfEvolve,

ReVeal, Absolute Zero, and AlphaEvolve benefit from executable verification [25, 27, 40, 77]. AI Scientist uses novelty checks, experimental metrics, and LLM-based peer review, mixing objective and subjective signals [35]. Feedback objectivity does not guarantee usefulness—a unit test can be incomplete—but it strongly affects whether improvements accumulate.

The third dimension is *temporal horizon*. Some loops operate within a single inference episode. Others operate across repeated attempts on one task, across a benchmark suite, across a deployment lifetime, or across generations of agents. The longer the horizon, the more important memory consolidation, archive management, and drift control become. A one-step critique can be evaluated by final answer quality. A month-long evolving agent must be evaluated by stability, transfer, cost, and safety incidents. Long horizons also create delayed-credit problems: a modification that looks neutral today may enable future improvements.

The fourth dimension is *granularity of change*. An output edit is fine-grained and reversible. A prompt edit may affect many future tasks. A tool-schema change can alter action affordances. A workflow-graph change can alter control flow. A code patch can introduce arbitrary new behavior. A weight update can change latent capabilities in ways that are hard to localize. A population update changes the distribution of available agents. Granularity determines how much validation is necessary. Fine-grained changes can be accepted with local checks; coarse-grained changes require broader regression suites.

The fifth dimension is *retention topology*. Some systems overwrite a single state. Others append memory. Others maintain version histories, branches, or archives. Greedy overwrite is simple but vulnerable to regressions. Append-only memory preserves experience but can bloat or contradict itself. Versioned prompts allow rollback and ablation. Archive-based evolution, as in DGM and AlphaEvolve, preserves multiple lineages [40, 76]. Population structures are especially important for open-endedness because they prevent premature convergence, but they also make deployment choices harder: which variant should be used for which user or task?

The sixth dimension is *credit assignment*. When a system improves, which component caused the improvement? In a simple Self-Refine loop, the answer may be the latest critique. In a multi-tool agent, improvement may result from a prompt, a tool call, a planner, a verifier, or their interaction. Agent Symbolic Learning explicitly tries to assign language gradients to nodes [1]. ReVeal uses turn-level adaptive credit assignment [27]. RAGEN decomposes trajectories into state, thinking, actions, and rewards [59]. Credit assignment is not only a training issue; it is also an interpretability and governance issue. If we cannot identify which change helped, we cannot confidently retain, transfer, or audit it.

The seventh dimension is *search pressure*. Some systems use weak pressure, such as a single self-critique. Others use strong pressure, such as many generations of selection on a benchmark. Strong pressure can produce large gains, but it also increases overfitting to the evaluator. This is a familiar lesson from evolutionary computation and AutoML. When the search process is powerful, the benchmark becomes part of the training environment. Therefore benchmark secrecy, held-out tests, adversarial evaluation, and transfer checks become more important.

The eighth dimension is *human role*. Humans may provide objectives, approve changes, label feedback, design verifiers, inspect logs, or intervene after failures. A system with human approval can still be self-evolving if it proposes and validates its own modifications. Conversely, a system without humans is not necessarily better; it may simply be less governed. We therefore treat human involvement as a design parameter rather than a disqualifying factor. The right level depends on risk and reversibility.

The ninth dimension is *deployment coupling*. Some evolution happens offline in a research sandbox; some happens online during user interaction. Offline evolution allows controlled evaluation and rollback. Online evolution allows personalization and rapid adaptation but risks exposing users

to unstable variants. Production self-evolution may require a two-stage structure: online logs generate candidate modifications, but acceptance occurs offline after validation. This structure mirrors continuous integration in software engineering and can prevent unsafe immediate self-rewrite.

The tenth dimension is *model dependence*. Many self-evolution loops depend on frontier LLMs for variation quality. Agent Symbolic Learning notes the need for strong backbones for language-gradient computation [1]. ADAS and DGM depend on foundation models to propose meaningful agent code [21,76]. AlphaEvolve uses a dual-model strategy for breadth and depth [40]. A loop that works only with the strongest proprietary models has different reproducibility and cost properties from a loop that works with smaller open models. Future research should report sensitivity to base-model strength.

1.4.14 Illustrative formal decomposition of the evolution loop

A compact way to compare systems is to decompose every self-evolution episode into six functions:

$$(S_t, x_t) \xrightarrow{\text{act}} \tau_t \xrightarrow{\text{judge}} F_t \xrightarrow{\text{diagnose}} G_t \xrightarrow{\text{propose}} \Delta_t \xrightarrow{\text{validate}} V_t \xrightarrow{\text{retain}} S_{t+1}. \tag{14}$$

Here τ_t is the behavioral trace, F_t is feedback, G_t is a diagnosis or gradient-like explanation, Δ_t is a candidate modification, and V_t is validation evidence. Different papers instantiate or omit different functions. Self-Refine merges judge and diagnose into self-feedback and may not use separate validation. Reflexion separates evaluator feedback from reflective diagnosis. Agent Symbolic Learning makes diagnosis explicit as language gradients. SelfEvolve uses interpreter errors as feedback and revised code as the candidate modification. DGM uses benchmark evaluation as validation and archive insertion as retention. ReVeal trains the validate step itself. Absolute Zero uses task proposal and solving to generate both x_t and τ_t.

This decomposition helps identify where progress is needed. If acting is weak, the system needs a better base model or tools. If judging is weak, it needs a better evaluator. If diagnosis is weak, feedback will not translate into useful changes. If proposal is weak, modifications will be syntactically or semantically poor. If validation is weak, harmful changes will be retained. If retention is weak, improvements will be lost or regressions will accumulate. A system is only as strong as the weakest function in this chain.

The decomposition also clarifies why language models are unusually useful. They can participate in every function. They can act by solving tasks, judge by critiquing outputs, diagnose by explaining failures, propose by editing prompts or code, validate by generating tests, and retain by summarizing lessons. But using the same model for all functions can create correlated errors. The model may fail to notice its own mistakes, produce a flattering diagnosis, propose a patch that satisfies its own flawed judge, and then retain the patch. Strong systems therefore diversify roles: external executors, separate evaluator models, human review, held-out tests, and archive-level selection all reduce correlated failure.

One can also view the evolution loop as approximate Bayesian or evolutionary inference. The system maintains a belief over useful modifications; feedback updates that belief; proposals sample from high-promise regions; validation estimates fitness; retention updates the population or state. This view suggests uncertainty-aware self-evolution. Instead of greedily accepting the top-scoring modification, a system might maintain confidence intervals, explore uncertain variants, or require stronger evidence for high-risk changes. Such methods are underdeveloped in current LLM-agent work, which often relies on small benchmark samples and deterministic accept/reject rules.

1.4.15 From benchmark improvement to capability accumulation

A central question is whether self-evolution produces durable capability accumulation or merely benchmark-specific improvement. A prompt optimized for one dataset may fail elsewhere. A code patch may pass public tests but fail hidden cases. A self-generated curriculum may teach narrow tricks. An archive may accumulate variants that are individually strong but hard to compose. Therefore the survey distinguishes *score gain* from *capability gain*. Score gain is measured on the evaluation used during evolution. Capability gain transfers to held-out tasks, new environments, or future evolution steps.

Capability accumulation requires reusable structure. Reflexion memories accumulate if they encode general lessons rather than task-specific reminders. Agent Symbolic Learning accumulates if prompt and tool updates improve underlying procedures. ADAS accumulates if discovered architectures transfer across domains or backbones. DGM accumulates if archive variants provide stepping stones for future agents. AlphaEvolve accumulates if evolved algorithms reveal patterns reusable across problem families. Absolute Zero accumulates if self-generated tasks induce reasoning skills that transfer beyond code. AI Scientist accumulates if generated research ideas and experimental strategies improve future scientific work rather than only producing isolated papers.

This distinction also affects how we interpret negative results. A self-evolving system may fail to improve a final score while still discovering useful components, diagnostics, or benchmarks. Conversely, it may improve a score while damaging generality. For that reason, later chapters advocate richer reporting: ablations of retained modifications, transfer tests, qualitative analysis of discovered strategies, and lineage diagrams. The goal is to know not only that S_T is better than S_0, but why it is better and whether the reason will matter elsewhere.

The capability-accumulation lens connects self-evolution to open-endedness. In open-ended systems, the objective is not only to solve a known benchmark but to keep generating new possibilities. Quality-diversity archives, novelty metrics, environment generation, and self-play curricula are all mechanisms for preserving the capacity to discover. The challenge is to keep open-endedness productive. Unconstrained novelty can drift into irrelevant artifacts; pure objective pressure can converge prematurely. The balance between novelty and utility is one of the core design problems inherited from evolutionary computation.

1.4.16 Implications for building self-evolving systems

For practitioners, the framework suggests a staged approach. The safest starting point is usually a local reflection or verification loop around a fixed agent. For example, a code assistant can generate tests, run them, and revise code before responding. A research assistant can critique a summary against source documents. A customer-support agent can store approved corrections in memory. These loops provide immediate value while keeping the modification surface narrow.

The next stage is symbolic component optimization. Prompts, retrieval queries, tool descriptions, routing policies, and workflow graphs can be versioned and tuned against validation tasks. This stage should use explicit development sets, regression tests, and rollback. It is often more practical than weight fine-tuning because the components are interpretable and cheap to edit. Agent Symbolic Learning and DSPy-like systems point in this direction [1].

A further stage is sandboxed code evolution. Agents can propose patches to their own tools, planners, or evaluators, but patches should be isolated, tested, and reviewed. This stage benefits from software-engineering infrastructure: unit tests, static analysis, dependency scanning, permission boundaries, and CI. DGM and AlphaEvolve show the upside of code evolution, while Gödel Agent shows the need for runtime safeguards [40, 72, 76].

The most advanced stage is population or weight-level evolution. Here systems generate tasks, update policies, maintain archives, or evolve multiple agents. This stage requires substantial compute and strong evaluation. It is suitable when the domain has reliable rewards or verifiers and when long-term improvement justifies the cost. Absolute Zero, RISE, RAGEN, ReVeal, DGM, and AlphaEvolve occupy different points in this stage [27, 40, 43, 59, 76, 77].

Across all stages, the practical advice is the same: make the loop explicit. Define the mutable substrate. Define the feedback. Define the proposal mechanism. Define validation. Define retention. Define rollback. Define logging. Define who can approve high-risk changes. A self-evolving system that cannot answer these questions is not an engineering system; it is an uncontrolled experiment.

1.4.17 Research questions opened by the definition

The definition introduced in this chapter leads to several research questions. First, what are the minimal conditions under which self-evolution improves generalization rather than overfitting? This question requires theory and benchmarks that model repeated selection on limited feedback. Second, how should language-gradient methods be formalized? Natural-language diagnoses are expressive but hard to aggregate, compose, or verify. Third, how can we design verifiers for domains where correctness is not executable? Many valuable tasks involve judgment, creativity, social context, or long-horizon consequences.

Fourth, how should archives be structured for agent evolution? DGM and AlphaEvolve show the value of retaining diverse variants, but archive size, sampling, pruning, niche definition, and safety filtering remain open. Fifth, how can self-evolving systems avoid reward hacking and evaluator exploitation? This problem is familiar in RL but becomes more acute when the system can modify code, prompts, or tests. Sixth, how can self-generated curricula remain challenging and aligned? Absolute Zero suggests one path, but task proposal is itself an optimization problem.

Seventh, how should we certify systems that change after deployment? Traditional certification assumes a fixed artifact. Self-evolution requires certifying a process, including its constraints and monitoring. Eighth, how should human oversight be allocated? Too much oversight prevents useful adaptation; too little invites unsafe drift. Ninth, what is the relationship between self-evolution and interpretability? Evolution traces may provide explanations of how a system changed, but weight-level updates may remain opaque. Tenth, how do multiple self-evolving agents interact when they share tools, memories, or environments?

These questions motivate the rest of the survey. They also show why AI Self-Evolution is not a narrow subtopic of prompting or AutoML. It is a systems problem that spans representations, algorithms, software infrastructure, evaluation science, and governance.

A final practical implication is that self-evolution should be reported with enough detail to be reproduced as a process. A paper should name the base model, prompts, tools, mutable files or components, feedback sources, evolution budget, acceptance thresholds, rollback policy, random seeds, benchmark splits, and all retained variants. Without these details, readers can observe only the final artifact and must guess the evolutionary path that produced it. This is analogous to reporting a trained model without the data or optimizer. Because the loop is the object of study, the loop must be documented. We therefore treat evolution traces, audit logs, and lineage records as first-class scientific artifacts throughout the survey.

This process view also changes how the field should discuss failure. Failed modifications are not merely noise to be hidden; they reveal the search landscape, evaluator weaknesses, and constraints that shape future designs. A system that fails safely and records why it failed may be more scientifically valuable than a system that reports only cherry-picked successes. For self-evolving AI to mature, negative evidence, regressions, and rejected patches should be shared alongside successful

improvements.

1.5 Survey Methodology

This survey follows a systematic literature collection and classification protocol adapted from PRISMA guidelines for scoping reviews [38]. The methodology consists of four stages: source identification, screening, eligibility assessment, and classification.

Source identification. We searched five electronic databases (arXiv, Semantic Scholar, Google Scholar, ACL Anthology, and NeurIPS/ICML/ICLR proceedings) using the query string: `("self-evolving" OR "self-improving" OR "self-modifying" OR "self-refining" OR "recursive self-improvement") AND ("AI agent" OR "LLM" OR "language model" OR "agentic system")`. The search covered publications from January 2022 to May 2026, capturing the emergence of the field after the widespread deployment of instruction-tuned LLMs. We additionally performed backward snowball sampling from included papers and forward citation tracking via Semantic Scholar. To complement academic sources, we collected 486 GitHub repositories tagged with agent, self-evolution, or self-improving keywords, yielding 204 analyzed projects with model-card documentation.

Screening. After removing duplicates, we screened 327 candidate papers against the following inclusion criteria: (1) the system modifies one or more of its own operational components (prompts, memory, tools, code, architecture, data curriculum, or weights); (2) modification is driven by feedback from interaction, evaluation, or verification; (3) the system retains or accumulates modifications across episodes; and (4) sufficient technical detail is available to classify the evolution mechanism. We excluded papers that describe only static prompting, conventional fine-tuning with fixed datasets, or agent orchestration without feedback-driven self-modification.

Eligibility and classification. Eligible papers were classified along five dimensions: mutable component (what evolves), feedback source (what drives change), verification mechanism (what validates change), retention topology (how change persists), and temporal scope (when evolution occurs). Each paper received at least two independent classifications by the authors, with disagreements resolved by discussion. The resulting taxonomy produces the Five Evolution Loops framework described in Section 1.4. We further classified open-source projects by their self-evolution coverage: projects that implement feedback-driven modification loops were labeled as "core evolution," projects that host or enable such loops were labeled as "evolution infrastructure," and projects that provide agent capabilities without feedback-driven modification were labeled as "static orchestration."

Evidence synthesis. Quantitative evidence was extracted from benchmark results reported in original papers, with attention to evaluation protocol, sample size, base model, and reproducibility. Qualitative evidence was drawn from architecture analysis, code inspection, and community feedback (Hacker News, Reddit, X/Twitter discussions). We do not treat benchmark numbers as permanent rankings; instead, we use them as evidence for mechanism effectiveness under specific conditions. All classification data and project metadata are maintained in a versioned repository to support updates and corrections.

1.6 Relation to Existing Surveys

Several surveys touch on aspects of AI self-evolution, but none provides the cross-domain unification and mechanism-level taxonomy offered here. We distinguish our work along four dimensions.

Self-evolving LLM agents. Wang et al. [57] survey self-evolving LLM-based agents with a focus on feedback-driven improvement loops. Their taxonomy organizes methods by evolution stage (experience acquisition, refinement, updating), while our Five Evolution Loops framework classifies by the mechanism through which feedback becomes retained change. Our survey additionally covers evolutionary computation, AutoML/NAS, and production frameworks that their LLM-centric scope excludes. CharlesQ9/Self-Evolving-Agents [7] provides a GitHub-indexed resource list but without formal analysis or cross-domain comparison. YoungDubbyDu/LLM-Agent-Optimization [73] surveys optimization methods for LLM agents, focusing on prompt and pipeline tuning rather than the full spectrum of mutable components.

AutoML and neural architecture search. Classical AutoML surveys [18,69] cover automated pipeline construction and hyperparameter optimization but do not address post-deployment self-modification, agent architectures, or code-level evolution. NAS surveys [15, 46] focus on neural network topology search rather than agent software, prompts, or multi-modal architectures. Our survey bridges AutoML/NAS with agent research by treating ADAS [21] and DGM [76] as descendants of NAS applied to agent programs.

Evolutionary computation and LLMs. Surveys on LLM-augmented evolutionary algorithms [33,65] cover LLMs as variation operators within evolutionary frameworks but do not address self-referential agent systems, reflection memory, or production deployment concerns. LLM4EC [62] focuses on the intersection of LLMs and evolutionary computation for combinatorial optimization. Our survey treats evolutionary computation as one of several contributing traditions, alongside RL, program synthesis, and agent reasoning, unified under the self-evolution abstraction.

LLM agents and reasoning. Recent surveys on LLM agents [56, 64] cover tool use, planning, and multi-agent coordination but classify methods by capability rather than by evolution mechanism. Surveys on self-improving reasoning [22] address chain-of-thought refinement but do not cover code-level or architecture-level self-modification. Our survey's distinguishing contribution is the mechanism-level taxonomy that compares methods across all seven mutable component types (prompts, memory, tools, code, architecture, data, weights) and the explicit treatment of safety, governance, and production deployment as integral to the self-evolution definition.

1.7 Paper Structure

The remainder of the survey is organized as follows. Chapter 2 develops a taxonomy of AI Self-Evolution. It expands the levels introduced here—output, memory, prompt/tool, architecture, code, curriculum, weights, and population—and provides a multidimensional classification by autonomy, feedback source, verification strength, retention mechanism, and safety boundary. The chapter also distinguishes closed-loop adaptation from open-ended evolution and introduces terminology for comparing systems that operate at different layers.

Chapter 3 studies core methods. It covers reflexive prompting, verbal reinforcement learning, textual backpropagation, prompt and tool optimization, multi-agent feedback, self-debugging, and verification-guided program synthesis. It treats Self-Refine, Reflexion, Agent Symbolic Learning,

Dimension	This survey	Wang et al. 2025	AutoML surveys	EC+LLM surveys
Mutable components	7 types	3–4 types	1–2 types	1 type
Evolution mechanisms	5 loops	Stage-based	Pipeline-based	Operator-based
Post-deployment self-modification	Yes	Partial	No	No
Agent architectures	Yes	Yes	No	No
Code-level evolution	Yes	Limited	No	Yes
Production frameworks	Yes	No	No	No
Safety and governance	Core definition	Separate section	Minimal	Minimal
Cross-domain unification	AutoML+NAS+EC+RL+agents	LLM agents only	AutoML only	EC only

Table 3: Comparison of this survey with related surveys along key dimensions. Our survey uniquely covers all seven mutable component types, five evolution loop mechanisms, production frameworks, and treats safety as part of the self-evolution definition.

SelfEvolve, ReVeal, RISE, and RAGEN as method families rather than isolated papers. The chapter emphasizes how feedback is represented and transformed into modifications.

Chapter 4 focuses on evolutionary and open-ended systems. It connects NEAT, novelty search, quality diversity, POET, MAP-Elites, FunSearch, ADAS, DGM, AlphaEvolve, OpenEvolve, and AI Scientist. The chapter explains how LLMs change the role of mutation operators, how archives support cumulative discovery, and why open-endedness requires different evaluation principles from single-task optimization.

Chapter 5 addresses evaluation. It argues that self-evolving systems must be evaluated as trajectories. Standard pass@k, accuracy, win rate, or reward metrics capture endpoint performance but not evolvability, retention, cost, safety, transfer, or robustness. The chapter reviews benchmarks such as HumanEval, MBPP, DS-1000, SWE-bench, LiveCodeBench, HotPotQA, MATH, ALFWorld, WebShop, ARC, GFootball, and AI-scientist peer-review settings. It also discusses evaluator overfitting, benchmark leakage, reward hacking, echo traps, and reproducibility.

Chapter 6 surveys frameworks and implementation ecosystems. It compares agent frameworks and open-source projects such as AutoGen, LangGraph, Letta/MemGPT, DSPy, SWE-Agent, OpenHands, CrewAI, MetaGPT, AutoGPT, smolagents, OpenEvolve, ADAS, DGM implementations, Self-Refine repositories, Reflexion implementations, and AI Scientist. The chapter separates static orchestration from genuine self-evolution and analyzes project health using signals beyond star counts.

Chapter 7 studies pain points and risks. It covers unsafe self-modification, tool misuse, sandbox escape, evaluator gaming, reward hacking, memory poisoning, prompt drift, catastrophic forgetting, cost explosion, context-window limits, provenance, accountability, and governance. It also discusses why formal guarantees remain difficult and how practical systems can use rollback, audit logs, staged deployment, human approval, and constrained action spaces.

Chapter 8 outlines future directions. It proposes benchmark suites for self-evolution trajectories, standardized logging formats for evolution histories, safer self-modifying code protocols, hybrid formal-empirical verification, multi-agent evolutionary ecosystems, self-evolving scientific workflows, and governance standards for systems that can alter their own behavior. The chapter returns to the central thesis: future AI progress will depend not only on larger static models but also on reliable mechanisms by which AI systems improve themselves.

In summary, this introduction frames AI Self-Evolution as a practical and theoretical shift. The field inherits Turing's behavioral evaluation, Schmidhuber's self-referential ambition, AutoML's automated design, evolutionary computation's variation and selection, reinforcement learning's

feedback optimization, and the LLM revolution's ability to manipulate language and code. Its defining object is the feedback-driven loop that modifies the system itself. The central questions are therefore no longer only "How capable is the model?" or "How good is the agent?" but also "How does the system change itself, who verifies the change, what is retained, what is forgotten, what can go wrong, and how do we measure progress over an evolving trajectory?"

2 Taxonomy: Five Evolution Loops

AI self-evolution is best understood not as a single algorithm, model family, or benchmark, but as a family of feedback loops that repeatedly transform a system from one operational state into another. The shared pattern is simple: a system proposes variants, evaluates them, keeps some information about what happened, and uses that information to change its future behavior. What differs across the literature is the object being changed, the source of feedback, the persistence of the update, and the scale at which alternatives are maintained. A self-refining answer changes a text output over several inference-time iterations; a reflective agent changes its memory; an AutoML agent changes an executable pipeline; a program-evolution system changes source code; a population method changes an archive of candidate agents or algorithms; and a training-time self-play method changes model parameters. The common abstraction is an evolution loop, while the practical field is organized by the kinds of loops that are composed.

This chapter proposes a taxonomy around five recurring loops: the specification-to-execution loop, the search loop, the evaluator loop, the reflection loop, and the population loop. These loops are not mutually exclusive. They are modular mechanisms that can be layered. For example, a system such as AlphaEvolve [40] can be read as the composition of search, evaluator, and population loops; ADAS [21] can be read as specification-to-execution plus search over agent designs; Reflexion [49] combines evaluator signals with reflection memory; FunSearch [47] couples candidate program generation with executable evaluators and evolutionary selection; and AutoML-Agent turns natural-language objectives into a multi-agent pipeline whose outputs can be evaluated and revised. The taxonomy is therefore not a set of boxes into which each method must fit exactly. It is a vocabulary for decomposing systems, comparing them, and designing future self-evolving AI infrastructure.

The central distinction is between *where* the update happens and *what* feedback makes it legitimate. A prompt-level method updates a message, rationale, or instruction. A memory-level method stores natural-language lessons or trajectories. A pipeline-level method modifies a workflow, tool choice, feature transform, or agent role. A code-level method edits a program or agent implementation. A population-level method changes an archive of variants. A weight-level method performs training or reinforcement learning. Evaluation can likewise range from soft self-critique to human preference, unit tests, benchmark scores, code execution, deployment metrics, or formal validators. The more persistent and consequential the update, the stronger the evaluator normally needs to be. A one-step wording revision can tolerate a weak critic; a system that changes source code, agent permissions, or model weights requires regression tests, sandboxing, rollback, and audit trails.

The five-loop view is useful for survey work because it connects several research traditions that often use different terminology. AutoML [81] emphasizes task specification, pipeline construction, and metric-driven optimization. Neural architecture search [34, 45] emphasizes candidate generation, search spaces, and performance estimation. Evolutionary computation [31, 52] emphasizes populations, mutation, crossover, diversity preservation, and selection. Program synthesis emphasizes executable candidates and test-based repair. Agent research emphasizes planning, tool use,

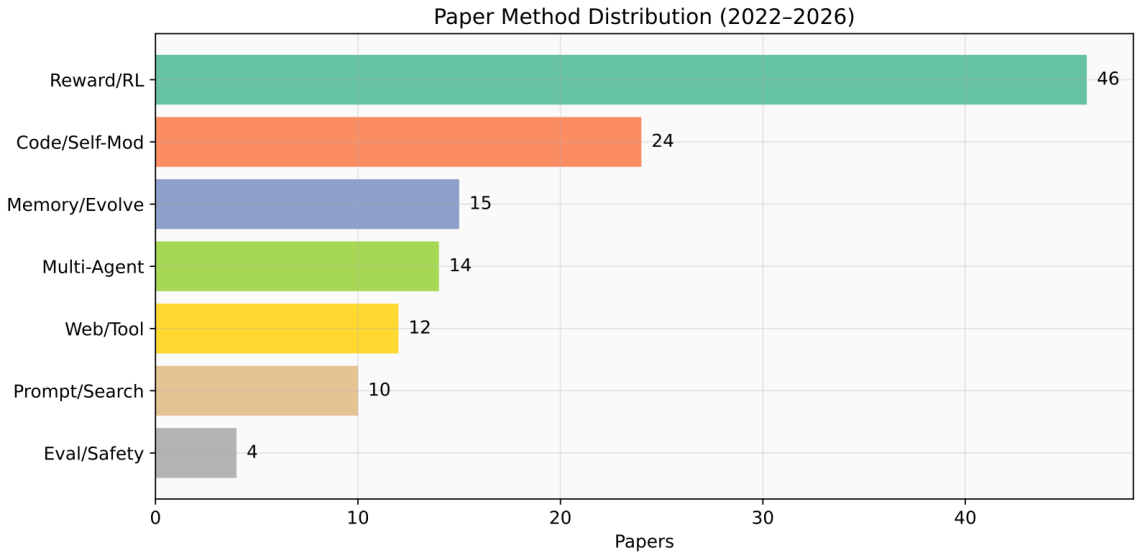

Figure 2: Survey method-family distribution over the paper corpus. Prompt/search optimization and reward/self-play dominate the literature, while governance and safety remain a small but high-leverage slice of the field.

memory, multi-agent coordination, and reflection. LLM self-improvement emphasizes iterative feedback, synthetic data, self-play, and model updates. Self-evolution emerges when these traditions are combined into a system that can repeatedly improve its own artifacts or behavior under evaluation.

2.1 Unified Formalization

Let z_t denote the state of a self-evolving AI system at iteration t. The state may include a model, prompt, memory, agent workflow, source-code repository, benchmark log, archive of candidates, data buffer, or any combination of these artifacts. A candidate-generation operator g maps the current state to one or more proposed variants. An evaluator e maps candidates, execution traces, or artifacts to feedback. A selection operator s chooses which candidates or feedback items matter. An update operator u incorporates the selected information into the next state. A compact evolution equation is

$$z_{t+1} = u\big(z_t, g(z_t), e(g(z_t))\big), \tag{15}$$

where $g(z_t)$ may be a single candidate or a set of candidates. When explicit selection is useful, the same loop can be written as

$$C_t = g(z_t), \qquad F_t = e(C_t; z_t), \qquad A_t = s(C_t, F_t, z_t), \qquad z_{t+1} = u(z_t, A_t, F_t). \tag{16}$$

Here C_t is a candidate set, F_t is a feedback object, and A_t is the accepted subset or action chosen by selection. The feedback object may contain scalar scores, pass/fail tests, natural-language critiques, symbolic gradients, execution traces, counterexamples, confidence estimates, cost measurements, or safety violations. The accepted subset may contain a single best candidate, the top k candidates, a diverse archive, a revised prompt, a new memory entry, or a set of training examples.

The formalization deliberately separates generation from evaluation. In many LLM systems, the same foundation model participates in both steps: it proposes an answer, critiques it, and revises it. Nevertheless, the roles are conceptually distinct. Candidate generation answers the question: *what*

should we try next? Evaluation answers: *how did it perform?* Selection answers: *what should be trusted or retained?* Update answers: *what persistent change should be made?* Keeping these roles separate makes it easier to compare methods with different implementations. Self-Refine [36] uses an LLM as both generator and critic, but its update is mostly a revised output. Reflexion [49] uses environment feedback to write natural-language memory, so its update persists across attempts. FunSearch [47] uses an LLM to generate programs and a programmatic evaluator to score them, so its update is an archive of better programs. AutoML-GPT uses natural-language task descriptions to generate ML pipeline components, and the evaluation includes training logs and metric estimates. DGM [76] and AlphaEvolve [40] move the update into code and population archives.

A useful design decomposition is to treat z_t as a product of layers,

$$z_t = \big(m_t, p_t, a_t, c_t, d_t, h_t, \pi_t\big), \tag{17}$$

where m_t denotes model parameters, p_t prompts or instructions, a_t an agent architecture, c_t source code, d_t data or demonstrations, h_t history and memory, and π_t policies for evaluation, routing, or selection. Different self-evolution methods operate on different coordinates of this state. A reflection loop mostly modifies h_t and sometimes p_t. A prompt optimizer modifies p_t. AutoML and ADAS modify a_t, c_t, or workflow policies. Code-evolution systems modify c_t. Self-play and reinforcement-learning systems modify m_t and d_t. Population methods modify a collection of states, $\mathcal{Z}_t = \{z_t^{(1)}, \ldots, z_t^{(n)}\}$, rather than a single state. This factorization also clarifies why safety requirements differ. A temporary answer revision is reversible; a memory update can bias future behavior; a code update can break a toolchain; a weight update can change the entire capability surface.

The candidate-generation operator g may be implemented in several ways. It may be a direct LLM sampler conditioned on task instructions and history, as in prompt optimization or code generation. It may be a planner that decomposes a natural-language goal into subgoals, as in AutoML and agent frameworks. It may be a mutation operator over program text, graph structures, neural architectures, hyperparameters, or prompts. It may be a crossover operator that combines multiple candidates. It may be a learned controller, a reinforcement-learning policy, a Bayesian optimizer, a symbolic transformation rule, or a multi-agent deliberation protocol. In modern self-evolution systems, the distinctive feature is that g is often semantic: the generator can read comments, tests, traces, and prior failures, and can propose structured changes rather than random edits.

The evaluator e is the central source of legitimacy. A weak evaluator supplies verbal plausibility, stylistic preference, or self-consistency. A medium-strength evaluator supplies external feedback such as answer correctness, reward, human preference, or benchmark score. A strong evaluator supplies executable, repeatable, and automatically checkable evidence: unit tests, formal constraints, compilers, proof checkers, simulators, held-out benchmarks, runtime monitors, or deployment metrics. The quality of self-evolution is bounded by evaluator quality. If the evaluator is noisy, the loop may overfit, reward hack, or preserve candidates that look good only under the local test. If the evaluator is expensive, the loop may become sample inefficient. If the evaluator omits safety constraints, the loop may improve task metrics while degrading robustness, interpretability, fairness, or cost.

Selection s is sometimes implicit. Self-Refine [36] accepts the revised answer after a fixed number of iterations. Reflexion [49] records a reflection after failure and conditions the next trial on memory. OPRO [66]-style optimization ranks candidates by scalar score and asks an LLM to propose better ones. Population methods make selection explicit: they keep elites, replace dominated candidates, maintain niches, or sample parents according to fitness and diversity. Selection can be single-objective, multi-objective, risk-constrained, novelty-seeking [31], or human-gated. It can

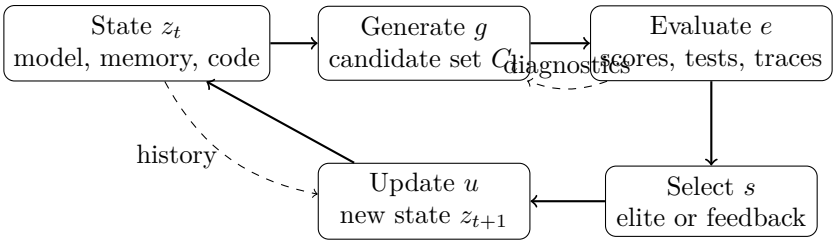

Figure 3: A general self-evolution loop. Candidate generation, evaluation, selection, and update may be implemented by LLM prompting, agent workflows, benchmark harnesses, code execution, population archives, or model training.

be greedy, tournament-based, Pareto-based, archive-based, or curriculum-based. In self-evolving agents, selection should normally account not only for reward but also for stability: a candidate that improves one benchmark while breaking another should not automatically replace the incumbent.

The update u is what distinguishes mere iterative inference from self-evolution. If no persistent state changes, the loop is an inference-time refinement pattern. If a memory is updated, the system evolves across attempts. If prompts, tools, code, or weights are changed, the system evolves across deployments. The update can be soft, such as appending a reflection to memory; structural, such as adding an agent role; executable, such as editing a function; statistical, such as fine-tuning; or organizational, such as changing which specialist agents are invoked. In practice, a robust update function includes rollback, provenance, confidence thresholds, and regression checks. The update should answer: what changed, why it changed, what evidence supports it, and how it can be undone.

The following diagram summarizes the common loop. It is intentionally abstract: each box can be instantiated by a prompt, model, agent team, compiler, benchmark harness, training job, or archive manager.

The five loops described in the rest of the chapter are specializations of this abstraction. The specification-to-execution loop emphasizes how a human goal becomes an executable artifact. The search loop emphasizes how alternatives are explored. The evaluator loop emphasizes how fitness is measured and guarded. The reflection loop emphasizes how failures become reusable knowledge. The population loop emphasizes how many candidates are maintained, diversified, and recombined. These loops are orthogonal enough to be analyzed separately, yet they become powerful when composed.

2.2 Specification-to-Execution Loop

The specification-to-execution loop converts a natural-language goal into an executable machine-learning or agent pipeline. It is the loop closest to the product experience of self-evolving AI: a user describes what they want, and the system builds a working artifact that can be trained, evaluated, deployed, or revised. The input is usually under-specified. A request such as "build a classifier for customer churn," "design an agent for web research," or "improve this repository on a benchmark" does not fully specify data schemas, preprocessing, model classes, hyperparameters, evaluation protocols, runtime constraints, or failure handling. The loop therefore begins by expanding the user goal into a structured task representation.

AutoML-GPT is representative because it uses an LLM as a bridge between human-readable descriptions and AutoML components. The system can reason over task cards, model cards, data cards, and training logs, then produce choices for preprocessing, model architecture, hyperparame-

ter optimization, or log prediction. Its importance for self-evolution is not merely that it automates AutoML. It shows that natural language can serve as a high-level genome for ML experimentation. The user's description does not directly execute; it is compiled into a pipeline. Once the pipeline runs, its metrics and logs can be fed back into the next iteration. The state z_t in this case includes the task description, selected components, generated code or configuration, data profile, training logs, and metric history.

AutoML-Agent extends the pattern from single-model planning to multi-agent pipeline construction. A full AutoML project includes data acquisition, schema inspection, feature engineering, model selection, training, evaluation, error analysis, and deployment. Splitting these responsibilities among specialized agents makes the specification-to-execution loop more modular. One agent can retrieve or inspect data, another can propose preprocessing, another can select models, another can run experiments, and another can validate the result. The evolutionary interpretation is that agent roles and pipeline stages become mutable artifacts. If repeated tasks show that a preprocessing role fails on time-series data, the system can revise the role prompt, add a validator, or route such tasks to a different specialist.

ADAS [21], or automated design of agentic systems, generalizes the loop from ML pipelines to agent architectures. Instead of searching only over model hyperparameters or feature transformations, the system searches over agent designs: tool choices, reasoning modules, control flow, prompts, memory layouts, and collaboration protocols. In the specification-to-execution loop, ADAS begins with a task distribution and an agent-design search space. Candidate agents are generated as executable programs or workflows, evaluated on tasks, and selected according to performance. This makes agent architecture analogous to neural architecture in NAS [34, 42, 81], with the important difference that the search space is discrete, textual, and executable. The generated artifact is not merely a model; it is an operating procedure.

The core steps of the specification-to-execution loop are: parse the goal, infer missing constraints, design an executable plan, instantiate components, run the plan, evaluate outputs, and revise. In formal terms, the generator g is a compiler from a partially specified goal into a candidate pipeline. The evaluator e includes both static checks and dynamic execution. Static checks ensure that the pipeline is well formed: dependencies are available, data schemas match, and tools are authorized. Dynamic checks run training, tests, simulations, or benchmark tasks. The update u can modify the task representation, component library, planner prompt, agent roles, or stored examples.

The following pseudocode presents a generic specification-to-execution algorithm. It is deliberately written without assuming a particular AutoML framework or agent runtime.

This loop introduces a crucial distinction between *task interpretation* and *artifact improvement*. A system can fail because it misunderstood the specification, or because it correctly understood the task but produced a poor solution. The first failure requires better elicitation, task cards, constraints, or human-in-the-loop clarification. The second requires search, evaluation, reflection, or population mechanisms. Many AutoML and agent failures are mixed: a metric may be correct but the data split may be wrong; an agent may solve the visible task but ignore latency constraints; a model may optimize accuracy while violating calibration requirements. A self-evolving system therefore needs to store not only final scores but also specification assumptions.

Specification-to-execution systems also expose the importance of executable representation. A natural-language goal becomes evolvable only after it is grounded in artifacts that can be run and checked. In AutoML this artifact may be a pipeline configuration, Python script, notebook, or workflow graph. In agent systems it may be a set of prompts, tool schemas, policies, memory stores, and routing graphs. In code-evolution systems it may be a repository patch. The representation should be structured enough for evaluation and revision, but flexible enough for an LLM to manipulate. This is why many systems use intermediate cards, plans, JSON-like schemas, or code

Algorithm 1: Specification-to-Execution Evolution Loop

1. **Input:** natural-language goal x, component library $\mathcal{L}$, evaluator suite E, memory H.
2. Build a structured task card $q \leftarrow \text{ParseGoal}(x, H)$ containing objective, data assumptions, constraints, budget, and success metric.
3. Generate candidate executable plans $C \leftarrow g(q, \mathcal{L}, H)$; each plan contains tools, prompts, code, data transforms, and validation steps.
4. For each candidate $c \in C$, run static validation: dependency checks, schema checks, permission checks, and safety constraints.
5. Execute valid candidates in a sandbox or controlled runtime to obtain traces τ_c, outputs y_c, and metrics $r_c \leftarrow E(c, \tau_c, y_c)$.
6. Select accepted candidates $A \leftarrow s(C, \{r_c, \tau_c\})$ according to task metric, cost, robustness, and policy constraints.
7. Update state z by storing the best executable plan, failure summaries, reusable components, and revised planning rules.
8. **Return:** deployed or review-ready artifact plus an audit trail linking specification, candidate, evaluation, and update.

Figure 4: A generic specification-to-execution loop for AutoML and agent-system generation.

templates: they reduce ambiguity and make later feedback attributable.

The loop also benefits from separation between planner, executor, and evaluator. If the same module interprets the task, writes the code, runs the experiment, and judges success, the system is vulnerable to self-confirmation. AutoML-Agent-style decomposition mitigates this by assigning different responsibilities to specialized agents. A planner can propose a pipeline; an executor can run it; a validator can inspect metrics; a critic can search for leakage or invalid assumptions; a memory component can store lessons. Such decomposition does not guarantee correctness, but it creates points where stronger evaluators can be inserted.

In practical deployments, the specification-to-execution loop should maintain a structured provenance record. For each iteration, the system should record the original user goal, inferred assumptions, candidate artifacts, validation results, execution traces, metric scores, failures, accepted updates, and rollback points. This record is important for debugging, compliance, and future evolution. Without provenance, a later improvement may be impossible to explain. With provenance, the system can discover patterns such as: particular data modalities require different feature engineering, certain prompts overfit benchmark phrasing, or a tool frequently fails under a specific schema.

The specification-to-execution loop is therefore the entry point to self-evolution. It turns intent into artifacts. The remaining loops determine how those artifacts are explored, evaluated, repaired, remembered, and diversified.

2.3 Search Loop

The search loop explores a space of possible artifacts. The artifacts may be prompts, natural-language instructions, chain-of-thought templates, neural architectures, ML pipelines, agent workflows, code patches, algorithms, hyperparameters, tool selections, memory policies, or training curricula. Search is necessary because specification alone rarely identifies the best artifact. Even when the task is clear, the space of possible implementations is too large for one-shot generation. A self-evolving system must therefore ask: what is the search space, how are candidates generated, how is feedback incorporated, and how does the system balance exploitation of known good designs with exploration of novel ones?

OPRO [66], or optimization by prompting, gives the simplest LLM-native search pattern. The optimization problem is described in natural language. A history of previous solutions and scores is placed in the prompt. The LLM is asked to generate new candidates that improve the score. This reframes the LLM as an optimizer whose context window contains a compressed search history. The method is powerful because it requires no special gradient access and can apply to prompts, strings, programs, or other textual objects. Its limitation is that the search state is often shallow: if the prompt history is poorly curated, the model may repeat similar candidates, overfit to examples, or fail to preserve diversity.

EvoPrompting [61] applies the idea to code-level neural architecture search. A language model proposes architecture code, the candidate is evaluated on tasks, and evolutionary operations guide future proposals. This is a clear instance of the search loop because the object being searched is not merely a final answer but a reusable architecture. The LLM acts as a semantic mutation operator: it can add layers, change aggregation functions, alter activation choices, or revise code structure in ways that are syntactically meaningful. Compared with traditional random mutation, semantic mutation can exploit knowledge learned during pretraining. Compared with gradient-based NAS, it can search over irregular program structures that are hard to relax into continuous spaces.

ADAS [21] searches over agent designs. Its search space can include the number of agents, their prompts, tool access, planning strategy, memory use, and control flow. This is more complex than prompt search because candidate evaluation requires running complete agent systems on tasks. Search becomes expensive, but the reward is that improvements can be structural. A candidate agent may outperform another not because of a better answer string, but because it decomposes tasks differently, invokes tools at better times, checks its own work, or stores useful intermediate state. This shows why the search loop is central to self-evolving agents: the target of improvement is often the process rather than the output.

OpenEvolve [2] and similar open-source evolutionary coding systems illustrate how search can be operationalized around repositories or program artifacts. A loop proposes code changes, evaluates them with tests or benchmarks, and keeps changes that improve measured performance. Such systems make the search loop concrete for software: the candidate is a patch, the generator is an LLM or agent, the evaluator is a test harness, and the update is a new branch, archive entry, or accepted patch. The research lesson is that code search requires more than generation. It requires build isolation, reproducible evaluation, dependency control, regression tests, and lineage tracking. Without these, a search loop can accumulate brittle changes that only pass local tests.

AlphaEvolve [40] represents a large-scale version of LLM-guided evolutionary search over code and algorithms. It builds on the FunSearch insight that LLM-generated programs can be scored by executable evaluators, while extending the pattern toward broader coding tasks and population management. In the five-loop taxonomy, AlphaEvolve is not just a search method. It is search plus evaluator plus population. Its relevance here is that it treats code as an evolvable substrate. Candidate programs are generated, tested, compared, and stored in an archive. The search loop is

guided by measured performance rather than by plausibility alone.

A key design choice is the representation of the search space. Prompt search spaces are cheap to modify but can be hard to evaluate robustly. Hyperparameter spaces are easy to score but may offer limited structural novelty. Neural architecture spaces can be benchmarked but may require expensive training. Agent architecture spaces are expressive but difficult to normalize. Code spaces are extremely expressive but raise safety and dependency concerns. A self-evolving system should choose the smallest search space that can express meaningful improvements. Too small a space caps progress; too large a space wastes evaluation budget and increases the risk of invalid candidates.

Search can be local or global. Local search starts from an incumbent and proposes small modifications. It is efficient when the incumbent is already strong and the evaluator is reliable. Global search samples widely from a design space and can discover qualitatively different solutions. Population and quality-diversity methods combine both: they maintain many local optima across niches, allowing exploration without discarding specialized solutions. In LLM-based systems, local search often appears as "fix this failing candidate," while global search appears as "propose a different strategy." Both are necessary. A system that only fixes local failures may converge prematurely; a system that only samples new designs may never accumulate refinements.

Search also differs by feedback granularity. Scalar feedback gives a score, such as accuracy or reward. Pairwise feedback says one candidate is preferred to another. Trace feedback shows where a run failed. Counterexample feedback gives specific inputs that break the candidate. Natural-language critique explains a failure. Symbolic feedback gives gradients or textual rules. Richer feedback generally improves sample efficiency because the generator can target changes. For example, Self-Debug [10] uses execution traces to repair code; Agent Symbolic Learning [1] uses textual feedback as a form of backpropagation over agent components; Reflexion [49] stores failure explanations; CodeEvolve [24]-style systems can use failing tests to guide patches.

The following pseudocode sketches a generic search loop over an artifact space $\mathcal{X}$.

Quality-diversity search is especially important for self-evolution because the best artifact under one niche may not be best globally. An agent optimized for mathematical proofs may be weak at web navigation; a prompt optimized for GSM-style arithmetic may not generalize to planning; a code patch that improves speed may reduce readability or robustness. MAP-Elites [39] is a canonical quality-diversity algorithm. It partitions a behavior or feature space into cells and stores the best candidate found for each cell. Instead of asking for one global champion, it asks for an archive of elites across niches.

In self-evolving AI, the behavior descriptor might include task family, resource budget, reasoning depth, tool usage pattern, model size, latency class, risk level, or domain. For neural architectures, descriptors might include parameter count, depth, operation type, or compute. For agents, descriptors might include number of tool calls, number of roles, memory usage, or planning strategy. For code, descriptors might include algorithmic family, asymptotic complexity proxy, dependency footprint, or supported input regime. MAP-Elites [39] makes these descriptors first-class, preventing the loop from collapsing to a single over-specialized artifact.

The search loop faces several failure modes. The first is benchmark overfitting: candidates improve by exploiting quirks of the evaluator rather than solving the underlying task. The second is mode collapse: the generator repeatedly proposes minor variations of the same design. The third is invalidity: many candidates fail to run, compile, or satisfy interface constraints. The fourth is cost explosion: evaluation dominates runtime. The fifth is hidden regression: a candidate improves the target metric while breaking other tasks. These failure modes motivate the evaluator and population loops. Strong validators reduce invalidity; archives preserve diversity; regression suites detect hidden losses; and multi-objective selection discourages narrow reward hacking.

A useful way to summarize search methods is by their unit of variation. OPRO [66] varies

Algorithm 2: LLM-Guided Search Loop

1. **Input:** initial artifact x_0, search space $\mathcal{X}$, generator g, evaluator e, budget B, history H.
2. Initialize incumbent set $P \leftarrow \{x_0\}$ and evaluate $F \leftarrow e(P)$.
3. For iteration $t = 1$ to B:
   (a) Build a search context from selected history: top candidates, diverse candidates, failures, and constraints.
   (b) Generate candidates $C_t \leftarrow g(P, H, \text{context})$ by mutation, recombination, or fresh proposal.
   (c) Reject candidates that violate syntax, schema, safety, or budget constraints.
   (d) Evaluate remaining candidates with e to obtain scores, traces, and diagnostics.
   (e) Update history H with candidates, feedback, and lineage.
   (f) Select the next incumbent set $P \leftarrow s(P \cup C_t, F)$ according to performance and diversity.
4. **Return:** best artifact, archive, and search log.

Figure 5: A generic search loop for prompts, architectures, agents, code, or hyperparameters.

strings or prompts. EvoPrompting [61] varies architecture code. ADAS [21] varies agent systems. OpenEvolve [2]-like systems vary code patches. AlphaEvolve [40] varies algorithms and programs under executable evaluation. The unit of variation determines the rest of the system: the parser, evaluator, safety wrapper, lineage representation, and update mechanism. A mature self-evolution platform should make this unit explicit so that users know what is allowed to change.

2.4 Evaluator Loop

The evaluator loop turns behavior into fitness. Without evaluation, self-evolution is only self-modification. An evaluator can be a benchmark, test suite, compiler, simulator, verifier, human rating, model-based judge, environment reward, deployment metric, or safety monitor. The evaluator loop repeatedly runs candidates, extracts evidence, and converts that evidence into feedback used by selection and update. It is the most important loop for reliability because it defines what counts as progress.

FunSearch [47] is a flagship evaluator-loop system. It combines an LLM that proposes programs with an evaluator that executes and scores them on mathematical or scientific objectives. The LLM supplies creative variation, but the evaluator supplies discipline. Candidate programs survive because they produce measurable improvements, not because the model claims they are good. This pattern is central to AI self-evolution: the more open-ended the generator, the more important it becomes to constrain it with executable feedback. FunSearch also shows that programs are attractive candidates because they can be run, scored, and archived.

Self-Debug [10] demonstrates evaluator feedback at the level of code repair. A model writes code, observes execution results or errors, explains what went wrong, and revises the program. The evaluator may be a unit test, interpreter, compiler, or natural-language problem specification paired with examples. The key is that failure is not only a scalar penalty. It is an informative trace:

Algorithm 3: MAP-Elites for Self-Evolving AI Artifacts

1. **Input:** descriptor function $b(x)$, fitness evaluator $f(x)$, archive grid $\mathcal{A}$, generator g, budget B.
2. Initialize archive $\mathcal{A}$ with empty cells.
3. Sample or generate initial candidates C_0; evaluate each candidate's fitness $f(x)$ and descriptor $b(x)$.
4. For each candidate x, place it into cell $\mathcal{A}[b(x)]$ if the cell is empty or if $f(x)$ exceeds the current elite's fitness.
5. For iteration $t = 1$ to B:
   (a) Sample one or more parent elites from occupied archive cells.
   (b) Generate variants $C_t \leftarrow g(\text{parents}, \text{archive summaries})$ using mutation, crossover, or LLM-guided redesign.
   (c) Evaluate candidates to obtain $f(x)$ and $b(x)$.
   (d) Insert each candidate into its descriptor cell if it improves the cell elite.
   (e) Optionally expand, merge, or reweight cells when new task niches are discovered.
6. **Return:** an archive of elites rather than a single best candidate.

Figure 6: MAP-Elites maintains high-performing candidates across behavioral or structural niches, making it attractive for self-evolving agents and code systems.

an exception, wrong output, missing condition, or violated invariant. The generator can use that trace to make targeted changes. This makes Self-Debug a bridge between evaluator and reflection loops.

CodeEvolve [24]-style systems make evaluation a repeated engineering process. Candidate code modifications are generated and tested; passing or improved candidates are retained; failing candidates are summarized and used to guide new proposals. The loop resembles continuous integration, but with an LLM in the candidate-generation step and with search over many alternatives. The evaluator can include unit tests, integration tests, linting, type checking, benchmark timing, memory usage, correctness checks, and differential comparison against an incumbent. The central design principle is that each accepted update must have evidence.

SelfEvolve [25] systems, broadly understood, use evaluators to support self-improvement across tasks and iterations. Depending on the implementation, the evaluator may score final answers, trajectories, code patches, agent workflows, or learned policies. The important taxonomic point is that the evaluator is not an afterthought. It is part of the state transition. The system evolves by internalizing evaluation results into future generation, selection, or training. A self-evolving method with no reliable evaluator risks drifting toward self-reinforcing hallucination.

Evaluator design involves four questions. First, what is measured? A classifier can be measured by accuracy, F1, calibration, fairness, robustness, cost, and latency. An agent can be measured by task success, number of tool calls, time, safety violations, and user satisfaction. A code patch can be measured by tests passed, speed, memory, maintainability, and security. Second, when is it measured? Evaluation can occur before execution, during execution, after execution, or after

deployment. Third, how is feedback represented? It can be scalar, categorical, textual, structured, or trace-based. Fourth, who trusts the evaluator? If the evaluator is itself an LLM judge, the system needs calibration, adversarial tests, or human review for high-stakes updates.

A strong evaluator loop often has multiple layers. The first layer checks basic validity: syntax, schema, dependency availability, permissions, and resource limits. The second layer checks functional correctness: unit tests, examples, benchmark tasks, or simulation outcomes. The third layer checks regression: the candidate should not degrade known capabilities. The fourth layer checks non-functional properties: cost, latency, memory, interpretability, fairness, security, and compliance. The fifth layer checks generalization: held-out tasks, randomized tests, adversarial cases, or cross-domain transfer. Selection can then use a vector of metrics rather than a single score.

Mathematically, an evaluator can be written as

$$e(c) = \big(r(c), v(c), \tau(c), q(c), \sigma(c)\big), \tag{18}$$

where $r(c)$ is a reward or fitness score, $v(c)$ is a validity flag, $\tau(c)$ is an execution trace, $q(c)$ is a quality vector, and $\sigma(c)$ is a safety or constraint report. Selection should rarely depend on $r(c)$ alone. A candidate with high task reward but invalid safety status should be rejected or quarantined. A candidate with promising reward but high variance may need repeated evaluation. A candidate with low reward but novel behavior may be stored in a diversity archive.

Evaluation also has a sampling problem. A benchmark is a finite sample from a task distribution. Improving benchmark score does not guarantee improving the real task. This is especially dangerous in self-evolving systems because repeated search can exploit benchmark artifacts. Mitigations include hidden test sets, rotating tasks, adversarial generation of new tests, cross-validation, human audits, benchmark versioning, and explicit generalization checks. For code, property-based testing and fuzzing can complement fixed unit tests. For agents, scenario randomization can reduce overfitting to scripted tasks. For prompts, evaluation across paraphrases and domains can reveal fragility.

The evaluator loop is also where cost enters. Some candidates are cheap to score: a prompt can be tested on a small validation set; a syntax check is fast. Others are expensive: training a neural architecture, running a long-horizon agent task, or evaluating a large codebase benchmark. Practical systems therefore use cascaded evaluation. Cheap filters remove invalid candidates; medium-cost tests rank plausible candidates; expensive benchmarks are reserved for finalists. The loop can allocate budget adaptively, giving more evaluations to candidates with high uncertainty or high potential.

Another important distinction is between *diagnostic* and *decisive* evaluation. Diagnostic evaluation helps the generator understand failures, while decisive evaluation determines acceptance. A model-generated critique may be useful diagnostically but insufficient decisively. A failing unit test is both diagnostic and decisive: it explains a failure and blocks acceptance. Human feedback can be decisive for product taste but expensive and inconsistent. A well-designed self-evolution system should label evaluators by authority. For example, an LLM judge may suggest revisions, but a compiler, test suite, safety policy, or human reviewer may gate deployment.

Evaluator loops can themselves evolve. A system can generate new tests, discover counterexamples, refine benchmarks, calibrate judges, or learn better reward models. This creates a meta-evaluation problem: who evaluates the evaluator? If test generation is also optimized, the system may learn either to create more informative tests or to create tests that are easy for its candidates to pass. The solution is to separate roles, preserve held-out validation, maintain adversarial test sets, and audit changes to evaluators. In high-stakes settings, evaluator updates should be more conservative than candidate updates.

The evaluator loop connects directly to safety. Self-modifying systems can amplify mistakes. An incorrect update can persist, influence future candidates, and become harder to diagnose. Evaluators should therefore include rollback conditions and canary tasks. Before replacing an incumbent, a candidate should be compared against the incumbent on a suite of known tasks. If the candidate wins narrowly on the target but loses significantly elsewhere, selection should either reject it, store it as niche-specific, or require human review. This is where population methods are preferable to global replacement: they can keep specialized improvements without overwriting the default system.

In summary, the evaluator loop is the empirical backbone of self-evolution. It defines the fitness landscape, produces feedback, protects against regressions, and determines whether updates are justified. Systems such as FunSearch [47], Self-Debug [10], CodeEvolve [24], and SelfEvolve [25] show different evaluator strengths, from executable program scoring to test-driven repair and trajectory-level feedback. The stronger and more structured the evaluator, the more ambitious the update can be.

2.5 Reflection Loop

The reflection loop converts failure into reusable information. It is the most language-native form of self-evolution because the update can be a natural-language memory, critique, rule, rationale, or revised instruction. The loop observes an attempt, identifies what went wrong, writes a reflection, and conditions future attempts on that reflection. Unlike pure search, reflection does not require a large candidate population. Unlike weight training, it does not require gradient updates. Its strength is low cost and interpretability; its weakness is that natural-language memories can be incomplete, wrong, stale, or overgeneralized.

Reflexion [49] is the canonical example. An agent attempts a task, receives feedback from the environment, and writes a verbal reflection that is stored in memory. On the next attempt, the agent reads the memory and changes its behavior. This has been described as a form of verbal reinforcement learning: the reward does not directly update neural weights, but it updates the context that influences future actions. In the unified formalization, e provides task feedback, s determines which failure information is worth remembering, and u appends or revises memory h_t.

Self-Refine [36] uses a related but more local loop. A model generates an output, critiques it, and refines it, often with the same model playing all roles. The update is usually the current answer rather than persistent memory. Self-Refine is therefore a minimal self-evolution loop: it improves an artifact through feedback, but the improvement may not persist beyond the instance. It is still taxonomically important because it establishes the generate–feedback–revise pattern that many larger systems reuse. When Self-Refine-style critiques are stored across tasks, the method moves closer to Reflexion.

STaR [75], or self-taught reasoner, illustrates reflection at the level of rationales and training data. A model generates rationales, filters or corrects them based on answers, and uses successful rationales to improve future reasoning. The loop can be interpreted as turning attempts into supervision. Unlike Reflexion, where memory is often retrieved at inference time, STaR-style methods can move reflected experience into a training set and eventually into model weights. The boundary between reflection and training is therefore porous: reflection can create data, and data can support weight-level self-improvement.

Agent Symbolic Learning [1] frames agent improvement as textual feedback propagation through symbolic components. Instead of updating continuous weights with gradients, the system can update prompts, rules, tool descriptions, or agent modules using language feedback. This is especially relevant for multi-agent systems, where failures may be localized to roles or communication pro-

tocols. A planner may decompose tasks poorly; a coder may ignore tests; a reviewer may miss edge cases; a router may choose the wrong specialist. Reflection becomes more precise when it is assigned to the component that caused the failure.

A reflection loop has five steps: capture, diagnose, compress, store, and retrieve. Capture records the task, actions, observations, evaluator feedback, and final outcome. Diagnosis identifies causes of success or failure. Compression turns the trace into a reusable lesson. Storage places the lesson in memory with metadata. Retrieval selects relevant lessons for future attempts. Each step can fail. If capture is incomplete, diagnosis is speculative. If diagnosis is wrong, memory becomes harmful. If compression is too vague, future agents cannot act on it. If storage lacks metadata, retrieval returns irrelevant lessons. If retrieval is too broad, the prompt becomes cluttered.

The reflection update can be formalized as

$$h_{t+1} = \mathrm{Store}\big(h_t,\ \rho(x_t, a_t, \tau_t, e_t)\big), \tag{19}$$

where h_t is memory, x_t is the task, a_t is the attempted action or artifact, τ_t is the trace, e_t is feedback, and ρ is a reflection function. Retrieval then conditions generation:

$$C_{t+1} = g\big(z_t, \mathrm{Retrieve}(h_{t+1}, x_{t+1})\big). \tag{20}$$

This representation makes clear that reflection is only useful if retrieval is relevant. A large memory that is never retrieved is inert; a memory that is always retrieved becomes noise.

Reflection differs from ordinary logging. A log records what happened. A reflection extracts a lesson that can change future behavior. For example, a log might say that a unit test failed with an off-by-one error. A reflection might say: "When generating range loops for inclusive problem statements, explicitly test both endpoints before finalizing." A stronger reflection might attach this lesson to a code-generation component and retrieve it only for tasks involving ranges. The more actionable and scoped the reflection, the more useful it is.

Reflection also differs from self-critique. A critique can be generated before execution and may be purely speculative. A reflection should be grounded in feedback. Self-critique is useful for catching obvious inconsistencies, but ungrounded critique can reinforce model biases or invent problems. Reflexion's [49] contribution is to tie reflection to environmental feedback. Self-Debug [10] similarly grounds reflection in execution. Agent Symbolic Learning [1] grounds feedback in task performance and component structure. The taxonomy therefore treats reflection as strongest when it is evaluator-grounded.

A mature reflection loop should manage memory lifecycle. Memories can be added, merged, revised, deprecated, or deleted. Early failures may become irrelevant after a component is fixed. A lesson that helped one domain may harm another. Memories may conflict. The update function should therefore include confidence, scope, timestamp, source evaluator, and evidence. Selection over memories can use recency, relevance, success rate, and contradiction detection. Some systems may maintain separate episodic memory, semantic lessons, procedural rules, and benchmark-specific notes.

Reflection can improve sample efficiency. Instead of rediscovering the same failure across iterations, the system can avoid known mistakes. This is especially valuable when evaluations are expensive. For long-horizon agents, each failed run may consume many tool calls, minutes, or dollars. A concise reflection can prevent repeated dead ends. In AutoML, reflection can remember that a dataset has leakage risks or that a model family underperformed. In code evolution, reflection can record that a benchmark requires deterministic seeding or that a previous optimization broke edge cases.

The weakness of reflection is that it is not guaranteed to be true. LLM-generated explanations can be post hoc rationalizations. A system may misattribute failure to the wrong component. It may store a lesson that is too broad: "always use tool X" when tool X only helped once. It may overfit to a benchmark: "prefer shorter answers" because a local judge rewarded brevity. To mitigate these risks, reflection should be validated. A reflection can be treated as a hypothesis that must earn confidence across future tasks. If following a memory improves outcomes, confidence rises; if it hurts, confidence falls.

Reflection is also a bridge to population methods. Different candidates can carry different memories. An archive may store not only code or prompts but also learned lessons associated with each lineage. A candidate agent that evolved for mathematical reasoning may carry memories about proof strategies, while another evolved for web tasks may carry memories about navigation. Population-level reflection prevents a single global memory from becoming contradictory. It supports specialization.

The reflection loop is thus the memory system of self-evolution. It makes failures reusable, explanations inspectable, and improvements cheaper. Reflexion [49], Self-Refine [36], STaR [75], and Agent Symbolic Learning [1] occupy different points on the persistence spectrum: output-level revision, episodic memory, rationale-derived training data, and component-level textual updates. Together they show that language is not only an interface for generation; it is also a medium for storing evolutionary experience.

2.6 Population Loop

The population loop maintains multiple candidates over time. Instead of updating a single incumbent, it keeps a set, archive, or distribution of artifacts. It selects parents, generates variants, evaluates offspring, and updates the population. Population methods are natural for self-evolution because AI artifacts often have multiple local optima. A single best agent may not exist across all tasks, budgets, and domains. An archive can preserve diversity, specialization, and lineage.

In classical evolutionary computation [45,52], the population contains genomes. In self-evolving AI, genomes can be prompts, architectures, programs, agent workflows, model checkpoints, memory states, or combinations of these. Mutation may be an LLM edit, prompt rewrite, code patch, hyperparameter perturbation, tool substitution, or memory revision. Crossover may combine prompts, merge code ideas, compose agent roles, or synthesize a new program from two parents. Selection may be tournament-based, rank-based, Pareto-based, novelty-based, or archive-based. The key difference from older evolutionary algorithms is that variation can be semantic: an LLM can understand and transform the candidate representation.

AlphaEvolve [40] is a representative population-loop system because it maintains and improves code candidates through evolutionary mechanisms. Its archive can preserve high-performing programs, while the generator proposes new variants conditioned on previous successes and failures. The evaluator supplies executable fitness. The population loop prevents the system from betting everything on one candidate. It can keep several promising approaches, compare them, and use them as parents for future improvements.

LLaMEA [6] shows the population loop at the meta-heuristic level. Instead of evolving solutions to a single optimization problem, it evolves algorithms that solve optimization problems. This is a deeper form of self-evolution: the system improves the optimizer, not just the object being optimized. In the unified formalization, the state includes a population of algorithm descriptions or code. The evaluator runs those algorithms on benchmark problems. Selection keeps algorithms that perform well. Mutation and recombination generate new algorithmic ideas. The result is a loop that searches over search procedures.

DGM [76], or Darwin Gödel Machine [48], illustrates open-ended code-level agent evolution. The system maintains an archive of self-modified agents, evaluates them on tasks, and uses successful variants as stepping stones for future modifications. Its importance is conceptual: self-evolution need not be a linear sequence of replacements. It can be an expanding archive of agents that modify their own code under evaluation. The archive makes it possible to return to earlier branches, preserve diversity, and discover stepping stones that a greedy hill climber would discard.

SPIN [11], or self-play fine-tuning, represents a population-like dynamic at the level of model behavior and training distributions. A model improves by competing with or learning from its own generated responses, creating a self-play signal. Although SPIN is not a classical genetic population, it shares the idea that improvement arises from a distribution of generated behaviors and selection pressure. The update moves into model weights rather than only prompts or archives. This makes evaluation and safety more consequential because the change can affect all future outputs.

Absolute Zero [77] represents a self-play and self-generated-task direction in which a system can create training signals without relying on external labeled data. It is relevant to the population loop because tasks, solutions, and policies can co-evolve. A model can generate problems, attempt solutions, verify outcomes, and learn from the resulting curriculum. This resembles open-ended evolution: the environment or task distribution is no longer fixed. The challenge is to avoid degenerate curricula, trivial tasks, or reward hacking. Strong validators and diversity mechanisms become essential.

The population loop can be described as

$$\mathcal{P}_{t+1} = U_{pop}\big(\mathcal{P}_t,\ s(\mathcal{P}_t \cup g(\mathcal{P}_t),\ e(g(\mathcal{P}_t)))\big), \tag{21}$$

where $\mathcal{P}_t$ is the population or archive. If U_{pop} always keeps only the single best candidate, the loop collapses to hill climbing. If it preserves elites across niches, it becomes quality-diversity search. If it maintains a probability distribution over candidates, it resembles estimation-of-distribution or policy-search methods. If it updates neural weights from selected samples, it becomes a training loop.

Population methods offer several benefits. First, they support diversity. Different candidates can specialize for different tasks, constraints, or risk profiles. Second, they support robustness. If a new candidate fails, the system can fall back to older elites. Third, they support lineage analysis. Designers can inspect which mutations led to improvements. Fourth, they support recombination. Ideas from separate lineages can be combined. Fifth, they support open-endedness. A population can continue to discover new niches rather than converging to one artifact.

Population methods also introduce complexity. They require storage, indexing, and comparison. Candidate artifacts may be large: code repositories, model checkpoints, or agent graphs. Evaluation may be expensive. Lineage can become difficult to interpret if changes are large. Crossover over code or prompts may produce invalid artifacts. Selection can prematurely reduce diversity if the fitness metric is too narrow. A population archive therefore needs metadata: parent identifiers, mutation descriptions, evaluator versions, benchmark scores, descriptors, safety status, and deployment eligibility.

The population loop is closely related to MAP-Elites. A standard population may preserve diversity implicitly, but MAP-Elites makes diversity explicit through descriptor cells. This matters for self-evolving agents because descriptors can represent capabilities. One cell might store the best low-latency coding agent; another stores the best high-accuracy but expensive agent; another stores the best agent for data cleaning; another stores the best proof-oriented agent. Instead of asking which agent is universally best, the system asks which agent is best for each niche.

Crossover is underexplored in LLM-based self-evolution. Traditional crossover swaps genome segments. For text and code, naive swapping is often invalid. LLMs make semantic crossover

possible: the model can read two parent candidates, identify complementary ideas, and synthesize a coherent child. For example, one agent may have a strong planning prompt, while another has a strong verification step. A semantic crossover operator can combine the planning strategy with the verification module. The evaluator then determines whether the combination actually improves performance.

Population-level safety is both easier and harder than single-incumbent safety. It is easier because the system does not need to replace the incumbent immediately; risky candidates can be quarantined. It is harder because the archive may contain unsafe or partially validated artifacts. The system needs access control, sandboxing, and metadata to prevent untrusted variants from being deployed. Selection should distinguish between research archive status and production status. A candidate may be valuable as a stepping stone even if it is not safe to deploy.

The population loop also changes how progress is measured. A single best-score curve is insufficient. We need archive coverage, diversity, best-per-niche performance, transfer performance, regression rates, and lineage depth. An archive that discovers many moderate candidates may be more valuable than one that slightly improves a single score. For open-ended systems, novelty and coverage are first-class metrics.

In summary, the population loop is the mechanism that turns self-evolution from iterative repair into an ecosystem of alternatives. AlphaEvolve, LLaMEA, DGM, SPIN, and Absolute Zero illustrate different population-like dynamics: code archives, algorithm populations, self-modifying agent lineages, self-play distributions, and self-generated curricula. The common theme is that improvement is distributed across candidates rather than concentrated in one mutable object.

2.7 Cross-Loop Interactions

The five loops become most powerful when composed. A specification-to-execution loop produces artifacts that search can vary. Search produces candidates that evaluators score. Evaluators produce failures that reflection stores. Reflection improves future generation. Population methods preserve multiple lineages and prevent premature collapse. The result is a compound system in which each loop compensates for another loop's weakness.

AlphaEvolve can be summarized as Search + Population + Evaluator. The search loop proposes program variants. The evaluator loop executes and scores them. The population loop stores and selects among candidates. Reflection may also appear if traces or failure summaries are fed back into prompts, but the defining composition is the triad of generation, executable fitness, and archive-based evolution. This composition is strong because no single LLM judgment decides progress. The LLM explores; the evaluator measures; the archive preserves.

FunSearch is Evaluator + Search with an evolutionary archive. It is narrower than a general agent-evolution platform because the target artifacts are programs for well-specified objectives. That narrowness is a strength: mathematical programs can be executed and scored. The system demonstrates that LLM creativity becomes more reliable when coupled to deterministic or high-quality evaluators. It also shows that search over code can produce discoveries rather than merely solve benchmark prompts.

Reflexion is Evaluator + Reflection. The environment or task feedback indicates failure or success. The reflection loop writes a memory that changes future attempts. If multiple attempts or strategies are maintained, Reflexion-like systems can also acquire a population dimension, but the central mechanism is memory update. Compared with AlphaEvolve, Reflexion has weaker artifact persistence but cheaper updates. It is therefore suitable for agents operating across repeated tasks where full code evolution would be too costly or risky.

Self-Refine is Generation + Critique + Revision. It is a small loop that can be embedded inside

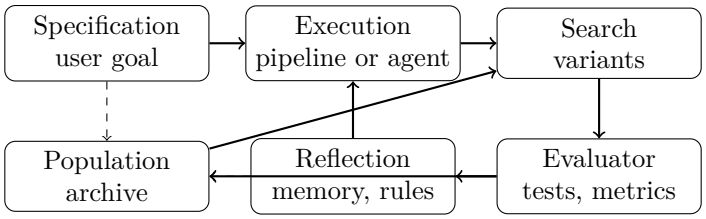

Figure 7: Cross-loop composition. Specification creates executable artifacts; search varies them; evaluators score them; reflection stores lessons; population archives preserve alternatives.

larger loops. For example, a search system can use Self-Refine to polish each candidate before evaluation. An AutoML planner can generate a pipeline, critique it for missing validation, and revise it before execution. A code-evolution agent can ask for a self-critique before running tests. However, self-critique should not replace external evaluation. It is best treated as a cheap pre-filter or diagnostic aid.

AutoML-Agent is Specification-to-Execution + Evaluator + Reflection, with possible Search. The system begins from a user goal, constructs a pipeline through specialized agents, evaluates the pipeline, and can revise based on feedback. If it maintains alternatives or searches over model choices, it includes a search loop. If it stores lessons across tasks, it includes reflection. If it maintains multiple agent configurations, it approaches a population loop. This illustrates a general pattern: production systems often start as specification-to-execution frameworks and gradually add self-evolution features.

ADAS is Specification-to-Execution + Search + Evaluator over agent architectures. It treats agent design itself as the object of optimization. The specification defines tasks and constraints; the generator proposes agent systems; the evaluator scores them; selection keeps better designs. If ADAS maintains an archive of agents, it also becomes population-based. Its key contribution to the taxonomy is shifting the evolvable unit from outputs or prompts to agent architecture.

DGM is Search + Population + Evaluator + Code Update. It is one of the clearest examples of self-modification because candidates can alter agent code. The archive stores lineages. The evaluator measures task performance. The generator proposes modifications. The update creates new executable variants. Reflection may be present if failure information is summarized for later proposals. DGM highlights why rollback and sandboxing are essential: code-level updates are more powerful and riskier than prompt-level updates.

LLaMEA is Search + Population + Evaluator at the optimizer level. Its candidates are algorithms for optimization, so the system evolves the mechanism of search itself. This is a meta-evolutionary composition. It suggests a future direction for self-evolving agents: instead of only improving task agents, systems may improve the generators, evaluators, selection policies, and memory mechanisms that drive self-evolution.

SPIN and Absolute Zero show how reflection, population, and evaluator loops can connect to training. Generated behaviors or tasks create a distribution of experiences. Verification or self-play provides selection pressure. The update changes model weights. This is a more persistent and less transparent form of evolution. It can produce broad capability changes, but it also raises evaluation burdens. Once an update enters weights, it is harder to attribute behavior to a specific memory or code patch.

The following diagram shows one common composition pattern for a self-evolving agent platform.

Cross-loop composition creates several design tensions. The first is speed versus certainty.

Reflection and self-critique are cheap but less reliable. Full benchmark evaluation is reliable but expensive. Population archives preserve alternatives but increase storage and evaluation cost. A practical system uses a cascade: cheap reflection for rapid iteration, tests for candidate filtering, benchmarks for selection, and human review for high-impact updates.

The second tension is specialization versus generality. Search and population loops can create specialized agents that excel in narrow niches. Specification-to-execution systems often need general-purpose behavior. If selection always favors average performance, specialized innovations may disappear. If selection always favors niche performance, the system may fragment. MAP-Elites and multi-objective selection address this by preserving best-per-niche candidates while maintaining a default generalist.

The third tension is plasticity versus stability. A highly plastic system changes quickly in response to feedback. A stable system resists changes that might cause regressions. Prompt updates are plastic; weight updates are persistent; code updates are structural. A mature self-evolution platform should make update levels explicit and require stronger evidence for more persistent changes. For example, a reflection memory might be added after one failure, but a code patch might require tests, and a model-weight update might require a full evaluation suite.

The fourth tension is autonomous improvement versus governance. Self-evolution suggests autonomy, but real systems need human oversight, policy constraints, and auditability. Cross-loop systems should expose their state transitions. A user or reviewer should be able to inspect what candidate was generated, how it was evaluated, why it was selected, what changed, and what rollback is available. This is especially important when the system modifies code, tools, permissions, or deployment behavior.

The fifth tension is evaluator evolution. If the system can improve candidates, it may also need to improve evaluators. But evaluator changes can invalidate score comparisons across time. A benchmark version change may make older lineages incomparable. The solution is to version evaluators, store raw traces where possible, re-evaluate elites under new suites, and separate candidate evolution from evaluator governance. In scientific discovery systems, evaluator correctness may be the limiting factor; in agent systems, evaluator coverage may be the limiting factor.

A useful architecture pattern is the "evolution ledger." The ledger stores every state transition: candidate ID, parent IDs, generator prompt or policy, evaluator version, scores, traces, selected update, memory changes, and deployment status. This ledger supports reproducibility, safety review, and meta-learning. It also enables later analysis: which generators produce useful diversity, which evaluators predict deployment success, which memories reduce repeated failures, and which lineages produce robust improvements.

Another pattern is the "safe baseline freeze." Before self-evolution begins, the system records an incumbent baseline and evaluation suite. Candidate updates are compared against this baseline. Regressions trigger rollback or niche-specific storage rather than global replacement. This pattern is important because self-evolution can otherwise create hidden drift. A system may look improved on recent tasks while losing older capabilities. Baseline freezing and regression gates make progress relative and auditable.

A third pattern is "relative-error control." Instead of asking only whether a candidate improved a mean score, the system measures where errors changed. Did the candidate fix previous failures? Did it introduce new failures? Are improvements statistically stable across seeds or task variants? This matters for stochastic LLM and agent systems. A single lucky run should not drive persistent updates. Repeated evaluation and uncertainty estimates should influence selection.

Cross-loop systems can be analyzed by the strongest evaluator in the composition and the deepest update they allow. A system with weak self-critique and prompt-only updates is low risk but limited. A system with executable tests and code updates is more powerful and riskier. A

system with self-play verification and weight updates can change broad behavior and therefore needs the strongest governance. This evaluator-update alignment is one of the main design principles of self-evolving AI.

2.8 Comparison Table

Table 4 summarizes representative methods by loop type, trainability, feedback source, and tasks. The categories are intentionally coarse. Many systems span several loops, and implementations vary. The goal is not to force a single label but to show which mechanism is primary.

Table 4: Representative methods mapped to loop type, trainability, feedback source, and task domain. Several methods compose multiple loops; the table lists the dominant interpretation for taxonomy purposes.

Method	**Primary loop type**	**Trainable?**	**Feedback source**	**Representative tasks**
AutoML-GPT	Specification-to-execution; search over ML pipeline choices	Usually no weight update; may call training jobs	Task cards, data cards, model cards, training logs, validation metrics	AutoML planning, preprocessing, model selection, hyperparameter choices
AutoML-Agent	Specification-to-execution; evaluator; reflection across stages	Mostly agent, prompt, and pipeline updates	Multi-stage validation, dataset metrics, deployment checks, agent critiques	Full-pipeline AutoML, data processing, model building, evaluation, deployment
ADAS	Search over agent architectures; evaluator; possible population	Agent code, prompt, and workflow trainability; usually no base-model training	Benchmark task performance, execution traces, success rates	Automated design of agentic systems, tool-using agents, task-specialized workflows
OPRO	Search loop over textual candidates	No model training required	Scalar scores placed in optimization prompt	Prompt optimization, mathematical optimization, black-box objective tuning
EvoPrompting	Search loop over architecture code	Candidate architectures trained and evaluated; LLM not necessarily trained	Task accuracy or benchmark performance of generated architectures	Code-level NAS, graph neural architectures, reasoning tasks
OpenEvolve	Search and evaluator loop over code artifacts	Code evolves; base model often fixed	Tests, benchmarks, execution results, performance metrics	Repository optimization, algorithm improvement, code repair
AlphaEvolve	Search + population + evaluator	Code/archive evolves; model may be fixed	Executable evaluators, benchmark scores, archive comparisons	Algorithm discovery, code optimization, scientific or engineering programs
FunSearch	Evaluator-driven program search with evolutionary archive	Program population evolves; base LLM fixed	Program execution and objective score	Mathematical discovery, combinatorial constructions, scientific programs
Self-Debug	Evaluator + reflection for code repair	No necessary weight update; output/code revised	Execution errors, tests, examples, natural-language debugging feedback	Program synthesis, bug fixing, code explanation and repair

Table 4: Representative methods mapped to loop type, trainability, feedback source, and task domain (continued).

Method	**Primary loop type**	**Trainable?**	**Feedback source**	**Representative tasks**
CodeEvolve	Evaluator-guided code evolution	Code evolves; possible prompt and memory updates	Unit tests, integration tests, benchmark metrics, traces	Software repair, performance optimization, generated code improvement
SelfEvolve	General self-improvement loop; evaluator + reflection/search	Varies: prompt, memory, code, or weights	Task feedback, benchmark scores, validators, self-generated data	Agent improvement, reasoning, code, tool use, benchmark adaptation
Reflexion	Reflection loop grounded in evaluator feedback	Memory update; no weight update required	Environment reward, task success/failure, verbal reflections	Agent tasks, reasoning, decision making, code generation
Self-Refine	Inference-time reflection/revision	No weight update	LLM-generated critique or feedback prompts	Text generation, reasoning, code, dialogue, style refinement
STaR	Reflection to rationale data; training loop	Yes; can fine-tune on rationales	Correct answers, generated rationales, filtering	Reasoning tasks, chain-of-thought improvement, rationale bootstrapping
Agent Symbolic Learning	Reflection and search over symbolic agent components	Updates prompts, rules, and modules, not necessarily weights	Textual feedback, task loss, component-level critiques	Multi-agent systems, prompt modules, agent architecture improvement
LLaMEA	Population loop over meta-heuristic algorithms	Algorithm population evolves	Benchmark optimization performance	Automatic design of meta-heuristics and optimizers
DGM	Population + search + evaluator with code self-modification	Agent code evolves; possible archive growth	Benchmarks, task success, code execution, lineage performance	Self-modifying coding agents, open-ended agent improvement
SPIN	Self-play/training loop with population-like generated behavior	Yes; model fine-tuning	Self-play comparisons, generated responses, training objectives	Language-model improvement, preference-like self-play, reasoning behavior
Absolute Zero	Self-play, self-generated tasks, evaluator, training	Yes; policy/model updates	Verifiable self-generated tasks, code execution or task validators	Zero-data reasoning/code improvement, curriculum generation, RL-style self-improvement

The table reveals several trends. First, early and lightweight methods operate at prompt or output level. Self-Refine and Reflexion require no model training and can be applied to many tasks, but their updates are limited to current outputs or memory. Second, code and program-search methods gain power from executable evaluation. FunSearch, Self-Debug, CodeEvolve, OpenEvolve, and AlphaEvolve can use tests or objective functions, making their feedback more reliable than self-critique alone. Third, agent-design methods move the unit of evolution from answers to workflows. ADAS and AutoML-Agent show that roles, tools, and control flow can be searched or revised. Fourth, population and training methods increase persistence. DGM and LLaMEA maintain archives; SPIN and Absolute Zero can update weights or policies.

The comparison also shows why "trainable" is not a binary property. A system can train a candidate model while keeping the LLM generator fixed. AutoML-GPT may train ML models as part of generated pipelines but does not necessarily train the planner. EvoPrompting may evaluate trained architectures while the language model remains unchanged. STaR and SPIN update model weights. Reflexion updates memory. DGM updates code. Population methods update archives. A survey of self-evolution should therefore specify the update substrate rather than simply asking whether the method trains.

Feedback sources form a hierarchy. Self-feedback is cheap but weak. Human feedback is meaningful but expensive and variable. Benchmark feedback is repeatable but can be overfit. Execution feedback is strong for code but only covers specified properties. Deployment feedback is realistic but risky and delayed. The best systems combine feedback sources. For example, an agent may use self-critique before running, tests during development, benchmark suites before acceptance, and human review before deployment.

The table also suggests a research gap: many methods optimize candidates but do not provide a complete governance layer. They may lack evaluator versioning, lineage records, rollback, safety constraints, or cross-task regression. As self-evolution moves from papers to production systems, these engineering layers become part of the scientific object. A method that improves a score without reproducible lineage is difficult to trust. A method that stores improvements without regression checks may degrade silently. A method that changes code or weights without audit trails is hard to govern.

2.9 Design Dimensions Across the Five Loops

Beyond method names, self-evolving systems can be compared along design dimensions. The first dimension is the *update substrate*: output, prompt, memory, workflow, code, data, archive, or weights. Output updates are temporary; prompt and memory updates are lightweight; workflow and code updates are structural; data and weight updates are statistical; archive updates are population-level. The update substrate determines persistence, reversibility, and safety burden.

The second dimension is the *candidate generator*. It can be a single LLM, a multi-agent planner, an evolutionary operator, a learned controller, a random sampler, a Bayesian optimizer, or a hybrid. LLM generators are flexible and semantic, but they need strong constraints. Evolutionary operators are simple and diverse, but may be inefficient without semantic guidance. Multi-agent generators can decompose tasks, but require coordination and can amplify communication errors. Hybrid generators are likely to dominate because they can combine structured search with language understanding.

The third dimension is the *fitness signal*. Fitness can be direct or proxy, dense or sparse, deterministic or noisy, scalar or structured. A direct deterministic evaluator, such as a compiler or exact checker, supports aggressive search. A noisy proxy evaluator, such as an LLM judge, requires caution and repeated validation. Sparse rewards make reflection and curriculum generation valuable. Structured traces make repair easier. Multi-objective fitness is often necessary because real artifacts must trade off task success, cost, latency, robustness, and safety.

The fourth dimension is *memory*. Some systems have no memory beyond the current prompt. Others store examples, reflections, logs, lineages, or learned parameters. Memory can be episodic, semantic, procedural, or archival. Episodic memory stores traces of attempts. Semantic memory stores generalized lessons. Procedural memory stores updated prompts, rules, or code. Archival memory stores candidate populations and their metadata. Weight memory stores experience in parameters. Each memory type has different retrieval and forgetting problems.

The fifth dimension is *selection pressure*. Greedy selection maximizes immediate score. Conser-

vative selection requires improvement beyond uncertainty. Diversity-preserving selection maintains niches. Risk-aware selection penalizes safety violations and regressions. Human-gated selection requires approval for certain updates. Curriculum selection chooses tasks that maximize learning. The selection policy shapes the system's long-term trajectory.

The sixth dimension is *autonomy*. Some loops are human-in-the-loop: users approve plans, edits, or deployments. Others are autonomous within a sandbox. Others are fully automated training loops. Autonomy should increase only with evaluator reliability and rollback capability. A system may autonomously revise a draft but require review for code changes, and require stricter governance for model-weight updates.

The seventh dimension is *openness*. A closed-loop system optimizes a fixed task set. An open-ended system can generate new tasks, new descriptors, new evaluators, or new goals. Open-endedness is attractive because it can discover unexpected capabilities, but it is difficult to evaluate. Absolute Zero-style self-generated tasks and DGM-style open-ended archives point in this direction. The challenge is to prevent drift into trivial, unsafe, or ungrounded objectives.

The eighth dimension is *lineage visibility*. In a self-evolving system, knowing that a candidate is good is not enough; we want to know where it came from. Lineage reveals which parent candidates, prompts, memories, or code changes produced an improvement. It supports reproducibility and debugging. It also enables meta-learning: the system can learn which mutation prompts or design patterns historically worked.

These dimensions help interpret methods that do not fit neatly into one loop. For example, a system may use Reflexion-style memory inside an ADAS search. Its primary loop might be search, but its memory dimension is reflection. A system may use MAP-Elites for code candidates and STaR-style rationale training for a generator. Its population and training dimensions are both important. The five-loop taxonomy is therefore a first layer; design dimensions provide a second layer.

2.10 Implications for Future Self-Evolving AI Systems

The five-loop taxonomy suggests several principles for building future systems. First, every self-evolving system should explicitly declare its evolvable substrate. Users and reviewers should know whether the system is allowed to change outputs, prompts, memories, workflows, code, data, archives, or weights. Hidden update channels make governance impossible. A system that claims to be a simple assistant but silently changes its own tool policies is riskier than one that openly maintains a versioned agent configuration.

Second, evaluator strength should match update depth. Prompt-level revisions can be guided by self-critique and lightweight tests. Memory updates need evidence and retrieval scope. Code updates need executable tests and rollback. Workflow updates need integration evaluation. Weight updates need broad benchmarks and safety audits. Population archive updates need metadata and quarantine status. This principle prevents a common failure: using weak feedback to justify strong updates.

Third, systems should preserve baselines and lineages. Self-evolution should not erase the path of improvement. Baselines make regressions visible. Lineages make improvements explainable. Archives make diversity recoverable. A candidate that is not selected globally may still be valuable later as a parent or niche specialist. This is why population methods and evolution ledgers are likely to become infrastructure rather than optional research details.

Fourth, reflection should be grounded and scoped. Natural-language memory is powerful because it is cheap and interpretable, but it can become harmful if unverified. Reflections should cite the feedback that produced them, indicate the component or domain to which they apply, and be

revisable. Systems should measure whether memories improve future outcomes. Memory should be treated as a hypothesis store, not an unquestioned truth store.

Fifth, search spaces should be designed, not merely opened. LLMs can edit almost anything, but unrestricted edit spaces are inefficient and unsafe. Good self-evolution systems define interfaces, templates, schemas, and constraints that make useful variation likely. AutoML systems use pipeline components; ADAS uses agent-design grammars or program templates; code-evolution systems use tests and patch boundaries; MAP-Elites uses descriptors. Structure helps the generator explore without producing invalid artifacts.

Sixth, self-evolution should include negative results. Failed candidates, rejected patches, broken prompts, and harmful memories are valuable data. They teach the generator what not to do, help evaluators identify blind spots, and prevent repeated mistakes. A system that stores only successes loses much of the evolutionary signal. However, negative results must be compressed and indexed so they do not overwhelm context.

Seventh, deployment should be separated from discovery. Population archives may contain experimental candidates that are useful for search but unsafe for users. The system should distinguish between discovered, validated, approved, and deployed states. A candidate can be a research elite without being production-ready. This distinction allows open-ended exploration while maintaining operational safety.

Eighth, cross-loop composition should be observable. When a system improves, observers should be able to tell whether the improvement came from better specification parsing, broader search, stronger evaluation, reflection memory, population diversity, or weight training. Without this decomposition, results are hard to reproduce and compare. The five-loop taxonomy provides a reporting template: describe the state, generator, evaluator, selector, update, and loop composition.

Finally, the field needs benchmarks that evaluate evolution rather than static performance. Traditional benchmarks ask how well a fixed model performs. Self-evolution benchmarks should ask how quickly and safely a system improves over a sequence of tasks, how well it preserves prior capabilities, how efficiently it uses evaluation budget, how robustly it generalizes, and how transparent its updates are. Metrics should include improvement curves, regression rates, archive diversity, update validity, cost, and human auditability.

2.11 Failure Modes, Safeguards, and Governance Across Loops

A taxonomy of self-evolution is incomplete without a taxonomy of failure. Every loop creates a characteristic risk because every loop creates a way for the system to change. The specification-to-execution loop can misunderstand the user's intent and faithfully optimize the wrong objective. The search loop can exploit quirks of the search space and produce brittle artifacts. The evaluator loop can reward shortcuts, leak test information, or miss important constraints. The reflection loop can store false lessons. The population loop can preserve unsafe variants or amplify a narrow selection pressure. These failures are not peripheral engineering details; they determine whether self-evolution produces durable improvement or uncontrolled drift.

The first major failure mode is *specification drift*. A user supplies an initial goal, but the system converts it into an internal task card that omits, distorts, or over-interprets part of the goal. Once this internal representation becomes the target of search, later iterations may optimize the wrong problem with increasing confidence. For example, an AutoML agent may treat accuracy as the only metric even though the user cares about calibration and interpretability. A coding agent may optimize benchmark runtime while ignoring maintainability. A research agent may maximize the number of collected sources rather than the quality of analysis. Specification drift is dangerous because all downstream loops can appear to work: candidates improve, evaluators report progress,

and reflections become more detailed, but the system is moving in the wrong direction.

Safeguards against specification drift include explicit task cards, assumption logs, human confirmation for ambiguous objectives, and metric pluralism. A task card should identify the objective, non-objectives, constraints, evaluation criteria, data assumptions, and deployment context. The system should record which assumptions were inferred rather than provided. When evaluator results are reported, they should be tied back to the original goal rather than only to internal metrics. In self-evolving systems, the task card itself may be updated, but such updates should be versioned and reviewed because changing the target can invalidate previous progress.

The second failure mode is *reward hacking*. A candidate may improve the measured fitness while violating the intended purpose. In program evolution, a candidate might exploit a test harness, hard-code answers, skip work, or rely on undefined behavior. In prompt optimization, a candidate may learn phrases that please an LLM judge without improving correctness. In agent benchmarks, a candidate may exploit environment artifacts or stop early to avoid penalties. Reward hacking is not unique to self-evolution, but self-evolution intensifies it because repeated search increases the chance of finding evaluator loopholes.

Safeguards against reward hacking include hidden tests, adversarial evaluation, randomized tasks, differential testing, invariant checks, and evaluator ensembles. For code, tests should include fixed examples, generated cases, and property-based checks. For agents, tasks should include varied scenarios and anti-cheating monitors. For LLM judges, outputs should be sampled across paraphrases and judged by multiple criteria. For AutoML, leakage checks and data-split validation are essential. The evaluator loop should treat suspiciously large improvements as signals for additional scrutiny rather than automatic success.

The third failure mode is *memory contamination*. Reflection loops can store inaccurate or overgeneralized lessons. A memory such as "always use a tree-based model" may help one tabular dataset and harm another. A reflection such as "the evaluator prefers concise answers" may be a local artifact rather than a general rule. A debugging lesson may become obsolete after the codebase changes. If contaminated memory is retrieved often, it can bias generation and selection across many future tasks. This turns a single mistaken reflection into a persistent system-level error.

Memory safeguards include evidence-linked reflections, scoped retrieval, confidence scores, expiration, contradiction checks, and performance-based memory validation. A memory entry should include the task context, evaluator evidence, component affected, and scope. Retrieval should prefer memories from similar tasks and avoid flooding the prompt with weakly related lessons. Memories should be revisable when new evidence contradicts them. The system can also run ablations: compare performance with and without a memory to estimate whether it actually helps. Treating memory as a hypothesis store rather than a fact store is a central governance principle.

The fourth failure mode is *population bloat*. Population systems can accumulate too many candidates, many of which are invalid, redundant, unsafe, or poorly documented. A large archive is not automatically valuable. Without descriptors, lineage, and selection discipline, it becomes a storage burden. Worse, an unsafe or low-quality candidate may later be selected as a parent because its metadata is incomplete. Population bloat also increases evaluation cost: re-testing all candidates under new benchmarks may become infeasible.

Population safeguards include archive schemas, descriptor-based indexing, quarantine states, pruning policies, and re-evaluation schedules. Each candidate should have parent IDs, mutation summaries, evaluator versions, scores, validity status, safety status, and intended niche. Candidates that fail safety checks should be stored separately or deleted, not mixed with deployable elites. Redundant candidates can be merged or pruned. Archive coverage should be measured, but coverage should not be confused with quality. A small, well-indexed archive can be more useful

than a large unstructured one.

The fifth failure mode is *regression masking*. A candidate improves on the current target but silently worsens old capabilities. This is common when selection uses a narrow score. A prompt optimized for one benchmark may lose generality. A code optimization may break rare edge cases. An AutoML pipeline may improve validation accuracy but degrade fairness. A model fine-tuned through self-play may become overconfident or less helpful outside the training distribution. Regression masking is dangerous because the system may report improvement honestly according to its current evaluator while still becoming worse in broader terms.

Regression safeguards include frozen baselines, canary tasks, multi-objective scorecards, statistical repeated runs, and rollback. Before accepting persistent updates, the system should compare candidates against the incumbent on both target tasks and representative prior tasks. It should track not only mean score but variance, worst-case performance, and failure categories. If a candidate improves one niche but regresses another, population storage may be preferable to replacement. The update should be conditional: deploy the candidate only for the niche where it is proven better.

The sixth failure mode is *evaluator drift*. Evaluators are themselves artifacts: tests change, benchmarks are updated, LLM judges are replaced, metrics are reweighted, and deployment environments evolve. If evaluator versions are not recorded, historical scores become incomparable. A candidate may appear to improve because the evaluator became easier. A population archive may preserve elites selected under obsolete criteria. Reflection memories may cite feedback from an evaluator that no longer exists.

Evaluator safeguards include versioning, raw trace storage, re-evaluation of elites, and clear separation between evaluator development and candidate selection. When an evaluator changes, the system should mark old scores as belonging to an older regime. Important elites should be re-run under the new evaluator. If raw traces are stored, some metrics can be recomputed without re-running expensive tasks. Evaluator updates should themselves be reviewed, especially when they alter safety or acceptance criteria. A self-evolving system should not silently move the goalposts.

The seventh failure mode is *unsafe autonomy escalation*. A system may begin with low-risk output revision and gradually acquire the ability to modify prompts, tools, code, workflows, or model weights. Each added update channel increases the possible impact of mistakes. If governance does not track these channels, the system can become more autonomous than intended. This is particularly concerning for agents with external tools, network access, file access, or deployment permissions.

Safeguards include capability boundaries, permission tiers, human gates, sandboxing, and deployment separation. The system should declare which artifacts it may modify and under what conditions. Code changes should run in isolated environments. Tool policies should not be self-modified without review. Weight updates should be separated from online user interaction unless strong controls exist. Discovery archives should be distinct from production deployments. The more persistent the update, the more conservative the gate.

The eighth failure mode is *explanation laundering*. A system may generate plausible explanations for why a candidate improved, even when the actual cause is accidental. This can happen in reflection loops and in human-facing reports. If reviewers accept explanations without checking evidence, the system's narrative becomes more trusted than its measurements. Explanation laundering is a subtle risk because self-evolving systems are often designed to produce logs, reflections, and rationales. These artifacts are useful, but they are not proof.

Safeguards include evidence-first reporting, links from claims to evaluator outputs, and distinction between measured facts and inferred explanations. A reflection should say which test failed or which metric changed. A candidate summary should distinguish "passed 47/50 tests" from "likely improved because of better planning." Lineage reports should include actual diffs or structural

changes, not only prose. Human reviewers should be able to audit the evidence trail.

Governance across loops can be summarized by four rules. First, make the update substrate explicit. Second, match evaluator strength to update depth. Third, version every artifact that influences selection. Fourth, preserve the ability to roll back. These rules do not prevent all failures, but they make failures observable and recoverable. Self-evolution without observability is drift; self-evolution with evidence, lineage, and rollback becomes an engineering discipline.

2.12 Loop Composition Patterns

The five-loop taxonomy can be used as a design language. Instead of asking whether a system is self-evolving in the abstract, we can specify a loop composition pattern. Different applications require different patterns. A low-risk writing assistant may need only reflection and lightweight evaluation. A code optimization system needs search, executable evaluation, and rollback. An AutoML product needs specification-to-execution, search, and evaluator loops. A research discovery engine needs population archives and strong validators. A long-term autonomous agent needs all five loops plus governance.

A first pattern is the *reflection-first assistant*. The system attempts a task, receives feedback, writes a scoped memory, and uses that memory on later attempts. This pattern is appropriate when tasks repeat, updates should be interpretable, and external evaluation is available but not necessarily executable. Reflexion-like agents fit here. The minimal state includes task traces, reflections, and retrieval metadata. The main risk is memory contamination, so the main safeguard is evidence-linked memory and retrieval scope.

A second pattern is the *test-driven code evolver*. The system generates code variants, runs tests, stores failures, and accepts only candidates that improve a scorecard without regressions. Self-Debug, CodeEvolve, OpenEvolve, FunSearch, and AlphaEvolve-like systems occupy this family at different scales. The main state includes code, tests, benchmark traces, candidate lineage, and an incumbent baseline. The generator may use LLM-guided mutation and semantic crossover. The evaluator is decisive: compilers, unit tests, benchmarks, and performance measurements should gate updates. The main risks are reward hacking, hidden regressions, and unsafe code execution, so sandboxing and regression suites are mandatory.

A third pattern is the *AutoML compiler*. The system converts user goals into data-processing and model-training pipelines. It searches over components and hyperparameters, evaluates validation metrics, and reflects on data or modeling failures. AutoML-GPT and AutoML-Agent illustrate this pattern. The main state includes task cards, data profiles, pipeline configurations, training logs, and model comparison tables. The main risks are specification drift, data leakage, and metric overfitting. Safeguards include explicit assumptions, leakage checks, held-out validation, and human review for deployment.

A fourth pattern is the *agent architecture search engine*. The system treats agent workflows as candidates. It varies role definitions, planning strategies, tool access, memory policies, and verification steps. ADAS and Agent Symbolic Learning are central references. The state includes agent graphs, prompts, tool schemas, component-level feedback, and benchmark results. The evaluator must run full tasks, making cost a major constraint. Cascaded evaluation is useful: cheap static checks, small task suites, then expensive benchmark runs. The main risks are invalid workflows, coordination failures, and overfitting to benchmark task formats.

A fifth pattern is the *quality-diversity archive*. The system maintains elites across task niches or behavior descriptors. MAP-Elites, AlphaEvolve-style archives, DGM, and LLaMEA illustrate this pattern. It is appropriate when there is no single universally best candidate or when stepping stones matter. The state includes an archive grid, descriptors, candidate artifacts, parent links,

and evaluator versions. The main benefits are diversity and specialization. The main risks are population bloat and unsafe archive contents. Safeguards include descriptors, pruning, quarantine states, and best-per-niche regression checks.

A sixth pattern is the *self-play trainer*. The system generates tasks, solutions, critiques, or opponent behaviors, verifies them, and uses the resulting data to update model parameters. SPIN and Absolute Zero-style systems fit here. This pattern is powerful because it can change the base model rather than only prompts or code. It is also difficult to govern because weight updates are broad and less interpretable. The main state includes generated curricula, verification results, training data, model checkpoints, and evaluation suites. The main risks are degenerate curricula, overfitting, and broad capability regressions. Safeguards include held-out benchmarks, task diversity metrics, checkpoint comparison, and conservative deployment gates.

These patterns can be layered. A test-driven code evolver may use a reflection-first assistant to summarize failures. An AutoML compiler may maintain a quality-diversity archive of pipelines for different data regimes. An agent architecture search engine may use self-play-generated tasks to expand its benchmark suite. A self-play trainer may use programmatic evaluators from the code-evolution pattern to verify generated tasks. The taxonomy encourages such modular composition while keeping responsibilities clear.

A practical design process can begin with five questions. What artifact should evolve? What candidates can be generated safely? What evaluator can be trusted? What update is allowed without human review? What history should be retained? The answers determine the loop composition. If the artifact is a final response and the evaluator is a user comment, reflection may suffice. If the artifact is source code and the evaluator is a test suite, search plus evaluator is appropriate. If the artifact is an agent organization and the evaluator is a task benchmark, architecture search and population archives may be needed. If the artifact is model weights, training governance is required.

Loop composition also provides a reporting standard for papers. A self-evolution paper should report the candidate representation, generator prompts or policies, evaluator suite, selection rule, update substrate, memory mechanism, population size or archive policy, compute budget, and regression checks. Many current papers emphasize final performance but underreport lineage and evaluator details. As the field matures, reproducibility will require more complete loop descriptions. Without them, it is hard to know whether a result came from a stronger generator, a larger search budget, a better evaluator, or a favorable benchmark.

The same reporting standard helps product builders. A self-evolving product should expose user-facing explanations at the right abstraction level. Users do not need every candidate trace, but they need to know when the system changed, what evidence justified the change, and how to recover if the change is harmful. Developers need deeper logs: prompts, diffs, tests, scores, and lineage. Governance teams need policy status, risk flags, evaluator versions, and deployment gates. The five-loop taxonomy maps naturally to these views.

Finally, loop composition clarifies a central research frontier: meta-evolution. Once systems can evolve prompts, code, memories, and populations, they can also evolve the generators, evaluators, and selection rules themselves. LLaMEA already points toward evolving optimizers. Agent Symbolic Learning points toward updating symbolic components. Future systems may search over evaluator designs, reflection schemas, archive descriptors, and governance policies. This is powerful but risky because it changes the rules of evolution. Meta-evolution should be slower, more audited, and more conservative than ordinary candidate evolution.

2.13 A Unifying Example: From User Goal to Evolved Agent

Consider a user who asks for an agent that can maintain a small software library. The user wants the agent to triage issues, write patches, run tests, update documentation, and avoid breaking public APIs. A non-evolving agent might be hand-designed once and deployed. A self-evolving system treats the request as the start of a sequence of loops.

The specification-to-execution loop first converts the request into a task card. The task card states the repository type, allowed files, testing commands, API stability constraints, documentation requirements, and success metrics. It identifies missing information, such as supported language versions or deployment targets. It builds an initial agent workflow: issue reader, planner, coder, test runner, reviewer, and documentation updater. This workflow is executable because each role has prompts, tool permissions, and handoff rules.

The search loop then proposes variants. One variant adds a static-analysis step before coding. Another adds a regression-test generator. Another changes the planning prompt to require API-impact analysis. Another splits the reviewer into correctness and style reviewers. Another changes the order of documentation updates. Each candidate is an agent workflow, not just an answer. The generator uses prior failures, examples from similar repositories, and constraints from the task card.

The evaluator loop runs candidates on a suite of repository-maintenance tasks. It measures tests passed, patches applied, API compatibility, documentation completeness, time, and tool-call cost. It records traces: failed commands, changed files, review comments, and final diffs. It also includes negative checks, such as whether the agent modified files outside scope or ignored a failing test. A candidate that solves more issues but frequently changes public APIs is not accepted as a global replacement.

The reflection loop summarizes failures. If the agent repeatedly forgets to update documentation after changing behavior, a memory is written for the documentation updater. If the coder frequently breaks public APIs, a reflection is attached to the planning and review components: public interfaces must be inspected before edits and regression tests must include downstream examples. If a static-analysis tool produces noisy warnings, the memory records when to trust it and when to ignore it. These memories are scoped to software-maintenance tasks and tied to evaluator evidence.

The population loop preserves specialized workflows. A low-latency workflow handles simple documentation fixes. A high-assurance workflow handles API changes. A dependency-update workflow handles package upgrades. A bug-fix workflow handles failing tests. MAP-Elites descriptors might include task type, risk level, latency budget, and required tool depth. The system does not force one workflow to handle every issue. Instead, it routes future tasks to the best elite for the niche.

As the system operates, cross-loop interactions become visible. A new class of issue appears, such as security vulnerability reports. The specification loop updates the task card schema to include severity and disclosure constraints. The search loop proposes security-specific reviewers. The evaluator loop adds vulnerability regression tests. The reflection loop stores lessons about unsafe patch patterns. The population loop creates a security-maintenance niche. If the system later trains a model or adapter on successful traces, the training loop must be evaluated against the frozen baseline to ensure that general coding ability did not regress.

This example shows why self-evolution is not merely "the model improves itself." The model may remain fixed while the system improves dramatically through better specifications, workflows, tests, memories, and archives. Conversely, a model may be fine-tuned without meaningful self-evolution if there is no closed loop from evaluated experience to targeted update. The essence is the structured transition from state to candidate to feedback to selected update.

It also shows why safety is inseparable from the taxonomy. If the agent can edit code, the evaluator must run tests and enforce file boundaries. If the agent can update its workflow, workflow changes need review. If it can store memories, memories need scope. If it can maintain populations, archive candidates need deployment status. If it can train weights, checkpoints need broad evaluation. Each loop adds capability and therefore requires a corresponding guardrail.

The same pattern applies beyond software. In scientific discovery, the specification is a research objective, search proposes hypotheses or programs, evaluators run simulations or proof checks, reflection stores failed hypotheses, and population archives preserve diverse approaches. In education, the specification is a learning goal, search proposes tutoring strategies, evaluators measure student outcomes, reflection stores pedagogical lessons, and population methods preserve strategies for learner types. In robotics, the specification is a task, search proposes policies or controllers, evaluators use simulators and real-world tests, reflection stores failure modes, and populations preserve behaviors for environments. The substrate changes, but the loops remain.

2.14 Summary

This chapter organized AI self-evolution around five loops. The specification-to-execution loop turns natural-language goals into executable artifacts, as illustrated by AutoML-GPT, AutoML-Agent, and ADAS. The search loop explores prompts, architectures, code, workflows, and hyperparameters, as illustrated by OPRO, EvoPrompting, ADAS, OpenEvolve, and AlphaEvolve. The evaluator loop supplies fitness through tests, benchmarks, validators, execution traces, and deployment metrics, as illustrated by FunSearch, Self-Debug, CodeEvolve, and SelfEvolve. The reflection loop turns failures into reusable memory and feedback, as illustrated by Reflexion, Self-Refine, STaR, and Agent Symbolic Learning. The population loop maintains multiple candidates, lineages, and niches, as illustrated by AlphaEvolve, LLaMEA, DGM, SPIN, and Absolute Zero.

The unified formalization $z_{t+1} = u(z_t, g(z_t), e(g(z_t)))$ is intentionally broad. Its value is that it makes hidden design choices explicit. What is the state? What candidates can be generated? What evaluator is trusted? How are candidates selected? What is updated? How persistent is the update? How is rollback handled? These questions cut across AutoML, NAS, program synthesis, agent frameworks, evolutionary computation, and LLM self-improvement.

The main thesis is that self-evolution is a systems property. It does not arise from generation alone, nor from reflection alone, nor from benchmarking alone. It arises when generation, evaluation, memory, selection, and update are connected in a persistent loop. The strength of the system depends on the alignment between evaluator quality and update power. Weak evaluators can support lightweight revisions; strong executable evaluators can support code and population evolution; broad validated feedback can support training-time updates.

The taxonomy also clarifies why the field is converging. AutoML contributes executable pipeline construction. NAS contributes architecture search and benchmark methodology. Evolutionary computation contributes population, diversity, and selection. Program synthesis contributes test-driven generation and repair. Agent research contributes tool use, memory, and multi-agent organization. LLM self-improvement contributes reflection, synthetic data, and self-play. The next generation of systems will likely combine all five loops into platforms that can receive specifications, generate artifacts, evaluate them, remember failures, maintain candidate populations, and update themselves under governance.

For survey purposes, the five-loop taxonomy provides a map. For system builders, it provides a checklist. A self-evolving AI system should specify its state, candidates, evaluator, selection policy, update substrate, memory, population strategy, and safety gates. Methods can then be compared not only by final score but by how they evolve, how safely they update, and how much of their

improvement is reproducible. This is the foundation for treating AI self-evolution as a coherent research and engineering discipline rather than a loose collection of prompting tricks, AutoML tools, and agent demos.

3 Self-Improvement Methods

This chapter surveys the algorithmic core of AI self-evolution: the methods by which a model or agent uses its own generations, traces, critiques, executions, preferences, memories, or environment interactions to become more capable. The term "self-improvement" is used in several incompatible ways in the literature. In the narrowest sense, it can mean a prompt-only inference routine in which an LLM writes an answer, critiques the answer, and rewrites it. In a stronger sense, it can mean an agent that records failures in a memory buffer and conditions future actions on those records. In an even stronger sense, it can mean a training procedure in which the model generates new data, filters or verifies that data, and updates its parameters with supervised learning, preference optimization, or reinforcement learning. Finally, in open-ended agent systems it can mean architecture search, code self-modification, or population-based evolution. This chapter focuses on the first three layers—inference-time loops, training-time loops, and their benchmark consequences—because these layers are the reusable methodological substrate for the broader self-evolving systems discussed elsewhere in this survey.

A useful unifying abstraction is to view each method as instantiating the same feedback transformation,

$$\theta_{t+1}, m_{t+1}, y_{t+1} = \mathcal{U}\big(\theta_t, m_t, x, y_t, \mathcal{F}(x, y_t, \tau_t)\big), \tag{22}$$

where x is the task input, y_t is the current output, τ_t is the execution or reasoning trajectory, m_t is non-parametric state such as memory or retrieved demonstrations, θ_t denotes model parameters, $\mathcal{F}$ is a feedback source, and $\mathcal{U}$ is an update rule. In inference-time self-improvement, $\theta_{t+1} = \theta_t$ and the update occurs in y or m: the model rewrites an answer, adds reflection text to context, or changes the next action plan. In training-time self-improvement, θ changes: the feedback is converted into new supervised data, preference pairs, filtered trajectories, rewards, or policy gradients. In agent-level self-evolution, θ may remain fixed while the executable system around the model changes, for example through prompts, tools, workflows, or code patches. The distinction is not merely taxonomic; it determines the cost, safety profile, reproducibility, and failure modes of the method.

The central design question is therefore not whether an AI system can "criticize itself." Modern LLMs can often produce plausible criticism on demand. The harder question is whether the criticism is grounded in a signal that is more reliable than the original generation. Self-Refine [36] grounds feedback in the same model's latent knowledge and task rubric. Reflexion [49] grounds feedback in episode-level reward and stores it as natural-language memory. Self-Debug [10] grounds feedback in code execution. SPIN [11] grounds improvement in a contrast between human demonstrations and the model's own previous responses. STaR [75] grounds rationale learning in final-answer correctness. ReST-EM grounds improvement in reward-model filtering and expectation–maximization style iteration. Constitutional AI [5] grounds harmlessness feedback in a human-written constitution and AI-generated preferences. RISE [44], RAGEN [63], and ReVeal [27] push the loop into reinforcement learning over multi-turn trajectories, where credit assignment, exploration, and reward hacking become first-class problems. The methodological progression is a shift from ungrounded self-talk to verifiable or preference-grounded optimization.

We organize the chapter into four parts. Section 3.1 analyzes inference-time self-improvement, emphasizing Self-Refine [36], Reflexion [49], and Self-Debug [10]. Section 3.2 analyzes training-time

methods, including SPIN, STaR, ReST-EM, Constitutional AI, RISE, RAGEN, and related verifier-based code agents. Section 3.3 provides a compact method comparison. Section 3.4 consolidates benchmark numbers from the research corpus and the primary papers, with special attention to HumanEval [9], GSM8K [14], BBH, MMLU, and SWE-Bench [26]. Throughout, we treat self-improvement as a loop design problem: What state is updated? What feedback is trusted? What prevents drift? What is the unit of selection? How is improvement measured?

3.1 Inference-Time Self-Improvement

Inference-time self-improvement is the lightest and most immediately deployable form of AI self-evolution. The base model is not retrained. Instead, the system spends additional computation at test time to produce, inspect, and revise its own outputs. This makes the method attractive for settings where model weights are inaccessible, deployment must remain stateless, or iteration speed matters more than amortized training cost. It also makes the method fragile: because the same underlying distribution produces both the candidate answer and the critique, feedback can be circular, overconfident, or stylistically persuasive without being correct. The practical success of inference-time self-improvement therefore depends on the introduction of asymmetry. The generator and critic may be the same model, but they should see different prompts, rubrics, examples, tools, memories, or execution results. Without such asymmetry, the loop can become a paraphrase engine rather than an optimizer.

The general inference-time loop can be written as

$$y_0 \sim p_\theta(\cdot \mid x, p_{\text{gen}}), \qquad c_t \sim p_\theta(\cdot \mid x, y_t, p_{\text{crit}}), \qquad y_{t+1} \sim p_\theta(\cdot \mid x, y_t, c_t, p_{\text{rev}}), \tag{23}$$

where p_{gen}, p_{crit}, and p_{rev} are generation, critique, and revision prompts. A stopping rule $S(x, y_t, c_t)$ terminates the loop when the critique reports no actionable issue, a verifier accepts the output, or a budget is exhausted. The important point is that the state update is external to the model: it is a change in the prompt context, output draft, tool trace, or memory. The method is cheap to implement but expensive at inference: T refinement rounds multiply latency and token usage by roughly $T + 1$ or $2T + 1$, depending on whether critique and revision are separate calls.

3.1.1 Self-Refine: iterative generation, critique, and refinement

Self-Refine, introduced by Madaan et al. [36], is the canonical single-model inference-time refinement loop. The same LLM plays three roles: it generates an initial answer, gives feedback on that answer, and revises the answer according to the feedback. The method requires no additional training data, no reinforcement learning, and no parameter update. Its appeal is that it converts a black-box LLM into an iterative optimizer over its own textual outputs. The core recurrence is

$$x_{t+1} = \text{Refine}\big(x_t, \text{Critique}(x_t)\big), \tag{24}$$

where x_t denotes the current draft. To make the dependence on task input explicit, we can write

$$y_0 = G_\theta(x; \rho_G), \tag{25}$$
$$f_t = C_\theta(x, y_t; \rho_C), \tag{26}$$
$$y_{t+1} = R_\theta(x, y_t, f_t; \rho_R), \tag{27}$$

where G, C, and R are prompted uses of the same model and ρ_G, ρ_C, ρ_R are role instructions. A typical stopping condition is

$$\text{stop}(t) = \mathbf{1}\{t \geq T_{\max} \ \vee \ f_t \in \mathcal{F}_{\text{no-change}} \ \vee \ V(x, y_t) = 1\}, \tag{28}$$

Figure 8: Self-Refine inference loop

Input: Input x, model M_θ, maximum iterations T, optional verifier V

1. $y_0 \leftarrow M_\theta(\textsc{GeneratePrompt}(x))$

for $t = 0$ to $T - 1$ **do**

2. $f_t \leftarrow M_\theta(\textsc{CritiquePrompt}(x, y_t))$

if $\textsc{NoActionableIssue}(f_t)$ or $V(x, y_t) = 1$ **then**

3. **return** y_t

4. $y_{t+1} \leftarrow M_\theta(\textsc{RefinePrompt}(x, y_t, f_t))$

5. **return** y_T

where V is an optional verifier. Without a verifier, the loop stops when the model declares that no further improvements are needed; with a verifier, it stops when the answer passes a task-specific test.

Self-Refine is best understood as a form of textual coordinate descent. The current answer is a point in a very high-dimensional discrete space. The critique names a direction of improvement, such as "the proof skipped the induction hypothesis," "the code does not handle empty input," or "the response violates the requested tone." The revision moves the answer in that direction. Unlike gradient descent, however, the direction is not computed from a differentiable loss. It is a natural-language hypothesis about what would reduce the loss. The method works when the model's critic mode has access to error information that the generator mode did not use effectively. It fails when the critic merely rationalizes the answer or when the task requires knowledge absent from the model.

A minimal Self-Refine implementation is shown in Algorithm 8. In practice, strong implementations separate the critique into dimensions. For code, the critique may inspect correctness, complexity, edge cases, and style. For writing, it may inspect factuality, organization, audience fit, and redundancy. For mathematical reasoning, it may inspect equation validity and whether each conclusion follows from previous steps. This decomposition matters because generic prompts such as "improve the answer" often lead to cosmetic edits, while dimension-specific prompts create actionable feedback.

The research corpus reports that Self-Refine achieved an average improvement of roughly 20% across seven diverse tasks. The local summary records exact examples: math reasoning accuracy improved from 56% to 67%; sentiment reversal improved from 74% to 86%; dialogue response win rate improved from 52% to 68%; constrained generation compliance improved from 65% to 78%; code readability improved from 3.5 to 4.1; acrostic poetry quality improved from 3.2 to 3.8; and code optimization improved by 28%. These results show the breadth of the loop: it is not a code-only or math-only technique. The same design pattern can improve reasoning, style, constraint satisfaction, and interaction quality, provided the critique prompt can expose the relevant failure mode.

The strongest use case for Self-Refine is local repair under a known rubric. If an answer must satisfy explicit constraints, such as "include exactly three bullet points," "avoid first-person lan-

guage," or "return valid JSON," the model can often identify and correct violations. The method is less reliable for hidden constraints and factual correctness. If the draft contains a subtle false statement and the model's internal knowledge supports the same false statement, self-critique will not fix it. In such settings, Self-Refine should be coupled to retrieval, execution, external judges, or human review.

Self-Refine also illustrates an important principle of self-evolution: a loop is not automatically beneficial. Additional iterations can degrade outputs by overfitting to the critic's preferences. In creative writing, repeated refinement may remove distinctive phrasing. In code, repeated refactoring may introduce unnecessary complexity. In reasoning, the model may "repair" a correct but concise answer into a verbose and flawed one. Therefore a robust Self-Refine system should log intermediate drafts, retain the best verified candidate rather than the last candidate, and use ablations to determine the optimal number of rounds.

3.1.2 Reflexion: verbal reinforcement learning through memory

Reflexion, introduced by Shinn et al. [49], is another inference-time method, but it changes the unit of improvement. Self-Refine improves a single output draft. Reflexion improves an agent across episodes by storing linguistic feedback in memory. It is therefore closer to reinforcement learning in structure, but the update is non-parametric: the model weights remain fixed, and the "learning" occurs through a growing context of natural-language reflections. The local research summary describes Reflexion as using three specialized models: an Actor M_a, an Evaluator M_e, and a Self-Reflection model M_r. These may be the same base LLM under different prompts or separate models.

The full Reflexion loop can be written as

$$a_e \sim M_a(\cdot \mid x, M_{e-1}), \tag{29}$$

$$r_e = M_e(x, a_e, \tau_e), \tag{30}$$

$$f_e \sim M_r(\cdot \mid x, a_e, \tau_e, r_e), \tag{31}$$

$$M_e = \mathrm{UpdateMemory}(M_{e-1}, f_e), \tag{32}$$

$$a_{e+1} \sim M_a(\cdot \mid x, M_e), \tag{33}$$

where e indexes episodes, a_e is the action or answer, τ_e is the trajectory, r_e is a scalar or binary evaluation, f_e is a verbal reflection, and M_e is the memory buffer after episode e. If one distinguishes successful and failed episodes, the update can be expressed as

$$M_e = M_{e-1} \cup \begin{cases} \{M_r(x, a_e, \tau_e, r_e)\}, & r_e < \kappa, \\ \varnothing, & r_e \geq \kappa, \end{cases} \tag{34}$$

with threshold κ. In interactive tasks, τ_e may contain observations and actions; in code generation, it may contain compiler errors and failed tests; in question answering, it may contain the proposed reasoning chain and final answer.

Algorithm 9 states the loop. The key conceptual move is that f_e acts like a verbal gradient. A numerical gradient says how to change parameters to reduce loss; a Reflexion memory says how to change behavior to avoid a repeated failure. For example, after a failed WebShop episode, the memory might say that the agent clicked on a superficially relevant item without checking size constraints. In the next episode, the actor reads this memory and attends to size constraints earlier. In code generation, the reflection might say that the previous solution forgot to import a library or mishandled zero-length arrays.

Figure 9: Reflexion with language memory

Input: Task x, actor M_a, evaluator M_e, reflector M_r, memory budget B, episode budget E

1. $\mathcal{M}_0 \leftarrow \varnothing$

for $e = 1$ to E **do**

2. $a_e, \tau_e \leftarrow M_a(x, \mathcal{M}_{e-1})$
3. $r_e \leftarrow M_e(x, a_e, \tau_e)$

if $\textsc{Success}(r_e)$ **then**

4. **return** a_e

else

5. $f_e \leftarrow M_r(x, a_e, \tau_e, r_e)$
6. $\mathcal{M}_e \leftarrow \textsc{CompressAndAppend}(\mathcal{M}_{e-1}, f_e, B)$
7. **return** best action according to M_e

The benchmark signature of Reflexion is unusually strong for a non-parametric method. The local research corpus reports 91.0% pass@1 on HumanEval [9] using Reflexion with GPT-3.5, compared with 80.1% for GPT-4 zero-shot and 67.0% for fine-tuned Codex. It also reports 97% success on ALFWorld [50] compared with a 75% base, and improvements on MBPP [4], APPS, and WebShop. These numbers should be interpreted carefully. Reflexion uses repeated attempts and evaluation signals, so it is not the same compute regime as one-shot generation. Nevertheless, the results demonstrate that language memory can substitute for parameter updates in tasks where failure signals are available and where future attempts can condition on previous mistakes.

Reflexion has several design degrees of freedom. First, the evaluator may be a programmatic verifier, a task environment, a reward model, or another LLM. Programmatic evaluators are preferred when available because they reduce self-confirmation bias. Second, the memory may be append-only, summarized, retrieved by similarity, or prioritized by recency and severity. A naive append-only buffer eventually exceeds context length and introduces stale advice. Third, the reflection prompt may ask for failure analysis, a future plan, or a general rule. General rules transfer better across tasks, but concrete failure analyses are more faithful to the episode. Fourth, the actor may see all reflections or only a retrieved subset. Retrieval reduces context cost but can hide important lessons.

Reflexion's limitations reveal why inference-time self-improvement alone is not enough for robust self-evolution. The method requires a reliable evaluation signal. If the evaluator is wrong, the memory stores misleading advice. Memory can also accumulate contradictions: one reflection says to be concise, another says to expand details, and the actor must infer which applies. Long-running agents may suffer from memory poisoning, where a single erroneous reflection affects many later actions. Finally, the method has no convergence guarantee. A sequence of reflections may improve behavior on a benchmark while merely teaching the agent benchmark-specific heuristics. For these reasons, Reflexion is best viewed as a powerful test-time adaptation mechanism, not as a complete

Figure 10: Execution-grounded Self-Debug

Input: Programming task x, model M_θ, tests or examples $\mathcal{T}$, budget T

1. $c_0 \leftarrow M_\theta(\textsc{CodePrompt}(x))$

for $t = 0$ to $T - 1$ **do**

2. $o_t \leftarrow \textsc{RunSandbox}(c_t, \mathcal{T})$

if $\textsc{Pass}(o_t)$ **then**

3. **return** c_t

4. $d_t \leftarrow M_\theta(\textsc{DebugPrompt}(x, c_t, o_t))$

5. $c_{t+1} \leftarrow M_\theta(\textsc{RepairPrompt}(x, c_t, o_t, d_t))$

6. **return** best verified candidate

replacement for training.

3.1.3 Self-Debug and execution-grounded repair

Self-Debug, associated with Chen et al. [10] and related code self-repair work, introduces the most important kind of asymmetry for code tasks: execution feedback. A program can be run. It can fail to parse, throw an exception, violate an assertion, time out, or produce output that differs from an expected value. These signals are often more reliable than an LLM's introspective critique. The model still performs the repair, but the critique is anchored in the external semantics of the programming language and test environment.

A generic Self-Debug loop is

$$c_0 \sim p_\theta(\cdot \mid x), \tag{35}$$
$$o_t = \text{Execute}(c_t, \mathcal{T}_x), \tag{36}$$
$$d_t \sim p_\theta(\cdot \mid x, c_t, o_t), \tag{37}$$
$$c_{t+1} \sim p_\theta(\cdot \mid x, c_t, o_t, d_t), \tag{38}$$

where c_t is code, $\mathcal{T}_x$ is a test harness or input-output specification, o_t is execution output, and d_t is a natural-language debugging explanation. If tests pass, the loop terminates. If tests fail, the error trace becomes a repair signal. In many implementations, d_t is optional: the model can directly revise code from the error output. However, an explicit debugging explanation can improve repair because it forces the model to localize the bug before editing.

Execution-grounded repair is the bridge between prompt-level self-improvement and verifier-based training. At inference time, the execution trace is used only to revise the current solution. At training time, the same traces can be stored as trajectories for supervised repair, preference optimization, or reinforcement learning. SelfEvolve [25], for example, uses self-generated knowledge and sandbox execution in a two-stage pipeline. The local corpus reports that SelfEvolve improved DS-1000 [29] pass@1 from 49.4 for a ChatGPT baseline to 57.2 for the full method, HumanEval [9] pass@1 from 74.39 for ChatGPT to 85.98, and TransCoder accuracy from 88.3 to 90.4. These

results are not identical to Self-Debug, but they illustrate the same principle: execution converts self-improvement from subjective critique into grounded repair.

The main advantage of Self-Debug is precision. A traceback points to a line and an exception. A failed assertion supplies an input where behavior is wrong. A compiler error identifies a syntactic or type-level violation. This makes the feedback more actionable than a broad critique. The main limitation is coverage. Passing visible tests does not guarantee correctness on hidden tests. The model may overfit to the supplied examples or patch symptoms rather than causes. A robust system therefore needs test generation, fuzzing, property checks, or hidden evaluation. Modern code self-evolution methods such as ReVeal [27] make this explicit by co-evolving code generation and test generation.

Self-Debug also raises security and isolation concerns. Running model-generated code is dangerous without sandboxing, resource limits, and filesystem isolation. A self-improving code agent should never execute arbitrary code with unrestricted permissions. It should constrain CPU, memory, network access, wall-clock time, file writes, and system calls. The same sandbox that produces feedback is also a safety boundary. In self-evolution systems, the evaluator is part of the threat model.

3.1.4 Inference-time design patterns

Across Self-Refine, Reflexion, and Self-Debug, several design patterns recur. First is role separation. Even if the same base model is used, prompts should create distinct generator, critic, verifier, and editor roles. Second is feedback grounding. Feedback should be connected to a rubric, environment reward, execution trace, retrieval source, or formal constraint. Third is state management. The loop must decide what to preserve: the latest draft, the best verified draft, all critiques, compressed reflections, or retrieved memories. Fourth is stopping. More iterations are not always better; stopping rules should be validated empirically. Fifth is auditability. Because the model may revise its own work, logs of drafts, critiques, tests, and decisions are essential for debugging and safety review.

A practical inference-time self-improvement system often combines the three methods. For example, a code agent may generate a solution, run tests, ask the model to explain the error, repair the code, and store a reflection about the root cause for future problems. The combined state update is

$$(c_{t+1}, \mathcal{M}_{t+1}) = \Big(R_\theta(x, c_t, o_t, d_t),\ \mathrm{Append}\big(\mathcal{M}_t, M_r(x, c_t, o_t)\big) \Big). \tag{39}$$

This hybrid approach is more powerful than pure Self-Refine because it uses external feedback, and more persistent than pure Self-Debug because lessons can transfer across episodes.

The central risk is self-confirmation. A model can generate an answer, generate a critique that agrees with its assumptions, and generate a revision that looks more polished while remaining wrong. External feedback reduces this risk but does not eliminate it. Tests can be incomplete, reward models can be hacked, and retrieved documents can be misread. Therefore inference-time self-improvement should be evaluated with held-out tasks, adversarial cases, and ablations that compare one-shot generation, critique-only, refinement-only, external-verifier-only, and full-loop variants. Claims of self-improvement are meaningful only when the loop beats strong compute-matched baselines.

3.2 Training-Time Self-Improvement

Training-time self-improvement converts feedback into parameter updates. This is a stronger and riskier form of self-evolution. It can amortize the cost of discovery: once a model learns from

synthetic rationales, filtered samples, self-play comparisons, or reinforcement trajectories, future inference can be cheaper. It can also create new capabilities that are difficult to elicit by prompting alone. However, it introduces familiar training risks: distribution shift, reward hacking, catastrophic forgetting, mode collapse, data contamination, and benchmark overfitting. The design problem shifts from "How should the model revise this answer?" to "Which self-generated data should be trusted enough to update the model?"

A generic training-time self-improvement loop is

$$\mathcal{D}_t = \text{Generate}(\pi_{\theta_t}, \mathcal{X}_t), \tag{40}$$

$$\tilde{\mathcal{D}}_t = \text{FilterOrLabel}(\mathcal{D}_t, R, V, C), \tag{41}$$

$$\theta_{t+1} = \text{Train}(\theta_t, \tilde{\mathcal{D}}_t, \mathcal{L}), \tag{42}$$

where $\mathcal{X}_t$ is a prompt/task distribution, R is a reward model or environment reward, V is a verifier, C is a constitution or criterion set, and $\mathcal{L}$ is a supervised, preference, or RL objective. The loop becomes self-improving when $\mathcal{D}_t$ is at least partly produced by the current model and θ_{t+1} is used to produce better data in the next round.

3.2.1 SPIN: self-play fine-tuning from weak to strong

Self-Play Fine-Tuning (SPIN) addresses a central question in self-evolution: can a weak supervised model become stronger without additional human labels? The method starts with an SFT model p_{θ_0} trained on high-quality prompt-response pairs $(x, y) \sim p_{\text{data}}$. At iteration t, the current model p_{θ_t} generates synthetic responses $y' \sim p_{\theta_t}(\cdot \mid x)$ for the same prompts. The next model is trained to distinguish human data from its own previous outputs and, in doing so, to move its distribution closer to the data distribution. The "self-play" is not a game against another agent; it is a game between the current model and its previous iteration.

The SPIN objective can be written as

$$\mathcal{L}_{\text{SPIN}}(\theta, \theta_t) = \mathbb{E}_{x\sim q,\, y\sim p_{\text{data}},\, y'\sim p_{\theta_t}} \left[\ell \left(\lambda \log \frac{p_\theta(y \mid x)}{p_{\theta_t}(y \mid x)} - \lambda \log \frac{p_\theta(y' \mid x)}{p_{\theta_t}(y' \mid x)} \right) \right], \tag{43}$$

where ℓ is a monotonically decreasing convex loss, often the logistic loss, and λ controls the scale. The update is

$$\theta_{t+1} = \arg\min_{\theta\in\Theta} \mathcal{L}_{\text{SPIN}}(\theta, \theta_t). \tag{44}$$

Intuitively, the model is rewarded for increasing the likelihood ratio of human responses relative to its previous policy and decreasing the likelihood ratio of its own synthetic responses when they differ from the human data.

The user requested the DPO loss formula because SPIN is often discussed next to DPO. Direct Preference Optimization assumes preference triples (x, y_w, y_l) and optimizes

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[\log \sigma \left(\beta \left(\log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right) \right) \right]. \tag{45}$$

SPIN resembles DPO when the loss is logistic and the "winner" is the human response while the "loser" is the model's previous response. The difference is procedural and conceptual. DPO uses a preference dataset and can be run as a one-shot preference optimization method. SPIN constructs its own rejected responses from the previous model and naturally iterates. SPIN therefore creates a self-play curriculum: as the model improves, its synthetic negative examples become harder to distinguish from human data.

Figure 11: Self-Play Fine-Tuning (SPIN)

Input: SFT dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, initial model p_{θ_0}, iterations T

for $t = 0$ to $T - 1$ **do**

for $i = 1$ to N **do**

1. $y_i' \sim p_{\theta_t}(\cdot \mid x_i)$
2. $\theta_{t+1} \leftarrow \arg\min_\theta \sum_i \ell \left(\lambda \log \frac{p_\theta(y_i|x_i)}{p_{\theta_t}(y_i|x_i)} - \lambda \log \frac{p_\theta(y_i'|x_i)}{p_{\theta_t}(y_i'|x_i)} \right)$
3. **return** θ_T

Algorithm 11 summarizes the procedure. The loop is simple, but it depends on the presence of high-quality initial SFT data. SPIN is not zero-data self-improvement. It is better described as self-play amplification of a supervised seed. The seed gives the system a target distribution; self-play supplies increasingly difficult contrasts.

The primary SPIN paper reports improvements on the HuggingFace Open LLM Leaderboard using Zephyr-7B-SFT-full as the base. The base model scored 26.76 on GSM8K [14] and 60.92 on MMLU, with an average of 58.14 across ARC, TruthfulQA, Winogrande, GSM8K, HellaSwag, and MMLU. After SPIN iteration 3, GSM8K rose to 38.97, while MMLU was 59.99 and the average was 63.16. Applying DPO after SPIN [11] iteration 3 further increased the average to 64.05, with TruthfulQA rising from 54.90 to 60.07. On a BBH subset, SPIN improved causal judgment from 56.15 to 59.36 by iteration 2 and formal fallacies from 49.6 to 51.2, while sports understanding decreased from 96.0 to 94.4. These numbers show both the promise and the unevenness of self-play: average capability improves, but not every task improves monotonically.

SPIN's key strength is that it turns the model's own weaknesses into training negatives. Its key weakness is that the positive side remains anchored to existing human data. If the SFT data is narrow, the self-play loop may improve imitation of that data without broadening competence. If the generated negatives are low quality, the discrimination problem is too easy and the model learns little. If the generated negatives become high quality, the optimization signal becomes subtle and may require careful regularization. SPIN [11] therefore occupies a middle position between supervised fine-tuning and fully autonomous self-play: it reduces the need for new labels but does not remove the need for an initial high-quality corpus.

3.2.2 STaR: Self-Taught Reasoner and rationale bootstrapping

STaR, the Self-Taught Reasoner of Zelikman et al. [75], is a rationale bootstrapping method. Its starting observation is that chain-of-thought rationales improve reasoning, but human-written rationales are expensive. If a model can generate rationales, and if the final answer can be checked, then correct self-generated rationales can be used as training data. Over iterations, the model becomes better at producing rationales that lead to correct answers. The loop is simple and powerful: generate rationales, keep those that yield correct answers, optionally rationalize failures by conditioning on the correct answer, fine-tune, and repeat.

Figure 12: STaR rationale bootstrapping

Input: Answer-labeled dataset $\mathcal{D} = \{(x_i, a_i)\}$, seed rationale examples $\mathcal{S}$, model M_{θ_0}

for $t = 0$ to $T - 1$ **do**

1. $\mathcal{R}_t \leftarrow \varnothing$

for each $(x_i, a_i) \in \mathcal{D}$ **do**

2. Generate rationale and answer $(r_i, \hat{a}_i) \sim M_{\theta_t}(x_i, \mathcal{S})$

if $\hat{a}_i = a_i$ **then**

3. $\mathcal{R}_t \leftarrow \mathcal{R}_t \cup \{(x_i, r_i, a_i)\}$

else

4. Generate rationalization $r'_i \sim M_{\theta_t}(x_i, a_i, \mathcal{S})$

if AnswerCheck(r'_i, a_i) **then**

5. $\mathcal{R}_t \leftarrow \mathcal{R}_t \cup \{(x_i, r'_i, a_i)\}$

6. Fine-tune M_{θ_t} on $\mathcal{R}_t$ to obtain $M_{\theta_{t+1}}$

7. **return** M_{θ_T}

Let $\mathcal{D} = \{(x_i, a_i)\}$ be problems with final answers but no rationales. At iteration t,

$$r_i^{(t)}, \hat{a}_i^{(t)} \sim p_{\theta_t}(\cdot \mid x_i), \tag{46}$$

$$\mathcal{D}_t^+ = \{(x_i, r_i^{(t)}, a_i) : \hat{a}_i^{(t)} = a_i\}, \tag{47}$$

$$\theta_{t+1} = \arg\max_{\theta} \sum_{(x,r,a) \in \mathcal{D}_t^+} \log p_\theta(r, a \mid x). \tag{48}$$

The rationalization variant augments $\mathcal{D}_t^+$ with rationales generated while conditioning on the correct answer:

$$r_{i,\text{rat}}^{(t)} \sim p_{\theta_t}(\cdot \mid x_i, a_i), \qquad \mathcal{D}_t^{\text{train}} = \mathcal{D}_t^+ \cup \{(x_i, r_{i,\text{rat}}^{(t)}, a_i) : \text{Valid}(r_{i,\text{rat}}^{(t)}, a_i)\}. \tag{49}$$

This step is important when the model rarely solves a problem unaided. Conditioning on the correct answer allows it to search backward for a plausible rationale, increasing the amount of training data. The risk is that rationalized explanations may be post-hoc and unfaithful.

STaR is a foundational self-improvement method because it separates answer supervision from rationale supervision. The dataset supplies final answers; the model supplies the reasoning traces. This is weaker than zero supervision but much cheaper than collecting full chain-of-thought annotations. The method is especially relevant to domains with cheap final-answer verification: arithmetic, multiple-choice commonsense reasoning, symbolic tasks, and some program synthesis problems.

Reported results show modest but meaningful gains for early models. The paper and secondary summaries report GSM8K [14] accuracy rising from 5.8% for an answer-only baseline to about 10.7% with STaR, and CommonsenseQA development accuracy around 72.5%, comparable to a fine-tuned

model roughly 30× larger. These numbers are low by modern standards, but the methodological contribution is durable: self-generated rationales can become training data when filtered by correctness. Later reasoning systems generalize this idea with stronger generators, process reward models, tree search, rejection sampling, and RL with verifiable rewards.

The main limitation of STaR is selection bias. The first iteration keeps rationales for problems the model can already solve, so the training set is biased toward easier examples. Rationalization mitigates this but introduces the risk of explanations that merely justify the known answer. Another limitation is that final-answer correctness is a weak filter for reasoning quality. A model can reach the correct answer for the wrong reason, and the rationale may still be added to the training set. Process supervision, proof checking, or execution can strengthen the filter, but then the method becomes more domain-specific.

3.2.3 ReST-EM: reinforced self-training with expectation–maximization

Reinforced Self-Training (ReST) is a growing-batch reinforcement learning approach for language modeling. It alternates between a Grow step, where the current policy generates candidate outputs, and an Improve step, where candidates are filtered or weighted by a reward model and used to update the policy. Although the original ReST paper focuses on machine translation, the pattern has become a general template for LLM self-training. The EM interpretation is natural: the unobserved high-quality response is treated as a latent variable; the E-like step samples and scores possible responses; the M-like step updates the model on the selected or weighted responses.

The ReST loop is

$$\mathcal{D}_g^{(t)} = \{(x_i, y_{ij}) : x_i \sim \mathcal{D},\ y_{ij} \sim \pi_{\theta_t}(\cdot \mid x_i),\ j = 1, \dots, N\} \cup \mathcal{D}, \tag{50}$$

$$F(x, y; \tau_t) = \mathbf{1}\{R(x, y) > \tau_t\}, \tag{51}$$

$$J(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}_g^{(t)}}\left[F(x, y; \tau_t)\mathcal{L}(x, y; \theta)\right], \tag{52}$$

$$\theta_{t+1} = \arg\min_{\theta} J(\theta). \tag{53}$$

The threshold τ_t can increase over Improve steps, creating a curriculum of higher-reward but smaller datasets. The loss $\mathcal{L}$ can be standard negative log-likelihood on filtered samples or an offline RL loss. The key efficiency benefit is that the expensive Grow step can be amortized across multiple Improve steps.

ReST is self-improving only to the extent that the reward function is aligned with true quality. In machine translation, reward models such as MetricX, BLEURT, or COMET provide dense automatic scores. In code, execution and tests can play the same role. In open-ended dialogue, reward models are harder to trust. The ReST paper reports that each additional Improve step increased reward model scores across IWSLT 2014 De-En, WMT 2020 Zh-En, and Web Domain En-Zh validation sets, and that a second Grow step improved performance by 5.3 points on IWSLT 2014 De-En and 0.8 points on Web Domain En-Zh over the first Grow step. These results support the growing-batch premise: improved policies generate better candidates, and better candidates support further improvement.

ReST-EM is a useful abstraction for many later systems. Rejection sampling fine-tuning, best-of-N distillation, self-training on verified code, and RLVR can all be seen as variants of the same loop. The design choices are the candidate distribution, the scoring function, the selection threshold, the training loss, and the schedule. The failure modes are also shared: reward hacking, loss of diversity, overfitting to the reward model, and collapse if the selected data is too narrow.

Figure 13: ReST / ReST-EM style self-training

Input: Initial policy π_{θ_0}, prompts $\mathcal{X}$, reward R, grow steps G, improve steps I

for $g = 1$ to G **do**

1. Generate candidates $\mathcal{D}_g \leftarrow \{(x, y) : x \in \mathcal{X}, y \sim \pi_\theta(\cdot \mid x)\}$
2. Score candidates with $R(x, y)$

for $i = 1$ to I **do**

3. Choose threshold τ_i or weights $w_i(x, y)$
4. Update $\theta \leftarrow \arg\min_\theta \mathbb{E}_{(x,y)\in\mathcal{D}_g}[w_i(x, y)\mathcal{L}(x, y; \theta)]$
5. **return** π_θ

3.2.4 Constitutional AI: self-alignment from AI feedback

Constitutional AI, introduced by Bai et al., is a self-improvement method for alignment rather than task accuracy. It uses a human-written list of principles—a constitution—to guide critique, revision, preference labeling, and reinforcement learning. The method has two phases. In the supervised phase, an initial model responds to potentially harmful prompts, critiques its own responses according to constitutional principles, revises them, and is fine-tuned on the revised responses. In the reinforcement learning phase, the model samples pairs of responses; an AI feedback model compares them according to the constitution; a preference model is trained on these AI-generated preferences; and RL uses the preference model as reward. The only direct human oversight for harmlessness is the constitution itself, not per-example harm labels.

The supervised Constitutional AI loop can be written as

$$y_i \sim \pi_{\theta_0}(\cdot \mid x_i), \tag{54}$$

$$c_i \sim \pi_{\theta_0}(\cdot \mid x_i, y_i, \mathcal{C}), \tag{55}$$

$$y'_i \sim \pi_{\theta_0}(\cdot \mid x_i, y_i, c_i, \mathcal{C}), \tag{56}$$

$$\theta_{\mathrm{SL}} = \arg\max_\theta \sum_i \log \pi_\theta(y'_i \mid x_i), \tag{57}$$

where $\mathcal{C}$ is the constitution. The RLAIF phase forms AI preference labels

$$z_i = \mathrm{AIJudge}_{\mathcal{C}}(x_i, y_i^{(1)}, y_i^{(2)}) \in \{1, 2\}. \tag{58}$$

trains a preference model $r_\phi(x, y)$, and optimizes a KL-regularized RL objective

$$\max_\theta \ \mathbb{E}_{x,y\sim\pi_\theta}[r_\phi(x, y)] - \beta\, \mathbb{E}_x\, \mathrm{KL}\big(\pi_\theta(\cdot \mid x) \| \pi_{\mathrm{ref}}(\cdot \mid x)\big). \tag{59}$$

Constitutional AI is important for self-evolution because it demonstrates that feedback can be generated by the model itself while still being anchored to human intent. The constitution acts as a compact human prior. The model expands that prior into many critiques and preferences. This is a scalable supervision pattern: humans specify principles; AI applies principles to cases. It is

not supervision-free, because the principles are human choices, but it avoids the need for humans to label every harmful output pair.

The method also exposes a deep issue for self-improving systems: a self-generated feedback loop inherits the values and blind spots of its constitution and judge. If the principles are vague, incomplete, culturally narrow, or internally inconsistent, the trained model will reflect those defects. If the AI judge systematically prefers evasive answers, the policy may become harmless but unhelpful. Bai et al. emphasize producing a harmless but non-evasive assistant, which means the optimization must balance refusal, explanation, and helpful redirection. In mathematical terms, alignment is multi-objective:

$$r(x, y) = \alpha r_{\text{helpful}}(x, y) + \beta r_{\text{harmless}}(x, y) + \gamma r_{\text{honest}}(x, y) - \delta r_{\text{evasive}}(x, y). \tag{60}$$

Self-improvement in this setting is not monotonic on a single scalar capability; it is movement along a Pareto frontier.

3.2.5 RISE: recursive self-improvement via reinforcement learning

RISE, Recursive IntroSpEction, moves self-correction from prompting into learned behavior. The local research corpus describes RISE as modeling multi-round reasoning as an MDP and using reinforcement learning to teach a model when to continue, revise, or terminate. The state contains the problem and the trajectory so far; the action is a reasoning or revision move; the reward is based on final-answer correctness. This is a key conceptual shift from Self-Refine. In Self-Refine, the model follows a prompt that says to critique and revise. In RISE, the policy is trained so that introspection becomes part of the model's action distribution.

The MDP formulation is

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma), \tag{61}$$

with

$$s_t = (x, \tau_{1:t}), \tag{62}$$

$$a_t \in \{\text{CONTINUE}, \text{REVISE}, \text{TERMINATE}\} \times \mathcal{Y}, \tag{63}$$

$$s_{t+1} = f(s_t, a_t), \tag{64}$$

$$R(\tau) = \mathbf{1}\{\text{Ans}(\tau) = a^\star\}. \tag{65}$$

The training objective can be expressed as

$$\max_\theta \ \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=1}^{T} \gamma^t r_t \right] - \beta \, \text{KL}(\pi_\theta \| \pi_{\text{ref}}). \tag{66}$$

The KL term prevents the learned self-revision policy from drifting too far from the reference model.

The local corpus reports GSM8K improvements from roughly 14% to roughly 24% for Llama2-7B and from roughly 38% to roughly 46% for Mistral-7B. The same summary reports significant gains on MATH. The important lesson is not the absolute score, but the fact that multi-turn self-correction can be trained. RISE makes introspection an acquired skill rather than a prompt instruction.

RISE's design also highlights the challenge of credit assignment. If a model produces five reasoning steps, revises step three, and eventually answers correctly, which action deserves credit? A terminal reward gives sparse feedback. Process rewards can help, but they require trustworthy intermediate evaluation. This is the same challenge later addressed by trajectory-level and turn-level RL methods such as RAGEN and ReVeal.

3.2.6 RAGEN: trajectory-level RL for agents

RAGEN generalizes self-improvement from reasoning answers to multi-turn agents. Its StarPO framework decomposes trajectories into State, Thinking, Action, and Reward components:

$$\tau = \{(s_1, t_1, a_1, r_1), \ldots, (s_T, t_T, a_T, r_T)\}. \tag{67}$$

The local corpus gives the StarPO objective as

$$\max_{\theta} \mathbb{E}\left[\sum_t \gamma^t r_t \log \pi_\theta(a_t \mid s_t, t_t)\right]. \tag{68}$$

This expression captures the policy-gradient intuition, although practical implementations may use PPO-like clipped objectives, advantage estimates, and KL regularization.

RAGEN's most notable contribution is the identification of the "Echo Trap." In multi-turn RL, an agent may learn to repeat its previous thoughts or actions rather than reason. This can create apparent training improvement while destroying generalization. The local summary describes an echo detector based on similarity between consecutive thoughts,

$$\text{echo}(t) = \mathbf{1}\{\cos\_\text{sim}(h(t_t), h(t_{t-1})) > \eta\}, \tag{69}$$

and a trajectory filter

$$\text{Keep}(\tau) = \mathbf{1}\left\{\frac{1}{T}\sum_{t=2}^{T} \text{echo}(t) < \eta_{\text{ratio}}\right\}. \tag{70}$$

StarPO-S applies such filtering and stabilization to reduce reasoning collapse.

RAGEN matters because self-evolving agents are not single-turn answerers. They inspect environments, call tools, remember previous observations, and adapt plans. Training such agents requires trajectory-level objectives. It also requires diagnostics for degenerate behavior. Echo Trap is one example; others include tool-call loops, reward-model overoptimization, premature termination, and observation neglect. A method that improves benchmark reward while increasing these pathologies is not a reliable self-evolution method.

3.2.7 ReVeal, Absolute Zero, and verifier-centric extensions

Two recent methods extend the training-time loop toward stronger autonomy. Absolute Zero proposes a zero-data self-play reasoning paradigm in which a model both proposes tasks and solves them, while a code executor verifies solutions. The local corpus describes the core loop as task proposal $t = \text{LLM}_{\text{proposer}}(\text{curriculum})$, solution $s = \text{LLM}_{\text{solver}}(t)$, verification $r = \text{Execute}(s, t) \in \{0, 1\}$, and RL updates for both proposer and solver. The proposer reward balances difficulty, solvability, and diversity:

$$R_{\text{prop}}(t) \propto \text{difficulty}(t)\,\text{solvability}(t)\,\text{diversity}(t). \tag{71}$$

The key idea is that the model becomes both curriculum designer and student. This is closer to AlphaGo Zero style self-play than SPIN because no external SFT prompts are required for each new task, though the verifier constrains the domain.

ReVeal focuses on self-evolving code agents via reliable self-verification. The local corpus describes a multi-turn RL framework that optimizes code generation and self-verification together. A trajectory is

$$\tau = \{(c_1, v_1, r_1), \ldots, (c_T, v_T, r_T)\}, \tag{72}$$

where c_t is code, v_t is verification activity such as test generation and tool execution, and r_t is turn-level reward. Its co-evolution loss is summarized as

$$\mathcal{L} = \mathcal{L}_{\text{generation}} + \lambda \mathcal{L}_{\text{verification}}. \tag{73}$$

The reported headline is that ReVeal can perform 20+ inference-time evolution turns on LiveCodeBench despite training on only 3 turns. This is an important capability: a model trained on short self-verification trajectories generalizes to longer inference-time improvement loops.

These verifier-centric methods suggest the direction of the field. Self-improvement becomes reliable when the system can create tasks, attempt them, verify them, and update itself without relying solely on its own verbal judgment. The verifier is the anchor. The open problem is breadth: code executors and math checkers are powerful but domain-limited. For open-ended real-world tasks, reliable verification remains difficult.

3.3 Comparison Table

Table 5 compares the main methods along dimensions that matter for system design. "Loop type" indicates whether the update occurs at inference, across episodes, or through training. "Training required" distinguishes prompt-only methods from parameter-updating methods. "Feedback source" identifies the grounding signal. "Applicable tasks" summarizes the domains where the method is most natural. "Key results" reports representative numbers from the research corpus and primary papers.

Table 5: Comparison of self-improvement methods by loop type, training requirement, feedback source, task domain, and representative results.

Method	**Loop type**	**Training required**	**Feedback source**	**Applicable tasks**	**Key results / evidence**
Self-Refine	Inference-time draft refinement	No	Same-model critique under rubric	Writing, code style, constrained generation, math reasoning, dialogue	Average $\sim$ 20% improvement across seven tasks; math 56% $\rightarrow$ 67%; sentiment reversal 74% $\rightarrow$ 86%; dialogue win rate 52% $\rightarrow$ 68%
Reflexion	Cross-episode inference memory	No parameter update	Evaluator reward plus verbal reflection memory	Code generation, ALFWorld, WebShop, interactive agents	HumanEval pass@1 91.0%; GPT-4 zero-shot reference 80.1%; Codex fine-tuned 67.0%; ALFWorld 97% vs 75% base
Self-Debug / SelfEvolve	Inference-time execution repair	No for repair; optional training	Compiler, interpreter, tests, tracebacks	Code generation, data-science programming, translation between languages	SelfEvolve DS-1000 49.4 $\rightarrow$ 57.2; HumanEval pass@1 74.39 $\rightarrow$ 85.98; TransCoder accuracy 88.3 $\rightarrow$ 90.4
SPIN	Iterative self-play fine-tuning	Yes	Human SFT response vs previous model response	Chat alignment, general LLM benchmarks	Zephyr-7B GSM8K 26.76 $\rightarrow$ 38.97; average leaderboard 58.14 $\rightarrow$ 63.16; SPIN+DPO average 64.05
STaR	Iterative rationale bootstrapping	Yes	Final-answer correctness; self-generated rationales	Arithmetic, CommonsenseQA, GSM8K-like reasoning	GSM8K 5.8% $\rightarrow$ 10.7%; CommonsenseQA dev about 72.5%

Table 5: Comparison of self-improvement methods (continued).

Method	Loop type	Training required	Feedback source	Applicable tasks	Key results / evidence
ReST-EM	Grow–Improve offline RL / EM	Yes	Reward model filtering or weighting	Machine translation, preference alignment, best-of-N distillation	ReST Improve steps raise reward scores across IWSLT/WMT/Web; second Grow adds 5.3 points on IWSLT and 0.8 on Web Domain
Const. AI	SL critique-revision plus RLAIF	Yes	Human-written constitution; AI critiques and preferences	Harmless/helpful assistants, safety alignment	Trains harmless but non-evasive assistant with far fewer human harm labels; uses SL-CAI and RLAIF phases
RISE	Multi-turn RL for introspection	Yes	Final-answer reward over reasoning trajectory	Math reasoning, multi-step QA	GSM8K Llama2-7B roughly 14% $\rightarrow$ 24%; Mistral-7B roughly 38% $\rightarrow$ 46%
RAGEN	Trajectory-level agent RL	Yes	Step and trajectory rewards; echo filtering	Tool agents, multi-turn decision tasks	StarPO/StarPO-S improves multi-turn agent training and identifies Echo Trap / reasoning collapse
Absolute Zero	Self-play task proposal and solving	Yes	Code executor verification	Code, math, logic with verifiable rewards	AZR-Coder-7B reported SOTA at 7B level on coding average; zero external training data
ReVeal	Multi-turn code RL with self-verification	Yes	Generated tests, tool execution, turn-level reward	Code agents, LiveCodeBench	20+ inference turns despite training on 3 turns; self-verification optimized explicitly

The table reveals three recurring axes. The first is the state update axis: draft-only, memory-only, data-only, parameter, trajectory-policy, or code/agent architecture. The second is the feedback reliability axis: same-model critique is cheapest but weakest; execution and formal verification are strongest but narrowest; reward models and AI judges are broad but vulnerable to misspecification. The third is the amortization axis: inference-time methods pay at every query, while training-time methods pay up front and amortize later. A system designer should choose the method by matching these axes to the deployment setting. If tasks are rare and high value, inference-time refinement with verification may be best. If tasks are common and verifiable, training on self-generated verified traces can amortize cost. If tasks are safety-critical, self-generated feedback must be anchored to human principles, external audits, or conservative verifiers.

3.4 Benchmark Data

Benchmark claims in self-improvement papers require careful normalization. A loop that uses five attempts and a verifier should not be compared naively with a one-shot model. A method that trains on answer-labeled datasets should not be described as zero-data. A method that improves an average score may still regress on individual tasks. This section consolidates exact numbers available in the project research corpus and primary papers for HumanEval, GSM8K, BBH, MMLU, and SWE-Bench. The purpose is not to crown a single winner, but to show which feedback loops have

Benchmark	Method / model	Baseline	Self-improved	Notes
HumanEval	Reflexion (GPT-3.5)	GPT-4 zero-shot 80.1%; Codex fine-tuned 67.0%	91.0% pass@1	Inference-time verbal memory with evaluator
HumanEval	Agent Symbolic Learning	Agents 68.3%	73.2% pass@1	Language-gradient agent optimization
HumanEval	SelfEvolve	ChatGPT 74.39% pass@1; 81.10% pass@10	85.98% pass@1; 87.81% pass@10	Execution/self-generated knowledge repair; GPT-4 reference 89.95% pass@1
GSM8K	SPIN on Zephyr-7B-SFT-full	26.76	38.97 at iteration 3	Self-play fine-tuning; MMLU slightly decreases
GSM8K	RISE	Llama2-7B $\sim$ 14%; Mistral-7B $\sim$ 38%	$\sim$ 24%; $\sim$ 46%	Multi-turn RL introspection
GSM8K	STaR	5.8% answer-only	10.7%	Rationale bootstrapping on early models
BBH subset	SPIN causal judgment	56.15	59.36 at iteration 2	Few-shot CoT BBH subset
BBH subset	SPIN formal fallacies	49.6	51.2	Modest improvement
BBH subset	SPIN sports understanding	96.0	94.4	Regression despite average gains
MMLU	SPIN on Zephyr-7B-SFT-full	60.92	59.99 at iteration 3; 59.68 after SPIN+DPO	Shows non-monotonic per-benchmark behavior
SWE-Bench	Darwin Godel Machine	20.0%	50.0% verified	Agent/code evolution rather than pure model fine-tuning

Table 6: Exact benchmark numbers cited in the research corpus and primary papers for representative self-improvement methods. Percent signs are shown where the source reports percentages; otherwise the table preserves the source score format.

evidence on which evaluation regimes.

HumanEval. HumanEval is the clearest success case for inference-time and execution-grounded self-improvement. Reflexion reports 91.0% pass@1, exceeding the 80.1% GPT-4 zero-shot reference and the 67.0% Codex fine-tuned reference in the local summary. SelfEvolve reports 85.98% pass@1 and 87.81% pass@10, compared with ChatGPT at 74.39% and 81.10%, respectively. Agent Symbolic Learning reports an increase from 68.3% to 73.2%. These numbers share a theme: code tasks provide relatively strong feedback, either through tests, execution, or evaluator signals. That makes them fertile ground for self-improvement loops.

GSM8K. GSM8K illustrates the diversity of training-time self-improvement. STaR nearly doubles an early answer-only baseline from 5.8% to 10.7% by bootstrapping rationales. RISE improves Llama2-7B from about 14% to about 24% and Mistral-7B from about 38% to about 46% by training multi-turn introspection. SPIN improves Zephyr-7B-SFT-full from 26.76 to 38.97 through self-play fine-tuning. These are different mechanisms—rationale SFT, RL introspection, and self-play contrastive fine-tuning—but all use generated reasoning behavior as training signal.

BBH. The SPIN paper reports BBH subset results for causal judgment, formal fallacies, and sports understanding. Causal judgment improves from 56.15 to 59.36 by iteration 2; formal fallacies improves from 49.6 to 51.2; sports understanding decreases from 96.0 to 94.4. The mixed result is important. A self-improvement loop can improve the average while harming saturated

or distribution-sensitive tasks. BBH-style evaluations should therefore be reported per task rather than only as an aggregate.

MMLU. SPIN is also instructive on MMLU. Zephyr-7B-SFT-full starts at 60.92; SPIN iteration 3 reports 59.99; SPIN iteration 3 followed by DPO reports 59.68. The average across six leaderboard tasks improves from 58.14 to 63.16 under SPIN and 64.05 after DPO, but MMLU does not improve. This suggests that self-play on instruction-following data may strengthen conversational, truthfulness, and math behavior without necessarily improving broad factual knowledge. MMLU is therefore a useful guardrail benchmark for detecting knowledge regressions during self-improvement.

SWE-Bench. SWE-Bench appears in the research corpus through Darwin Godel Machine, which is broader than the methods in this chapter but highly relevant to self-evolution. DGM improves SWE-Bench verified performance from 20.0% to 50.0% by evolving agent implementations in an archive. This result shows that self-improvement can occur outside model weights: the system improves by modifying agent code, prompts, and workflows. For a survey on AI self-evolution, SWE-Bench is therefore a bridge benchmark between model self-training and agent architecture evolution.

3.5 Cross-Method Synthesis

The methods in this chapter can be placed on a ladder of increasingly strong feedback. At the bottom is same-model critique, as in Self-Refine. It is broadly applicable and cheap to set up, but weakly grounded. Next is memory-conditioned verbal reinforcement, as in Reflexion. It preserves lessons across attempts but depends on evaluator quality and memory management. Next is execution-grounded repair, as in Self-Debug and SelfEvolve. It provides reliable signals where execution is available. Next is filtered self-training, as in STaR and ReST, where generated data becomes training data after correctness or reward filtering. Next is preference and self-play optimization, as in DPO, SPIN, and Constitutional AI. Finally, trajectory-level RL methods such as RISE, RAGEN, and ReVeal train policies for multi-turn self-correction and tool use.

This ladder is not strictly ordered by performance. A simple Self-Debug loop can beat a complex RL method on a code task if tests are strong. A prompt-only Self-Refine loop can be preferable to training if the task distribution changes daily. A Constitutional AI loop may be necessary for safety even if it does not improve benchmark accuracy. The point of the ladder is to clarify assumptions. Each higher rung requires more infrastructure: data generation, filtering, reward modeling, training pipelines, safety checks, and evaluation. Each lower rung requires more inference-time computation and accepts weaker amortization.

A recurring mathematical pattern is contrast. Self-Refine contrasts a draft with a rubric. Reflexion contrasts a failed trajectory with a desired future behavior. Self-Debug contrasts program behavior with tests. SPIN contrasts human responses with previous-model responses. DPO contrasts preferred and rejected responses. STaR contrasts rationales that lead to correct answers with those that do not. RAGEN contrasts productive trajectories with echo-trap trajectories. Self-improvement is rarely produced by generation alone; it is produced by generation plus a selection or correction signal.

Another recurring pattern is curriculum. Reflexion's memory accumulates lessons over episodes. STaR's model solves more examples as its rationale ability improves. SPIN's negatives become harder as the previous model improves. ReST's thresholds rise over Improve steps. Absolute Zero's proposer must generate tasks that are neither trivial nor impossible. Curriculum is essential because

self-generated data can be too easy, too hard, or too narrow. A self-improving system must regulate difficulty to stay in the learning zone.

The strongest empirical results often occur in domains with verifiers. HumanEval, DS-1000, TransCoder, LiveCodeBench, and SWE-Bench all provide some form of executable or objective feedback. Math benchmarks provide final-answer checks, though not always process checks. Open-ended dialogue and safety require preference models or constitutions, which are broader but less objective. This creates a research frontier: extending reliable verification beyond code and math. Retrieval-grounded QA, scientific reasoning, long-horizon web tasks, and real-world agents need evaluators that are both semantically rich and resistant to reward hacking.

3.6 Failure Modes and Evaluation Principles

Self-improvement methods share a family of failure modes. The first is circular critique. If the same model generates, critiques, and revises without new information, the loop may amplify the model's prior rather than correct it. The second is reward hacking. If a reward model or test suite is incomplete, the model may learn to exploit it. The third is mode collapse. Iterative filtering can reduce data diversity until the model becomes narrow or repetitive. The fourth is memory poisoning. Incorrect reflections can persist and misguide future episodes. The fifth is benchmark overfitting. Self-improvement loops can tune to visible benchmarks while harming generalization. The sixth is compute confounding. A method may appear better simply because it uses more samples, retries, or tokens.

Evaluation should therefore include compute-matched baselines. Self-Refine should be compared with best-of-N sampling using the same token budget. Reflexion should be compared with repeated attempts without memory. Self-Debug should be compared with test-driven repair without natural-language explanations. SPIN should be compared with additional SFT epochs, DPO, and rejection sampling. STaR should be compared with fine-tuning on answer-only data and with human rationale data where available. RISE and RAGEN should be compared with prompted multi-turn reasoning under the same inference budget. Without such comparisons, the contribution of the self-improvement mechanism is unclear.

Evaluation should also preserve intermediate artifacts. For inference loops, store drafts, critiques, revisions, tests, and reflections. For training loops, store generated candidates, scores, filters, preference labels, and rejected examples. For RL loops, store trajectories, rewards, advantages, KL values, and degeneracy diagnostics such as echo ratios. These artifacts allow researchers to distinguish genuine improvement from evaluator exploitation.

Finally, evaluation should report regressions. SPIN's MMLU and BBH sports-understanding numbers are valuable precisely because they show that average gains can hide task-specific declines. Self-evolution systems should be treated like continual learning systems: every update risks forgetting or behavioral drift. A credible report should include a safe baseline freeze, a single-variable ablation, held-out tasks, and rollback criteria.

3.7 Practical Design Guidelines

For practitioners, the simplest guideline is to choose the strongest feedback source available. If code can be executed, execute it. If a proof can be checked, check it. If a database answer can be validated, validate it. Use same-model critique only when stronger feedback is unavailable or as a supplementary explanation layer. The second guideline is to separate roles and prompts. Generator, critic, verifier, reviser, and memory writer should have distinct instructions and outputs. The third guideline is to keep the best verified candidate, not necessarily the last candidate. The fourth

guideline is to budget iterations adaptively: easy tasks should stop early, hard tasks may deserve more refinement.

For training-time systems, data governance is central. Generated data should carry provenance: model version, prompt, sampling parameters, score, verifier result, and filtering reason. Preference pairs should record the judge and criterion. Reflections should record the episode that produced them. Without provenance, self-generated data becomes an opaque mixture that is difficult to debug. This is especially important when self-improvement loops run continuously.

Safety requires conservative update gates. A new self-improved model or agent should pass regression tests before deployment. For code agents, this includes sandbox tests and adversarial prompts. For dialogue agents, this includes harmlessness, honesty, privacy, and refusal evaluations. For tool agents, this includes tool misuse and loop detection. For autonomous self-evolution, updates should be reversible. The ability to roll back is as important as the ability to improve.

The final guideline is humility about the word "self." Most successful systems are not self-improving in a fully autonomous philosophical sense. They rely on human-written prompts, answer keys, test suites, reward models, constitutions, or benchmark environments. Their autonomy lies in how they expand, apply, and learn from these signals. A precise survey should therefore describe the feedback source rather than merely saying that the model improves itself.

3.8 A Unified Taxonomy of Feedback, Update, and Selection

The methods above can be further unified by decomposing every self-improvement loop into four operators: a proposal operator, an observation operator, a selection operator, and an update operator. The proposal operator P creates candidate behavior: drafts in Self-Refine, actions in Reflexion, code in Self-Debug, synthetic responses in SPIN, rationales in STaR, and trajectories in RAGEN. The observation operator O converts the candidate into evidence: critiques, scalar rewards, execution traces, answer checks, preference labels, or environment transitions. The selection operator S decides which evidence or candidate should influence the future: keep the revised draft, append the reflection, filter high-reward samples, choose preferred responses, discard echo trajectories, or accept an agent modification. The update operator U changes either the visible output, the context memory, the training set, the model parameters, or the executable agent. In symbols,

$$z_t \sim P_{\theta_t}(\cdot \mid x, h_t), \qquad e_t = O(x, z_t), \qquad q_t = S(z_t, e_t, h_t), \qquad (\theta_{t+1}, h_{t+1}) = U(\theta_t, h_t, q_t), \tag{74}$$

where h_t denotes all non-parametric state. This factorization is useful because many apparent differences between papers are differences in only one operator. Self-Refine and Self-Debug both update a candidate answer, but their observation operators differ: one observes a language critique, the other observes an execution trace. SPIN and DPO both train on contrasts, but their proposal and selection operators differ: DPO usually receives a preference dataset, while SPIN proposes losers from the previous policy. RISE and RAGEN both optimize trajectories, but RAGEN adds selection against echo-like degeneracy.

The feedback source is the most important operator because it determines the ceiling of trustworthy improvement. We can rank feedback sources by epistemic strength, while recognizing that the ranking is domain-dependent. At the weakest end is self-consistency of style: a model decides whether its answer sounds better. This is useful for fluency but weak for correctness. Next is rubric-grounded critique: the model checks explicit constraints supplied by the user. Next is answer-key verification: the model or program checks a final answer. Next is execution: a programming language, environment, theorem prover, simulator, or test harness produces an objective trace. Next is preference feedback: a human, reward model, or constitutional judge compares outputs. Preference feedback is broad but not necessarily more reliable than execution; it is stronger for subjective

values and weaker for factual correctness. Finally, there is deployment feedback: users, markets, scientific experiments, or real-world environments provide consequences. Deployment feedback is rich but costly, noisy, delayed, and ethically constrained.

The update target determines whether improvement is transient or amortized. Updating only y_t improves the current answer but leaves the next query unchanged. Updating memory m_t improves future episodes within the same context or agent. Updating a dataset $\mathcal{D}_t$ prepares for future training but has no immediate effect until optimization. Updating parameters θ_t amortizes improvement across future queries, but risks broad behavioral drift. Updating code or agent architecture changes the computational process and can create improvements unavailable to weight updates alone. In practice, strong systems combine these targets. A code agent may repair the current file, store a reflection, add the failing case to a regression suite, fine-tune a repair model later, and modify its planning prompt if the same failure recurs.

The selection operator is where most self-improvement systems either succeed or fail. A weak selector admits noisy or harmful data; an overly strict selector starves the learner. STaR's selector keeps only rationales with correct final answers, but this can bias the dataset toward easy problems. ReST's selector uses thresholds that can be raised over time, but too high a threshold can reduce diversity. SPIN's selector implicitly treats human SFT responses as winners and previous-model samples as losers, but if the previous-model samples are already high quality, the margin becomes difficult. RAGEN's selector filters echo trajectories, but if the threshold is too aggressive it may remove legitimate repeated reasoning patterns. Selection must therefore be tuned not only for average quality but also for coverage, diversity, and robustness.

This taxonomy also clarifies how to compare methods fairly. A method with a stronger observation operator should not be credited solely to a clever update rule. If Self-Debug beats Self-Refine on code, the gain may come from execution rather than from a better language prompt. If ReVeal beats a baseline on LiveCodeBench, the gain may come from optimizing verification rather than from longer generation. If Constitutional AI improves harmlessness, the constitution and AI preference model are part of the method, not incidental details. Survey writing should therefore identify the feedback source explicitly in every claim.

3.9 Implementation-Level Design Choices

The papers surveyed here often present concise algorithms, but practical systems require many additional decisions. The first decision is candidate multiplicity. A loop can refine a single candidate through time, or it can generate a batch of candidates and select among them. Single-candidate refinement is cheap and easy to trace. Batch generation is more robust because a selector can choose the best candidate, but it costs more and can hide failure modes if only the final selected answer is inspected. A hybrid strategy is often best: generate N candidates, run a cheap verifier, refine the top K, then run an expensive verifier.

The second decision is whether critique should be free-form or structured. Free-form critique is flexible but difficult to parse and compare. Structured critique uses fields such as `correctness`, `missing constraints`, `edge cases`, `evidence`, and `recommended edit`. Structured formats make it easier to log, retrieve, and train on feedback. In Self-Refine, a structured critique can prevent the model from spending all effort on style. In Reflexion, structured memory can separate general lessons from task-specific facts. In Self-Debug, structured error localization can distinguish syntax errors, runtime exceptions, wrong-answer cases, and performance bottlenecks. For training-time methods, structure supports filtering: one can keep only examples whose critique identifies a concrete, verifiable issue.

The third decision is compression. Memory and trajectory logs grow quickly. Reflexion faces

this directly because every failed episode can add a reflection. If the memory buffer stores raw traces, context length is exhausted. If it stores only short summaries, important details may be lost. A practical memory system should maintain multiple layers: raw logs for offline audit, concise reflections for immediate context, embeddings for retrieval, and periodically distilled rules for long-term behavior. The compression function should be conservative: it should preserve the task, failure condition, root cause, and future instruction. A good reflection is not "be more careful." It is "When a coding problem requires preserving input order, avoid converting the list to a set; the previous failure lost duplicates and order."

The fourth decision is verifier design. Execution verifiers are not binary in practice. A program can pass sample tests but fail hidden tests; a generated unit test can be wrong; a timeout can indicate either inefficient code or an overly strict resource limit. Verifier output should therefore include uncertainty. Instead of treating $V(x, y) \in \{0, 1\}$ as absolute, systems can use a vector

$$V(x, y) = \big(v_{\text{syntax}}, v_{\text{samples}}, v_{\text{hidden}}, v_{\text{property}}, v_{\text{resource}}, u\big), \tag{75}$$

where u is uncertainty or coverage. A repair loop should prioritize failures with high diagnostic value. A training loop should avoid treating low-coverage passes as strong positives.

The fifth decision is rollback. Any self-improvement loop can make things worse. In inference-time systems, rollback means returning the best verified candidate rather than the last candidate. In memory systems, rollback means removing or down-weighting a reflection shown to be harmful. In training systems, rollback means preserving model checkpoints and evaluating updates against regression suites. In agent architecture evolution, rollback means rejecting code modifications that fail validation. A self-evolving system without rollback is not an optimizer; it is an uncontrolled drift process.

The sixth decision is how to allocate compute. Some tasks benefit from many cheap samples; others benefit from one deep refinement trajectory. A math problem may require long reasoning; a JSON-formatting task may need only one validation pass; a code problem may need repeated test-and-repair. Adaptive compute policies can be learned or heuristic. For example,

$$B(x) = B_{\min} + \alpha \ \text{uncertainty}(x) + \beta \ \text{failure\_severity}(x) + \gamma \ \text{value}(x), \tag{76}$$

where $B(x)$ is the iteration budget. RISE's observation that models learn to allocate more introspection to harder problems points toward such adaptive policies.

3.10 Method-by-Method Failure Analysis

Self-Refine fails when the critique channel has no new information. A common pattern is cosmetic convergence: the answer becomes more polished, more verbose, and more confident, but not more correct. Another pattern is instruction drift: the revision optimizes a critique that partially contradicts the original user request. For example, a critic may ask for more explanation even when the task requested brevity. A third pattern is oscillation: one iteration adds detail, the next removes it, and the loop consumes budget without improvement. Mitigations include explicit rubrics, verifier checks, diff-based editing, and a rule that revisions must preserve satisfied constraints.

Reflexion fails through memory errors. One failure is overgeneralization: from a single failed episode, the agent writes a broad rule that harms future tasks. Another is stale memory: advice that was useful under one environment version remains in context after the environment changes. A third is retrieval mismatch: the agent retrieves a reflection that is semantically similar but operationally irrelevant. A fourth is reflection hallucination: the self-reflection model invents a cause not supported by the trajectory. Mitigations include grounding reflections in quoted evidence

from the trace, attaching scope conditions, decaying old memories, and using evaluator-confirmed counterfactuals before creating general rules.

Self-Debug fails when tests are weak or misleading. It can overfit to visible examples, hard-code outputs, or patch the immediate exception while leaving the underlying algorithm wrong. It can also be trapped by non-local bugs: the error appears in one function but originates in an earlier design choice. Another failure is destructive repair, where the model rewrites working parts of the program and introduces new bugs. Mitigations include generated adversarial tests, property-based testing, minimal diffs, line-level localization, and preserving passing behavior as regression tests.

SPIN fails when the self-play negatives do not define the right contrast. If the previous model's responses are obviously worse than human responses, the task is too easy and learning saturates. If they are nearly indistinguishable, optimization may be noisy. If human responses contain biases or outdated style, SPIN amplifies them. If the prompt distribution is narrow, self-play does not broaden competence. The MMLU non-improvement reported in Table 6 is an example of a capability boundary: a method can improve instruction-following averages without increasing broad factual knowledge. Mitigations include mixing diverse prompts, preserving a knowledge regression suite, combining SPIN with retrieval or domain data, and monitoring per-benchmark deltas.

STaR fails through rationale unfaithfulness. A rationale can be fluent and lead to the correct answer while not representing the true causal computation. Fine-tuning on such rationales may teach the model to imitate plausible explanations rather than robust reasoning. The rationalization step intensifies this risk because the model knows the answer and can work backward. Mitigations include process verifiers, intermediate checks, adversarial answer choices, and training objectives that reward consistency under perturbations. Another mitigation is to store negative rationales and teach the model to distinguish why they fail, rather than only imitating positives.

ReST-EM fails when the reward model becomes the task. The model learns to optimize reward-model features that correlate imperfectly with human or task quality. In translation, this might mean exploiting metric biases; in dialogue, it might mean producing verbose responses that a reward model likes; in code, it might mean passing weak tests. Threshold schedules can also reduce diversity: if only the highest-scoring candidates survive, rare but valuable solution styles disappear. Mitigations include reward ensembles, held-out human evaluation, diversity constraints, conservative thresholds, and periodic refresh of the prompt distribution.

Constitutional AI fails when principles are underspecified or when the AI judge applies them mechanically. A constitution can say to be harmless, but real prompts require tradeoffs between helpfulness, autonomy, privacy, legality, and user intent. If the critique model is too cautious, the assistant becomes evasive; if it is too permissive, the assistant remains unsafe. If the constitution is not transparent, downstream users cannot understand the normative basis of refusals. Mitigations include explicit principle hierarchies, adversarial red-teaming, human audits of AI preference labels, and separate reporting of helpfulness and harmlessness metrics.

RISE and RAGEN fail through trajectory-level pathologies. Sparse rewards make credit assignment hard. Long trajectories create many opportunities for the model to repeat itself, ignore observations, or learn superficial markers of reasoning. The Echo Trap is especially important because it resembles reasoning in logs: the agent appears to think, but the content is copied or circular. Mitigations include turn-level rewards, echo filters, entropy bonuses, observation-attention diagnostics, and evaluation on out-of-distribution environments. ReVeal's turn-level adaptive policy optimization is one response to this general problem: credit must be assigned at the granularity where behavior can be corrected.

3.11 Relation to Broader Self-Evolving Agent Systems

Although this chapter focuses on model and reasoning methods, the same loop structure appears in agent architecture evolution. Agent Symbolic Learning treats prompts, tools, and pipelines as symbolic weights and computes language gradients. The local corpus reports HotPotQA F1 improving from 36.2 to 39.1, MATH from 56.0% to 60.7%, HumanEval from 68.3% to 73.2%, and a software development executability score from 2.4 to 3.8. This is not parameter fine-tuning; it is optimization in the symbolic design space around the model. The connection to Self-Refine is direct: both use language feedback as an update signal, but Symbolic Learning propagates feedback through agent components rather than only rewriting an answer.

Darwin Godel Machine extends the update target further to executable agent code and open-ended archives. The local corpus reports SWE-Bench improvement from 20.0% to 50.0% and Polyglot score from 14.2% to 30.7%. Here the proposal operator is an LLM generating code modifications; the observation operator is benchmark evaluation; the selection operator is archive insertion; the update operator changes the population of agents. DGM therefore resembles SPIN and ReST in its iterative improvement structure, but the object being improved is an implementation rather than a model distribution.

ADAS, AlphaEvolve, and related evolutionary coding systems show that self-improvement can be population-based rather than single-policy. A population maintains diversity, reducing the risk of local optima. MAP-Elites-style archives preserve high-performing solutions in different behavioral niches. This is relevant to training-time self-improvement because model fine-tuning tends to collapse many candidate behaviors into one parameter vector. A future self-evolving AI system may need both: parameter updates for amortized skill and archives for diversity, fallback, and exploration.

The conceptual bridge is that all these systems perform search under feedback. Prompt refinement searches in text space. Reflexion searches in memory-conditioned behavior space. STaR searches in rationale space. SPIN searches in policy space. ReST searches in generated-output datasets. RAGEN searches in trajectory-policy space. DGM searches in code space. The survey category "AI self-evolution" is therefore not defined by a single algorithm, but by a family of feedback-driven searches where the AI system contributes candidates, critiques, or updates to its own future behavior.

3.12 Open Problems for Self-Improvement Methods

The first open problem is verifier generalization. Code and math benefit from relatively crisp verification. Most real tasks do not. A business strategy, scientific hypothesis, medical explanation, legal argument, or long-form plan cannot be fully verified by unit tests. The field needs evaluators that combine retrieval, simulation, tool use, expert rules, uncertainty estimation, and human escalation. Without better verifiers, self-improvement in open domains will remain vulnerable to polished hallucination.

The second open problem is faithful process supervision. Final-answer correctness is often insufficient. STaR can keep rationales that reach correct answers for wrong reasons. RISE can reward a trajectory whose final answer is correct even if intermediate revisions were unnecessary. ReVeal addresses this in code with turn-level rewards, but general reasoning lacks widely available process verifiers. Research directions include proof-carrying reasoning, executable scratchpads, causal intervention on reasoning traces, and consistency checks across paraphrases and decompositions.

The third open problem is lifelong memory governance. Reflexion-style memory is powerful but dangerous. What should be remembered? When should memory expire? How should contradictions

be resolved? Can a model challenge its own old reflections? How can users inspect and edit agent memory? A self-evolving agent needs memory hygiene analogous to data hygiene in machine learning. Otherwise, memory becomes an uncurated training set inside the context window.

The fourth open problem is safe autonomous data generation. SPIN, STaR, ReST, Absolute Zero, and ReVeal all depend on generated data. Generated data can contain hidden contamination, bias, spurious rationales, or reward-model exploits. The more autonomous the data loop, the more important provenance and filtering become. Future systems should attach metadata to every generated sample: source model, prompt, seed, verifier, reward, selection decision, and downstream effect on metrics. This would make self-improvement auditable.

The fifth open problem is multi-objective improvement. Capability, helpfulness, harmlessness, honesty, efficiency, and diversity do not always move together. SPIN's average gains with MMLU regression show this in miniature. Constitutional AI makes the multi-objective nature explicit for alignment. A mature self-evolution framework should optimize a vector of metrics and maintain Pareto frontiers rather than a single scalar leaderboard score. It should also allow domain-specific priorities: a medical assistant may sacrifice creativity for safety, while a brainstorming assistant may accept more diversity.

The sixth open problem is theoretical characterization. Most self-improvement methods are empirically motivated. We lack strong guarantees about convergence, stability, or monotonic improvement. SPIN has a distributional interpretation related to matching the data distribution. DPO has a derivation from KL-regularized reward optimization. ReST has an EM-like interpretation. But Reflexion memory, Self-Refine critique, and agent code evolution are harder to analyze. A useful theory would characterize when language feedback is informative, when self-generated data improves sample complexity, and when iterative updates amplify errors.

The seventh open problem is social and institutional evaluation. Self-improving systems can change after deployment. Users, auditors, and regulators need to know what changed, why it changed, and whether it passed tests. This requires model cards and system cards that include self-improvement logs, not just static training descriptions. It also requires boundaries: some systems should be allowed to refine answers at inference time but not update persistent memory; others may update memory but not weights; high-risk systems may require human approval before any persistent update.

3.13 Recommended Reporting Checklist

To make results comparable, papers on AI self-improvement should report at least ten items. First, state the update target: output, memory, dataset, parameters, tool policy, or code. Second, state the feedback source and whether it is human, AI, programmatic, environmental, or mixed. Third, state the amount of additional inference compute used by the loop. Fourth, include compute-matched baselines such as best-of-N sampling. Fifth, report both average gains and per-task regressions. Sixth, describe filtering and selection criteria for self-generated data. Seventh, disclose whether benchmark tasks, answers, tests, or rationales were visible during data generation. Eighth, preserve ablations for each loop component. Ninth, report safety and robustness checks. Tenth, provide rollback or stopping criteria.

For inference-time methods, the report should include the maximum number of iterations, average number of iterations used, token cost, and examples where refinement harmed the output. For memory methods, it should include memory size, compression policy, retrieval policy, and examples of helpful and harmful memories. For training-time methods, it should include generated data volume, acceptance rate, reward distribution, number of training steps, KL to the reference model, and evaluation before and after each iteration. For RL agents, it should include trajectory

length, reward sparsity, advantage estimation, degeneracy filters, and environment generalization.

This checklist is intentionally demanding because self-improvement claims are easy to overstate. A polished loop diagram can hide the fact that the method depends on hidden human labels, leaked tests, large inference budgets, or reward-model overfitting. Transparent reporting allows the field to distinguish real self-evolution from benchmark engineering.

3.14 Worked Design Example: Building a Self-Improving Code Agent

A concrete design example makes the distinctions in this chapter operational. Suppose we want to build a code agent for programming contest tasks and small software issues. A minimal one-shot agent would read the problem and generate code. A self-improving agent should instead implement a layered loop. The first layer is Self-Refine: after generating a draft, the model checks the solution against the problem statement, constraints, edge cases, and complexity target. The second layer is Self-Debug: the code is executed against sample tests and generated tests, and the trace is fed back to a repair prompt. The third layer is Reflexion: if the task fails after the repair budget is exhausted, the agent writes a short memory about the failure mode. The fourth layer is training-time data collection: every generated candidate, test, error, repair, and final result is stored for offline filtering. The fifth layer is periodic training: verified successful repairs become supervised examples, failed-vs-successful pairs become preference data, and longer trajectories become RL data.

The resulting online loop is

$$c_0 \sim \pi_\theta(c \mid x, m), \tag{77}$$
$$q_0 = \mathrm{Critique}_\theta(x, c_0), \tag{78}$$
$$c_1 \sim \pi_\theta(c \mid x, c_0, q_0), \tag{79}$$
$$o_t = \mathrm{Run}(c_t, \mathcal{T}_{\mathrm{sample}} \cup \mathcal{T}_{\mathrm{gen}}), \tag{80}$$
$$c_{t+1} \sim \pi_\theta(c \mid x, c_t, o_t), \tag{81}$$
$$m' = \mathrm{Reflect}_\theta(x, c_{0:T}, o_{0:T}). \tag{82}$$

The offline loop is

$$\mathcal{D}_{\mathrm{train}} = \mathrm{Filter}\{(x, c_t, o_t, c_{t+1}, m') : \mathrm{HiddenPass}(c_{t+1}) = 1\}, \qquad \theta' = \mathrm{Train}(\theta, \mathcal{D}_{\mathrm{train}}). \tag{83}$$

This example combines all three update targets: the current code changes immediately, memory changes for future attempts, and parameters change after offline training.

The most important engineering choice in this example is test generation. If the agent only sees public samples, it will overfit. A stronger system asks the model to generate tests, asks a second role to attack the solution, and uses property-based fuzzing where possible. However, generated tests are themselves fallible. A generated test should not automatically define correctness; it should be treated as a hypothesis. If a generated test fails, the agent should inspect whether the test is valid before changing the code. This creates a nested self-improvement loop: improve the solution and improve the verifier. ReVeal's emphasis on co-evolving code generation and verification is a direct response to this need.

The memory layer should avoid storing problem-specific answers. A memory saying "for problem 1842 use dynamic programming" is not a general lesson and may leak benchmark information. A better reflection is "When constraints allow $O(n^2)$ but not $O(n^3)$, derive the transition before coding; the failed attempt used a triple loop and timed out." The memory should include scope conditions and should be retrieved only when the new task shares those conditions. If a future task

has small constraints, the same memory may be irrelevant. This is why Reflexion-style systems need retrieval and memory typing, not only append-only text.

For offline training, the data should be divided by role. Code drafts train generation. Error-to-fix pairs train repair. Test-generation traces train verification. Reflections train memory writing or retrieval. Preference pairs train ranking: a passing minimal solution should be preferred to a failing elegant solution, while among passing solutions the system may prefer clarity or efficiency. Mixing all these into one undifferentiated supervised corpus can blur capabilities. A model trained to write final answers may not learn to write good tests; a model trained on verbose reflections may become verbose in final code. Role-specific data preserves specialization.

Evaluation of this code agent should include at least four baselines. The first is one-shot generation with the same model. The second is best-of-N sampling with the same total token budget. The third is execution repair without verbal critique. The fourth is execution repair with generated tests but without persistent memory. Only if the full system beats these baselines can we attribute improvement to the complete self-evolution loop. The evaluation should also report hidden-test pass rate, public-sample pass rate, generated-test validity, average number of repairs, timeouts, and regressions where repair changed a passing solution into a failing one.

This example generalizes beyond code. In a research assistant, execution is replaced by retrieval and citation checking. In a web agent, tests are replaced by environment success signals. In a theorem prover, execution is replaced by proof checking. In a dialogue assistant, generated tests are replaced by constitutional or policy checks. The architecture remains the same: propose, observe, select, update, and audit.

3.15 Worked Design Example: Building a Self-Improving Reasoning Model

A reasoning model requires a different design because final answers are easier to check than intermediate reasoning. Consider a GSM8K-like arithmetic setting. A simple STaR loop generates rationales, keeps those whose final answers are correct, and fine-tunes. A more robust loop adds three safeguards. First, it perturbs the problem statement with equivalent numbers or paraphrases to test whether the rationale is stable. Second, it checks intermediate arithmetic with a calculator or symbolic tool. Third, it stores negative rationales and asks the model to explain their first invalid step. The training set then contains positive traces, repaired traces, and diagnostic contrasts.

The data format might be

$$(x, r^+, a), \qquad (x, r^-, e_{\text{first error}}, r^{\text{repair}}, a), \qquad (x, r^+, \tilde{x}, \tilde{r}^+, \tilde{a}), \tag{84}$$

where $\tilde{x}$ is a perturbation and $e_{\text{first error}}$ localizes the first wrong step. The model is trained not only to produce a correct rationale, but also to identify and repair faulty reasoning. This connects STaR to RISE: rationale bootstrapping supplies data, while multi-turn introspection trains the behavior of checking and revising.

A reasoning self-improvement loop should distinguish answer accuracy from reasoning faithfulness. If the model predicts the right answer but the rationale contains invalid arithmetic, the example should not be treated as fully positive. Conversely, if the reasoning is mostly correct but the final arithmetic slips, the example is valuable for repair training. A binary final-answer filter throws away this structure. Process labels are expensive, but tools can provide some of them. Calculators can check arithmetic; symbolic solvers can check algebra; theorem provers can check formal steps; unit tests can check executable reasoning in code-like tasks.

The training objective can combine supervised rationale likelihood, repair likelihood, and preference ranking:

$$\mathcal{L} = \mathcal{L}_{\text{SFT}}(r^+, a \mid x) + \lambda_1 \mathcal{L}_{\text{repair}}(r^{\text{repair}} \mid x, r^-, e) + \lambda_2 \mathcal{L}_{\text{pref}}(r^+ \succ r^- \mid x). \tag{85}$$

The preference term can use the DPO form in Equation 45, with correct or verified rationales as winners. If the model is later trained with RL, the reward should combine final correctness, process validity, and brevity to avoid unnecessarily long chains.

Evaluation should include GSM8K-style exact match, but also perturbation consistency, first-error localization accuracy, and calibration. A model that answers correctly only when numbers appear in familiar positions has not learned robust reasoning. A model that can locate its own first error is more useful for self-improvement because it can produce actionable repair signals. RISE's core insight is that revision behavior can be trained; the worked design here shows one way to create data for that behavior.

3.16 Choosing Between Prompting, Fine-Tuning, and Reinforcement Learning

A practical survey should also say when not to use a method. Prompt-only self-improvement is appropriate when the task distribution is fluid, the model is accessible only through an API, the cost of errors is moderate, and external feedback can be obtained at inference time. Self-Refine and Reflexion are especially useful in product prototypes because they require little infrastructure. They are less appropriate when latency is strict, output consistency matters, or the same task distribution recurs millions of times. In those cases, repeated inference is wasteful and training-time amortization becomes attractive.

Supervised self-training is appropriate when high-quality generated positives can be identified. STaR is appropriate when final answers are available and rationales are missing. ReST is appropriate when a reward model or metric can score many candidates cheaply. SelfEvolve-style repair data is appropriate when execution traces can identify successful fixes. Supervised self-training is less appropriate when positives are rare, filters are noisy, or generated data lacks diversity. A model trained on its own narrow successes may become more confident on the same narrow slice while failing elsewhere.

Preference optimization is appropriate when relative quality is easier to judge than absolute correctness. DPO, SPIN, and Constitutional AI all exploit contrasts. Preference methods are useful for dialogue, helpfulness, harmlessness, style, and tasks where multiple answers may be acceptable. They are less direct for tasks with objective hidden correctness unless paired with verifiers. A preference model can learn that an answer looks good without being correct. Therefore preference optimization should be supplemented with factuality and execution checks when possible.

Reinforcement learning is appropriate when the system must learn a policy over multi-turn actions, tool calls, or revision decisions. RISE, RAGEN, and ReVeal are examples. RL is powerful because it can optimize delayed rewards and non-differentiable outcomes, but it is expensive and unstable. It requires careful reward design, KL control, exploration management, and degeneracy diagnostics. RL should not be the first tool for a simple formatting or one-shot reasoning problem. It becomes necessary when the behavior to be learned is sequential and cannot be reduced to independent supervised examples.

A useful decision rule is: start with the cheapest loop that uses the strongest available feedback. If a prompt loop plus verifier solves the problem, do not train. If the same verified repairs recur frequently, collect them for supervised training. If preferences matter and examples can be compared, use preference optimization. If the system must learn long-horizon action policies, use RL with strong monitoring. This staged approach reduces risk and provides data for each stronger method.

3.17 Conclusion

Self-improvement methods form the engine of AI self-evolution. Self-Refine shows that a single model can improve outputs through generate–critique–refine loops. Reflexion shows that natural-language reflections can serve as non-parametric memory and approximate verbal reinforcement learning. Self-Debug shows that execution feedback can ground repair. SPIN shows that a model can use its own previous responses as self-play negatives to improve a supervised seed. STaR shows that rationales can be bootstrapped from answer correctness. ReST-EM shows how generated candidates and reward filtering create a growing-batch training loop. Constitutional AI shows how a compact human constitution can be expanded into critiques, revisions, and AI feedback for alignment. RISE, RAGEN, Absolute Zero, and ReVeal show the frontier: multi-turn RL, trajectory-level credit assignment, self-generated curricula, and explicit self-verification.

The field is converging on a common recipe: generate candidates, obtain feedback, filter or revise, update state, and evaluate under a guardrail. The open questions are how to make feedback reliable, how to prevent degenerate loops, how to transfer lessons across tasks without memory poisoning, how to train multi-turn agents without reasoning collapse, and how to evaluate improvement without compute confounds. The benchmark evidence is strongest in code and math because verifiers are strongest there. Broader self-evolution will require broader verification.

4 Evolutionary Code and Algorithm Discovery

The recent literature on AI self-evolution contains a striking convergence: the object being improved is no longer only a model checkpoint, a prompt, or a policy, but an executable artifact that can be inspected, modified, evaluated, archived, and selected. In this chapter, *evolutionary code and algorithm discovery* denotes this family of methods. The term covers prompt optimization systems such as OPRO, program-search systems such as FunSearch, codebase-level evolutionary agents such as AlphaEvolve and OpenEvolve, and agent-level systems such as ADAS, the Darwin Gödel Machine, Gödel Agent, Agent Symbolic Learning, Absolute Zero, and SelfEvolve. These systems differ in scale and substrate, yet they instantiate the same closed loop: a population of candidate artifacts is represented in language or code; a generator proposes variants; an evaluator assigns scores; a memory or archive stores outcomes; and a selection rule decides what information should condition the next proposal.

This loop is central to self-evolution because it gives large language models (LLMs) a structured way to improve systems without changing the base model weights. Standard LLM inference is episodic: the model receives a context, emits an answer, and forgets the attempt unless a user or external memory preserves it. Evolutionary code discovery makes the episode persistent. A failed program becomes a negative example, a successful heuristic becomes a parent, an error trace becomes mutation guidance, and a benchmark score becomes a scalar reward. Over many iterations, the surrounding system can accumulate knowledge even when the LLM itself remains frozen. This is why code evolution occupies a bridge position between prompt engineering, AutoML, reinforcement learning with verifiable rewards, and open-ended evolutionary computation.

The chapter follows six questions. First, how did OPRO show that an LLM can operate as an optimizer through natural language history rather than through gradients? Second, how can an LLM be treated as an evolutionary operator that performs mutation, crossover, and selection-aware recombination over programs? Third, why did FunSearch make program evolution credible as a route to mathematical discovery, and how should its achievements be distinguished from the later AlphaEvolve result on matrix multiplication? Fourth, what do open-source systems expose about engineering requirements, licenses, and reproducibility? Fifth, how do agent self-evolution

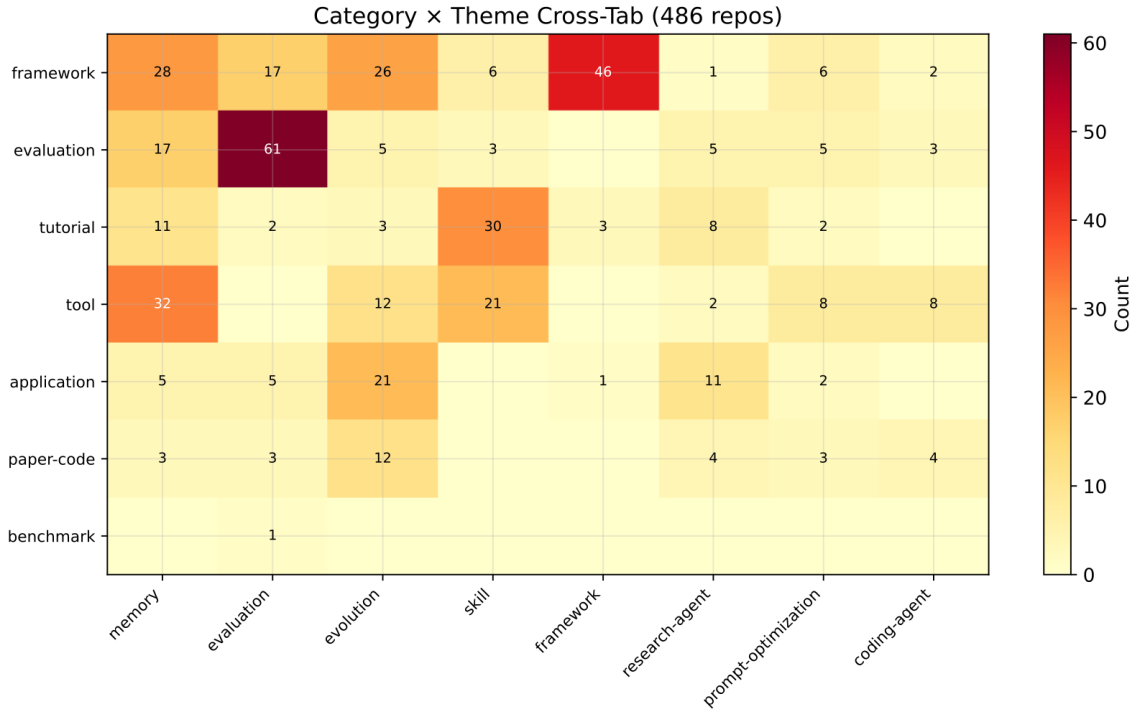

Figure 14: Category–theme heatmap from the survey corpus. The figure shows why system-level evidence cannot be read from repository labels alone: coding agents, memory substrates, benchmarks, prompt optimizers, and framework projects overlap across multiple self-evolution themes.

systems extend the same loop from individual functions to whole agent architectures? Sixth, what benchmark evidence is currently available, and what are its methodological limitations?

A useful abstraction is to separate the *inner solver* from the *outer evolutionary process*. The inner solver may be a prompt, a Python function, a scheduling heuristic, a reasoning trajectory, or a multi-agent workflow. The outer process owns candidate generation, evaluation, archive maintenance, and search policy. In classical evolutionary computation, mutation and crossover are usually syntactic or numeric operators. In LLM-driven evolution, these operators become semantic: a model can read a parent program, infer its intent, identify a bottleneck, import a new algorithmic idea, and write a child that is not a small edit in token space but may still be a meaningful descendant in design space. This semantic variation is the key advantage and the key danger. It allows discontinuous jumps to powerful ideas, but it also makes safety, credit assignment, and reproducibility harder.

The strongest systems therefore combine neural creativity with hard verification. OPRO evaluates candidate instructions on held-out tasks. FunSearch executes programs in a sandbox and admits only candidates that pass syntactic and semantic filters. AlphaEvolve and its open-source descendants rely on automated evaluators for objective scores. SE-Agent evaluates evolved reasoning trajectories on SWE-bench Verified. Absolute Zero uses a code executor to verify self-generated tasks and solutions. The common lesson is that self-evolution is not merely "asking the model to improve itself." It is the design of an evaluation ecology in which generated artifacts can safely compete, fail, and leave traces that shape future generations.

4.1 LLM as Optimizer: OPRO

OPRO, or Optimization by PROmpting, introduced a clean formulation of the LLM-as-optimizer paradigm [66]. The central idea is deceptively simple. Instead of hand-designing a gradient estimator, a Bayesian surrogate, or a genetic operator, the system expresses the optimization problem in natural language. It gives the optimizer LLM a history of candidate solutions and their scores, asks for new candidates that should perform better, evaluates those candidates with a scorer, and repeats the process. When the candidate is an instruction for a downstream LLM, OPRO becomes prompt optimization. When the candidate is a vector of numerical variables or a route for a combinatorial problem, the same outer loop becomes a generic black-box optimizer.

Let $\mathcal{X}$ be a candidate space and let $f : \mathcal{X} \to \mathbb{R}$ be a black-box objective. At iteration t, OPRO maintains a history

$$\mathcal{H}_t = \{(x_i, f(x_i))\}_{i=1}^{n_t}, \tag{86}$$

where x_i may be a natural-language instruction, a parameter tuple, or a combinatorial object serialized as text. A meta-prompt constructor M converts the history, task description d, constraints c, and optional examples e into a prompt:

$$m_t = M(\mathcal{H}_t, d, c, e). \tag{87}$$

The optimizer LLM with parameters θ samples one or more new candidates,

$$x_{t+1}^{(j)} \sim p_\theta(x \mid m_t, \tau_{\text{opt}}), \qquad j = 1, \dots, k, \tag{88}$$

where τ_{opt} is typically high enough to maintain diversity. After filtering duplicates or invalid outputs, the candidates are scored,

$$s_{t+1}^{(j)} = f\big(x_{t+1}^{(j)}\big), \tag{89}$$

and the history is updated by appending the new pairs and optionally truncating to the most informative top-scoring examples:

$$\mathcal{H}_{t+1} = \text{Truncate}\left(\mathcal{H}_t \cup \{(x_{t+1}^{(j)}, s_{t+1}^{(j)})\}_{j=1}^{k}\right). \tag{90}$$

The final output is usually $x^* = \arg\max_{(x,s)\in\mathcal{H}_T} s$ for maximization, or $\arg\min$ for loss minimization.

This formulation is important because the optimizer is not trained on gradients of f. It is given a textual trace of empirical evidence and asked to infer a better move. The prompt therefore becomes an optimizer interface. If the history is ordered, bucketed, or annotated, the model can exploit ordinal relationships. If the prompt includes high-frequency mistakes, the model can target weak regions. If the prompt includes constraints, the model can internalize feasibility. The optimization algorithm is partly encoded in the surrounding program and partly encoded in the model's pretraining distribution over explanations, heuristics, and problem-solving patterns.

In prompt optimization, OPRO uses a two-layer architecture. The outer layer is the meta-prompt supplied to the optimizer LLM. It contains prior instructions and their scores, a statement that higher scores are better, and a request to generate a new instruction in a parseable format. The inner layer is the task prompt supplied to the scorer LLM. For each candidate instruction, the scorer LLM solves a collection of training examples, and the system computes an accuracy or task-specific score. The local materials report four instruction placement modes: before the question, at the beginning of the question, at the end of the question, and at the beginning of the answer. This matters because prompt optimization is not only the search for words; it is also the search for where those words interact with the model's generation policy.

The implementation details illustrate a general engineering pattern for self-evolution. The optimizer temperature is high, encouraging diverse candidate generation, while the scorer temperature is low, making evaluation more deterministic. OPRO's default instruction-optimization loop can run for about two hundred search steps and generate multiple candidate instructions per step. The meta-prompt may retain a bounded number of top instructions rather than the entire history, both to fit within context and to emphasize successful patterns. Scores can be discretized into buckets so that the optimizer does not overfit tiny score differences that may be noise. Candidate extraction uses explicit delimiters such as `<INS>` and `</INS>` for one model family and bracket conventions for another; invalid, duplicate, overly long, or obviously malformed instructions are filtered before scoring.

The mathematical significance of OPRO is that it turns optimization into conditional generation over a history of evaluations. The search policy is not a hand-coded evolutionary rule, yet it can emulate evolutionary behavior. A high-scoring instruction acts as an elite. A low-scoring instruction acts as a negative example. A newly sampled instruction may mutate phrasing, recombine ideas from several parents, or introduce a new heuristic learned from pretraining. Selection is explicit in the history truncation and ranking. Thus OPRO is a minimal bridge from prompt engineering to evolutionary search: it lacks a population database, island model, or formal quality-diversity archive, but it already contains the essential ingredients of generate–evaluate–remember.

For reasoning tasks, OPRO's optimizer can rediscover familiar prompt patterns such as step-by-step reasoning, careful checking, or decomposition. The point is not that every discovered instruction is semantically surprising. The point is that the system can discover, rank, and reuse such instructions automatically for a specific task distribution. This task adaptation is especially relevant in AI self-evolution because a deployed agent often faces a distribution that differs from the model's original training mixture. An OPRO-like outer loop can tune the agent's operating prompt to the environment without requiring gradient updates.

OPRO also demonstrates that the same language interface can optimize non-linguistic variables. In the linear-regression example, a candidate may be represented as a textual pair such as $w = 15, b = 14$, and the objective is a squared error,

$$f(w, b) = \|y - (Xw + b)\|_2^2. \tag{91}$$

In a traveling-salesperson example, a candidate is a route serialized as a trace and scored by tour length. The optimizer LLM does not receive derivatives with respect to w, b, or the route. It receives a record of previous values and objective scores. This suggests a broad design principle: when a domain can be serialized into a representation that an LLM can interpret and when candidate quality can be automatically evaluated, an LLM can serve as a black-box proposal distribution over improvements.

However, OPRO also exposes limitations that persist in later systems. First, the optimizer can be misled by noisy evaluations. If the scorer LLM is stochastic or the training set is small, the history may reward spurious instructions. Second, the representation of history is a bottleneck. The model can only condition on what fits into context, so summarization and selection policies become part of the optimizer. Third, OPRO's natural-language mutations have no formal guarantee of monotonic improvement. Even when the prompt says "generate a better instruction," the sample may be worse, redundant, or invalid. Fourth, prompt optimization can overfit to the training examples used for scoring. Validation intervals and held-out sets are therefore essential.

For a survey of self-evolution, the main contribution of OPRO is conceptual. It shows that a language model can be wrapped as a general-purpose optimizer if the search state is expressed in language and if a reliable score is available. Later systems replace instructions with functions,

agents, trajectories, and codebases, but they preserve the same core equation: new artifacts are sampled from a model conditioned on the archive of old artifacts and their measured consequences. OPRO is thus the first step in a progression from "LLM writes an answer" to "LLM optimizes the process that writes answers."

4.2 LLM as Evolutionary Operator: Mutation, Crossover, and Selection

Once an LLM is used to generate candidates from a history of scored artifacts, it is natural to interpret it as an evolutionary operator. Classical evolutionary algorithms usually define mutation as a random perturbation, crossover as a recombination of parent genomes, and selection as a rule that biases reproduction toward higher-fitness individuals. LLM-driven evolution changes the representation and the operators. The genome may be a program, a prompt, a workflow, or a reasoning trajectory. Mutation may be a semantic edit proposed after reading an error trace. Crossover may be a synthesis of two algorithmic ideas described in code comments. Selection may be implemented by an archive that exposes successful parents to the next LLM call while hiding or down-weighting failures.

A general LLM-evolution loop can be written as follows. Let A_t be an archive of artifacts a with fitness $F(a)$ and optional behavior descriptor $b(a)$. Parent selection chooses one or more parents,

$$(a_{p_1}, \ldots, a_{p_m}) \sim q_t(a \mid A_t), \tag{92}$$

where q_t may depend on fitness, novelty, age, or niche coverage. A prompt constructor serializes parent code, evaluation traces, failures, and instructions into z_t. The LLM samples a child,

$$a_c \sim p_\theta(a \mid z_t, \tau, \text{mode}), \tag{93}$$

where *mode* may request mutation, recombination, repair, simplification, or targeted optimization. The evaluator executes or judges a_c, producing $F(a_c)$ and $b(a_c)$. The archive update rule then decides whether to insert, discard, or quarantine the child:

$$A_{t+1} = U(A_t, a_c, F(a_c), b(a_c)). \tag{94}$$

This formalism covers OPRO, FunSearch, AlphaEvolve, OpenEvolve, CodeEvolve, and agent-level variants. The differences lie in q_t, the prompt constructor, the evaluator, and U.

LLM mutation is more powerful than token-level mutation because it can preserve intent while changing implementation. For example, a model may replace a quadratic loop with a heap, introduce memoization, rewrite a heuristic priority function, or add a guard that prevents a known failure mode. These changes may be large in edit distance but small in conceptual distance. Conversely, a tiny textual edit may be semantically catastrophic. This mismatch means that evolutionary distance in LLM systems should not be measured only by tokens or lines. It should be measured by behavioral descriptors, test signatures, resource profiles, and genealogical relationships.

LLM crossover is similarly semantic. A classical crossover operator splices substrings or subtrees. A model-based operator can read two parents and produce a coherent child that combines the first parent's data structure with the second parent's scoring rule. It can also reject a naive combination if the pieces are incompatible. Systems such as SE-Agent make this explicit at the trajectory level: recombination takes useful fragments from multiple reasoning paths and constructs a new path intended to inherit their strengths. Open-source code evolution systems present multiple "inspiration" programs to the model, which gives the model context for recombination without forcing a mechanical splice.

Selection remains partly non-neural and should remain so. The reason is simple: self-evolution needs a grounding signal outside the model's preference for plausible text. If the model both proposes and declares success without external verification, the loop can collapse into self-confirmation. Selection therefore depends on unit tests, benchmarks, mathematical objective functions, execution traces, or separately controlled judges. Even when an LLM judge is used, robust systems prefer low-temperature evaluation, multiple samples, calibration sets, or verifiable subcomponents. The evaluator is the anchor that prevents an evolutionary language loop from becoming mere rhetoric.

AlphaEvolve is the most prominent large-scale example of the LLM-as-evolutionary-operator paradigm [40]. It is described as a Gemini-powered coding agent for scientific and algorithmic discovery. Its architecture combines a fast model for breadth, Gemini Flash, with a stronger model for depth, Gemini Pro. The practical intuition is that exploration and exploitation need different model economics. Breadth requires many candidate ideas at acceptable cost and latency. Depth requires more expensive reasoning for difficult improvements, abstraction, or repair. A dual-model architecture therefore creates a search policy over model capabilities, not only over program candidates.

AlphaEvolve extends the FunSearch style from single evolved functions to broader code artifacts. It uses automated evaluators and an evolutionary database. The public descriptions emphasize quality-diversity search, especially MAP-Elites, rather than a single best-so-far chain. Quality-diversity methods aim to illuminate a space of solutions by preserving high-quality artifacts across diverse behavioral niches. In algorithm discovery, this matters because the first solution that reaches a local optimum may not be the stepping stone to the next breakthrough. A slower or less elegant candidate may contain a structural idea that becomes valuable after later recombination.

MAP-Elites maintains an archive indexed by behavior descriptors. Let $\mathcal{B}$ be a discretized descriptor space and let $b(a) \in \mathcal{B}$ map an artifact to a cell. The archive stores at most one elite per cell:

$$E[c] = \arg \max_{a : b(a) = c} F(a). \tag{95}$$

When a child a_c is evaluated, it is inserted if its cell is empty or if it improves the elite already in that cell:

$$E[b(a_c)] \leftarrow \begin{cases} a_c, & E[b(a_c)] = \varnothing, \\ a_c, & F(a_c) > F(E[b(a_c)]), \\ E[b(a_c)], & \text{otherwise.} \end{cases} \tag{96}$$

The archive is therefore not a leaderboard but a map of high-performing diversity.

The algorithm box abstracts AlphaEvolve-like systems while also describing open implementations. Several design choices are hidden inside the prompt constructor. The prompt may include parents sorted by score, a concise failure analysis, a list of forbidden regressions, or a request for a patch rather than a full file. It may ask the model to preserve function signatures, stay within an evolution region, or add only deterministic code. It may also use model routing: a fast model proposes many rough edits, while a stronger model repairs, explains, or intensifies promising branches. In this sense, the prompt is not merely an input string; it is an operator specification.

AlphaEvolve's reported achievements illustrate why this architecture attracted attention. The local materials report that it found a 48-multiplication algorithm for 4×4 complex matrix multiplication, described as the first improvement over the Strassen-era baseline in 56 years. They also report that across more than fifty mathematical problems, AlphaEvolve rediscovered state-of-the-art results for about 75% and found better solutions for about 20%. In infrastructure applications, the same source reports improvements such as recovering 0.7% of worldwide compute in data-center scheduling, simplifying hardware accelerator design, and speeding LLM training kernels by 23% or

Figure 15: LLM-guided MAP-Elites for evolutionary code discovery

Input: Seed programs P_0, evaluator Eval, descriptor function b, LLM ensemble $\mathcal{M}$, budget T

1. Initialize archive E with empty cells

for all $p \in P_0$ **do**

2. $(F(p), b(p), r(p)) \leftarrow \text{Eval}(p)$
3. Insert p into E if it is elite for cell $b(p)$

for $t = 1$ to T **do**

4. Select one or more elites P_t from E using fitness, novelty, age, and coverage weights
5. Construct an evolution prompt with parent code, scores, descriptors, failures, and instructions
6. Choose model $m_t \in \mathcal{M}$, e.g., fast model for breadth or strong model for depth
7. Sample candidate edits or complete child programs $C_t \sim m_t(\cdot)$

for all $c \in C_t$ **do**

8. Parse, sanitize, and sandbox c
9. $(F(c), b(c), r(c)) \leftarrow \text{Eval}(c)$

if c is valid and improves the elite in cell $b(c)$ **then**

10. $E[b(c)] \leftarrow c$

else

11. Store failure trace $r(c)$ for diagnostics or future prompts **return** archive E and the global best elite $\arg\max_{c \in E} F(c)$

32% in particular attention-related settings. These numbers are domain-specific and should be read as examples of evaluator-grounded optimization rather than universal guarantees.

The dual-model architecture is a major systems lesson. In evolutionary search, the marginal value of a model call depends on where it is used. Early generations benefit from diversity and cheap exploration. Later generations may require deeper reasoning to escape plateaus. A system can allocate model budgets analogously to mutation rates or temperature schedules in classical evolution. A cheap model can populate the archive with many variants; an expensive model can be invoked when a niche is promising, when a failure trace suggests a precise repair, or when a parent has high novelty. Self-evolution therefore becomes a resource-allocation problem over artifacts, evaluators, and models.

Quality-diversity also reframes success. A single best program is useful for deployment, but an archive of elites is useful for continued self-improvement. If the environment changes, a non-best elite may become the best starting point. If a later prompt needs examples of different trade-offs, the archive supplies them. If human researchers want insight, the map reveals which behavioral

niches are easy, hard, empty, or overrepresented. This is particularly relevant for AI self-evolution because open-ended improvement depends on preserving stepping stones that do not immediately maximize the current score.

The main risks are evaluator overfitting and specification gaming. If the evaluator measures speed on a narrow input distribution, evolution may discover brittle shortcuts. If tests are visible or inferable, a model may produce code that passes tests without solving the intended task. If performance is measured on a shared machine without careful isolation, noise may drive selection. Systems therefore need hidden tests, randomized inputs, resource controls, multiple metrics, and audit trails. They also need a policy for failed candidates: failures are valuable search information, but including too many raw failures in prompts can pollute context and encourage degenerate patterns. A good archive stores both successes and summarized failure modes.

4.3 Program Evolution and Mathematical Discovery: FunSearch

FunSearch is the canonical demonstration that LLM-guided program evolution can produce mathematical discoveries rather than only better benchmark prompts [47]. The system combines a pretrained LLM with an evaluator and a population database. Instead of searching directly over solutions, it searches over programs that generate or prioritize solutions. This distinction is crucial. A raw combinatorial object may be too large, brittle, or unstructured for efficient search. A program can encode a reusable heuristic, a constructive rule, or a priority function that generalizes across instance sizes. The LLM's knowledge of code and mathematics becomes useful because the candidate artifact is executable source text.

The FunSearch pipeline has three central components: a sampler, an evaluator, and a programs database. The database produces prompts containing prior program variants. The sampler asks the LLM to complete a new version of the function marked for evolution. The evaluator parses and executes the candidate, computes a score, and registers successful programs back into the database. Users mark the problem with two decorators: one function is the run/evaluation entry point and another function is the function to evolve. The evolved function may be a priority function for greedily constructing a mathematical object or a heuristic used inside a solver.

The local implementation analysis reports that FunSearch uses a population structure with islands, clusters, and programs. Islands are independent subpopulations. Clusters group programs with the same score signature across test cases. Within a cluster, shorter programs are preferred, applying an Occam-style bias toward concise heuristics. Periodic island resets remove weaker islands and restart them from strong founders, which helps avoid premature convergence while preserving successful discoveries. This is a classical evolutionary idea adapted to LLM-generated code.

The prompt design is one of FunSearch's most important contributions. A prompt may show a chain of versions such as `priority_v0`, `priority_v1`, and an empty `priority_v2` with a docstring saying it should improve the previous version. Earlier versions appear with imports and helper functions, allowing the LLM to see the evolving code context. Versions can be sorted so that stronger examples appear later, exploiting recency effects in transformer context. The prompt does not merely request a solution; it stages an evolutionary lineage and invites the model to continue it.

FunSearch's evaluator is deliberately strict. It checks that the generated code parses, runs within a sandbox and timeout, does not call ancestor functions directly, and returns a numeric score. Candidate code may be trimmed with AST-based recovery when the LLM emits incomplete text, but semantic admission depends on execution. The evaluator also records a signature: a vector of scores across test instances. This signature is used for clustering, so two programs with the same aggregate score but different behavior can be distinguished if their instance-level patterns

Figure 16: FunSearch-style program evolution

Input: Initial program with `@run` evaluator and `@evolve` target; LLM; programs database D; evaluator suite E_{eval}

1. Initialize islands in D with the seed program

while budget remains **do**

2. Sample an island and construct a prompt from high-scoring program versions
3. Ask the LLM to complete the next evolved function version
4. Parse candidate code; trim or reject malformed bodies
5. Execute candidate in sandbox on benchmark instances

if candidate runs, avoids ancestor calls, and returns numeric scores **then**

6. Compute signature vector and reduced score
7. Register program in the corresponding island and cluster

else

8. Discard or log the failure

if reset period elapsed **then**

9. Reinitialize weak islands from founders sampled among strong islands **return** best programs and their mathematical constructions

differ.

A simplified FunSearch loop can be described as:

FunSearch's headline achievements include new constructions for the cap set problem and better heuristics for bin packing. It also discovered solutions in related combinatorial settings such as admissible sets and cyclic graph independent sets. These results are scientifically significant because the output programs can be inspected. A human can read the evolved priority function, reason about why it works, and potentially translate it into mathematical insight. This inspectability differentiates program search from black-box neural prediction.

It is important to correct a common attribution confusion. The first improvement over the Strassen-era result for 4×4 complex matrix multiplication is associated with AlphaEvolve in the local AlphaEvolve materials, not with the original FunSearch paper. FunSearch is the predecessor that established LLM-plus-evaluator program evolution for mathematical discovery. AlphaEvolve later extended the approach to larger code artifacts and reported the matrix-multiplication improvement. For historical accuracy, Table 7 lists FunSearch's discoveries separately from the AlphaEvolve successor result.

The methodological lesson is that mathematical discovery becomes possible when three conditions align. First, the domain must admit a compact executable representation. Second, the evaluator must be cheaper than the value of the discovery and reliable enough to guide search.

Table 7: Representative program-evolution discoveries. The matrix-multiplication row is included because it is often discussed alongside FunSearch, but the reported result belongs to AlphaEvolve rather than the original FunSearch system.

System	Domain	Reported result	Interpretation
FunSearch	Cap set constructions	Better construction than previously known in the studied setting	Demonstrates that evolved programs can yield new mathematical objects rather than merely solve training instances.
FunSearch	Bin packing	Improved heuristic rules	Shows that program evolution can discover practical algorithms with interpretable logic.
FunSearch	Cyclic graphs and related combinatorics	New constructive programs in selected problems	Supports the claim that function-space search generalizes across mathematical domains with evaluators.
AlphaEvolve	4×4 complex matrix multiplication	48 multiplications, reported as first improvement over Strassen-era baseline in 56 years	Successor system extends program evolution to broader algorithmic code and quality-diversity search.
AlphaEvolve	50+ mathematical problems	Recovered SOTA for roughly 75% and improved roughly 20% in local materials	Indicates breadth when evaluator design is available, but results are problem- and budget-dependent.

Third, the search system must preserve diversity because mathematical breakthroughs often require stepping stones. FunSearch satisfies these conditions for several combinatorial problems. It does not solve domains where evaluation is ambiguous, expensive, or dependent on human taste.

FunSearch also reveals why program evolution is different from ordinary code generation. A code-generation benchmark usually asks the model to solve a problem once. Program evolution asks the model to participate in an iterative population process. The model is not judged only by the first sample; it is judged by whether its samples can be selected, recombined through context, and improved over generations. A mediocre sample may still be valuable if it introduces a structural motif that later variants exploit. The unit of scientific productivity is therefore the search process, not an individual completion.

For AI self-evolution, FunSearch provides a template that can be lifted from functions to agents. Replace a priority function with a planning policy, a tool-selection rule, or a memory-update function. Replace the mathematical evaluator with a task benchmark. Replace island clusters by agent behavior signatures. The same loop becomes agent evolution. The difficulty increases because agents have larger state spaces, more sources of nondeterminism, and more opportunities to exploit evaluator loopholes. Nevertheless, the FunSearch architecture remains a foundational pattern: evolve executable artifacts, evaluate them automatically, preserve a population, and expose interpretable lineages.

4.4 Open-source Implementations: OpenEvolve, FunSearch, CodeEvolve, and SE-Agent

Open-source implementations matter because they reveal the engineering surface hidden behind research papers. A paper may state that an evaluator executes programs, but a reusable system must decide how to isolate file systems, limit memory, parse model outputs, checkpoint populations, route among model providers, recover from failed subprocesses, and present results to users. It must also choose a license and a reproducibility posture. In self-evolution, these engineering details are not incidental; they define what kinds of artifacts can safely evolve.

FunSearch's released code is best viewed as a research framework. It exposes the sampler, evaluator, program database, decorators, AST utilities, and island model, but it leaves the LLM interface and secure execution environment as components that users must implement or adapt. This is appropriate for a Nature paper whose purpose is to demonstrate a scientific method. The code makes the algorithm intelligible and auditable, but it is not a turnkey platform for arbitrary codebase evolution.

OpenEvolve presents itself as an open-source implementation inspired by AlphaEvolve. The current repository describes it as an evolutionary coding agent that turns LLMs into autonomous code optimizers, supports Python library usage, works with OpenAI-compatible providers, and includes MAP-Elites quality-diversity, island-based populations, an LLM ensemble, and an artifact side channel for error feedback. Its README reports examples such as circle packing, adaptive sorting, filter design, prompt evolution, and GPU-related optimization, and the repository lists an Apache-2.0 license. This positions OpenEvolve as a practical bridge between research-grade program evolution and user-facing automated optimization.

CodeEvolve is another open-source evolutionary coding agent. Its arXiv abstract describes an islands-based genetic algorithm with modular LLM orchestration, execution feedback, task-specific metrics, context-aware recombination, adaptive meta-prompting, and targeted refinement. The repository README describes Apache-2.0 licensing, optional MAP-Elites integration, distributed islands, prompt populations, SEARCH/REPLACE diff application, resource-limited sandboxed evaluation, and support for open-weight and proprietary models through OpenAI-compatible APIs. CodeEvolve is therefore especially useful as an engineering reference for multi-island orchestration, exploration–exploitation scheduling, and structured patch generation.

SE-Agent is different in substrate. It evolves reasoning trajectories for multi-step LLM code agents rather than directly evolving only source functions. Its repository describes three operations: Revision, Recombination, and Refinement. Revision analyzes failed trajectories and generates alternative strategies. Recombination synthesizes new trajectories from strengths of multiple paths. Refinement removes redundancies and risk patterns from promising paths. The repository reports 80% Top-1 performance on SWE-bench Verified and lists an MIT license. This makes SE-Agent a key example of evolutionary ideas applied to agent traces and problem-solving procedures, not only to code artifacts.

The comparison highlights three axes. The first axis is artifact granularity. FunSearch evolves one function; OpenEvolve and CodeEvolve can evolve larger code regions; SE-Agent evolves trajectories; AlphaEvolve targets broader algorithmic code. Larger granularity increases expressive power but also increases failure modes. A single function can be sandboxed and scored relatively cleanly. A codebase-level agent may need dependency installation, compilation, integration tests, performance profiling, and security isolation.

The second axis is diversity management. FunSearch uses islands and signature clusters. OpenEvolve and CodeEvolve expose MAP-Elites or island-based populations. SE-Agent uses trajectory pools and recombination. These mechanisms all address the same problem: greedy best-only search loses stepping stones. In self-evolving systems, diversity is not decorative. It is the memory of alternative hypotheses about how to solve the task.

The third axis is evaluation. FunSearch and AlphaEvolve are strongest when the evaluator is a crisp objective function. OpenEvolve and CodeEvolve generalize by letting users define evaluators, which increases applicability but shifts responsibility to the practitioner. SE-Agent's evaluator is a software-engineering benchmark where success depends on resolving real repository issues. This is more realistic but also more expensive and harder to reproduce. The more open-ended the artifact, the more evaluation becomes a systems problem.

Licensing also shapes adoption. Apache-2.0 code in FunSearch, OpenEvolve, and CodeEvolve

Table 8: Comparison of representative open and semi-open evolutionary code or trajectory systems. Public repository details can change; the table records the evidence available in the local research materials and web verification performed for this draft.

System	Evolved artifact	Search and evaluation mechanism	Status	Best use case
FunSearch	Single marked Python function inside a problem scaffold	Island model, program clusters by score signature, LLM completion of versioned functions, and sandboxed execution for cap set, bin packing, admissible/cyclic graph tasks.	Apache-2.0 code; CC-BY materials in local notes.	Interpretable mathematical program discovery.
OpenEvolve	Functions, programs, prompts, and optimization targets	MAP-Elites quality-diversity, islands, LLM ensemble, artifact feedback, and user-defined evaluators for circle packing, prompt optimization, sorting, and scientific computing.	Apache-2.0 repository.	Practical AlphaEvolve-style experimentation with provider flexibility.
CodeEvolve	Code regions and algorithmic solutions	Islands-based genetic algorithm, context-aware recombination, adaptive meta-prompting, optional MAP-Elites, and sandboxed task-specific metrics.	Apache-2.0 repository.	Reproducible multi-island code evolution and ablation studies.
SE-Agent	Multi-step reasoning trajectories for code agents	Revision, recombination, and refinement across trajectories, evaluated on SWE-bench Verified with repository-reported 80% Top-1.	MIT repository.	Improving agent reasoning procedures over real software issues.
AlphaEvolve	Broader algorithmic code artifacts	Gemini Flash/Pro dual-model evolution with a quality-diversity archive for math problems, matrix multiplication, data-center scheduling, chip design, and LLM training kernels.	Not generally public in local notes.	Large-scale evaluator-rich algorithm discovery.

supports permissive use with patent grants, while SE-Agent's MIT license is similarly permissive but shorter. AlphaEvolve, by contrast, is described in papers and reports but is not generally available as a full public system in the local notes. This means that open-source successors are not merely clones; they are the artifacts through which the community tests which parts of the AlphaEvolve/FunSearch paradigm are reproducible outside a large industrial lab.

A practical takeaway is that open-source evolutionary systems need robust defaults. Users should not have to design from scratch how to checkpoint an archive, how to summarize failure traces, how to separate training and validation evaluations, or how to prevent an evolved program from reading hidden answers. The next generation of frameworks should treat these as first-class concerns. Configuration should expose model routing, evaluator budgets, archive policy, diversity descriptors, sandbox limits, and validation gates. Without these controls, self-evolution can appear to work in demos while failing under scientific replication.

4.5 Agent Self-Evolution Systems

The systems above evolve prompts, functions, code, or trajectories. Agent self-evolution raises the abstraction level further: the evolving artifact is an agent architecture, a workflow, a tool interface, a memory system, or the agent's own source code. This shift is conceptually important because agents are not single input–output functions. They perform multi-step reasoning, call tools, maintain state, interact with environments, and may coordinate with other agents. Evolution therefore has to optimize not only what the agent says, but how it senses, plans, acts, remembers, critiques, and modifies itself.

ADAS, or Automated Design of Agentic Systems, formalizes this direction [21]. Its Meta Agent Search algorithm uses a meta agent to write code for new agent architectures. The search space is the set of valid Python programs implementing agents, which is Turing-complete in principle. Each iteration, the meta agent reviews previously discovered designs and their performance, writes a new agent, evaluates it on benchmark tasks, and records the result. The output is the best design discovered in the archive. This is evolutionary code discovery applied to agent design rather than to mathematical heuristics.

ADAS's significance lies in the scope of its search space. Hand-designed agent frameworks usually expose a fixed menu: chain-of-thought, reflection, tool use, debate, tree search, or planning. A code-level design search can combine these motifs or invent new control flows. The local materials report that ADAS discovered agents outperforming state-of-the-art hand-designed agents on benchmarks such as ARC and Google Research Football, with transfer across domains and across model backbones. The exact numbers are less detailed in the local summaries than for some other systems, but the qualitative result is that agent design itself becomes an optimization target.

The Darwin Gödel Machine (DGM) extends this idea with open-ended archives and self-modification [76]. It maintains a growing archive of agent implementations, samples parent agents with a diversity bias, uses a foundation model to modify the agent code, evaluates the child on coding benchmarks, and keeps improvements. The local materials report improvement on SWE-bench from 20.0% to 50.0% and on a Polyglot multi-language coding benchmark from 14.2% to 30.7%. DGM is "Darwin" because it uses population search and selection; it is "Gödel" because agents can modify their own code in an open-ended process inspired by self-referential improvement.

DGM illustrates the value of an archive over a single lineage. If a child fails, the parent remains. If a branch discovers a useful tool-use pattern but not the best score, it can persist as a stepping stone. If a later task distribution changes, the archive may contain a better starting point than the current champion. This is more robust than hill-climbing and closer to open-ended evolution. The trade-off is cost: each child must be evaluated on meaningful software tasks, and code modifications may break syntax, dependencies, or safety assumptions.

Gödel Agent takes a more direct self-referential route [72]. It allows an agent to inspect and modify its own runtime logic using LLM-driven monkey patching. The loop is: execute on tasks, collect results and feedback, analyze the current code and behavior, propose a patch, apply it at runtime, validate, and accept or roll back. This architecture is closer to online self-modification than to population-level evolution. Its safety mechanisms include rollback, validation, and high-level objective constraints. The local materials report improvements on HotPotQA, ALFWorld, and broader text tasks, though the summarized numbers are qualitative rather than a detailed table.

Gödel Agent exposes a different risk profile. Runtime monkey patching is flexible and fast, but it is Python-specific and can introduce nondeterminism. It also blurs the boundary between a candidate under evaluation and the evaluator environment. A population archive such as DGM can quarantine children; a runtime-patching agent must be especially careful about rollback and validation because the executing system is the object being changed. For safety-critical self-evolution, this distinction matters.

Agent Symbolic Learning offers a more structured and less code-destructive view of agent self-evolution [1]. It treats an agent as a symbolic network whose learnable "weights" are prompts, tools, and pipeline connections. After executing a task, an LLM produces a language loss from the trajectory. Then language gradients are backpropagated through the symbolic network, and optimizers update prompt nodes, tool nodes, or pipeline nodes. The formulas from the local

materials are:

$$\mathcal{L}_{\text{lang}} = \text{LLM}(\mathcal{P}_{\text{loss}}(\tau)), \tag{97}$$

$$\nabla^n_{\text{lang}} = \text{LLM}\left(\mathcal{P}_{\text{gradient}}(\nabla^{n+1}_{\text{lang}}, \mathcal{I}_n, \mathcal{O}_n, \mathcal{P}_n, \mathcal{T}_n, \mathcal{L}_{\text{lang}})\right), \tag{98}$$

followed by symbolic updates to prompts, tools, and pipeline structure. The metaphor is backpropagation in language space. Unlike DGM, the update target is not arbitrary source code; it is a structured agent graph.

The benchmark numbers in the local materials suggest broad but moderate gains. On HotPotQA, symbolic learning improves F1 from 36.2 to 39.1 and exact match from 22.8 to 25.6. On MATH with a GPT-4 backbone, accuracy rises from 56.0% to 60.7%. On HumanEval, pass@1 improves from 68.3% to 73.2%. On a software-development executability score from 1 to 4, the reported average increases from 2.4 for agents to 3.8. These results support the idea that language-space credit assignment can improve deployed agents without weight updates.

Absolute Zero moves self-evolution into reinforcement learning with verifiable rewards and no external data [77]. A single model plays two roles: task proposer and task solver. It generates reasoning tasks, attempts to solve them, and receives rewards from a code executor. The proposer is rewarded for tasks that are solvable, challenging, and diverse; the solver is rewarded for correct solutions. The local materials describe this as zero-data self-play RLVR, with PPO or GRPO-style updates for both roles. The key equation is a dual reward loop:

$$t \sim \pi_{\text{prop}}(\cdot \mid s), \qquad y \sim \pi_{\text{solve}}(\cdot \mid t), \qquad r = \text{Execute}(y, t). \tag{99}$$

The model improves by generating its own curriculum.

Absolute Zero is not evolutionary code search in the narrow sense, but it belongs in this chapter because it shares the self-evolution loop: generate tasks, evaluate outcomes, update the generator/solver, and repeat. It also demonstrates the importance of verifiability. Without a code executor, self-generated tasks could drift into ambiguous or self-serving problems. With execution-based rewards, the system can train on an automatically grounded curriculum. The local materials report that AZR-Coder-7B reaches state-of-the-art results among 7B models on coding averages and remains competitive on HumanEval+, MBPP+, LiveCodeBench, GSM8K, and MATH, despite using zero external training data.

SelfEvolve, from 2023, is an earlier code self-improvement pipeline [25]. It has two stages. First, the LLM generates knowledge such as API documentation or algorithm descriptions from its own parameters. Second, generated code is executed in a sandbox, and error messages are fed back for iterative revision. The local materials give formulas for knowledge-conditioned generation and revision, including:

$$P(Y \mid K) = \prod_i p_\theta(Y_i \mid Y_{<i}, X, K), \tag{100}$$

where K is self-generated knowledge, and a revision distribution conditioned on the original problem, generated code, knowledge, and error message. SelfEvolve is closer to self-debugging than to population evolution, but it supplies a key mechanism later systems reuse: execution feedback as a natural-language training signal for the next attempt.

Together, these systems define a spectrum. ADAS searches agent architectures in code space. DGM evolves a population of self-modifying coding agents. Gödel Agent performs runtime self-modification. Agent Symbolic Learning updates structured prompts, tools, and pipelines through language gradients. Absolute Zero trains a model through self-generated verifiable tasks. SelfEvolve iteratively refines code using self-generated knowledge and execution errors. All instantiate self-evolution, but their update substrates differ: architecture code, agent source, runtime methods, symbolic graph weights, model policy, and generated code.

Table 9: Agent self-evolution systems by substrate, feedback signal, and reported benchmark evidence.

System	Evolved substrate	Feedback signal	Reported evidence in local materials	Main limitation
ADAS	Python programs implementing agent architectures	Benchmark scores on agent tasks	Outperforms hand-designed agents on ARC and GFootball; transfer across domains and models	Expensive search over huge code space
DGM	Archive of coding-agent implementations	SWE-bench and Polyglot scores	SWE-bench 20.0% → 50.0%; Polyglot 14.2% → 30.7%	Evaluation cost and safe self-modification
Gödel Agent	Runtime agent logic via monkey patches	Task feedback plus validation	Reported gains on HotPotQA, ALFWorld, and multi-domain text tasks	Runtime patching risk and weaker portability
Agent Symbolic Learning	Prompts, tools, and pipeline connections	Language loss and language gradients	HotPotQA F1 36.2 → 39.1; MATH 56.0% → 60.7%; HumanEval 68.3% → 73.2%	Depends on strong LLM for gradient quality
Absolute Zero	Proposer/solver policy through RLVR	Code-executor rewards on self-generated tasks	7B-level SOTA coding average in local notes; competitive math transfer	Curriculum collapse and executor-limited domains
SelfEvolve	Generated code plus self-generated knowledge	Sandbox errors and test feedback	DS-1000 49.4 → 57.2; HumanEval pass@1 74.39 → 85.98	Mainly handles API/assertion errors; hand-written prompts
SE-Agent	Reasoning trajectories for code agents	SWE-bench Verified outcomes	Up to 55% relative improvement in local notes; repository reports 80% Top-1	Expensive multi-trajectory search

A unifying perspective is that agent self-evolution requires three memories. The first is episodic memory: what happened in a particular task attempt. The second is design memory: which prompts, tools, code changes, or trajectories have worked across attempts. The third is evaluative memory: which measurements are trustworthy and under what conditions. Systems that lack episodic memory repeat mistakes. Systems that lack design memory cannot accumulate improvements. Systems that lack evaluative memory overfit, regress, or reward hacks.

Another unifying issue is containment. Prompt and symbolic updates are relatively easy to sandbox because they alter text fields. Code updates need tests, permissions, dependency control, and rollback. Runtime self-modification needs even stronger containment because the subject and mechanism of change overlap. RL self-play needs reward containment because a model may learn to propose tasks that exploit the verifier. Self-evolution is therefore as much an infrastructure problem as an algorithmic one.

4.6 Benchmark Comparison and Evidence Quality

Benchmark comparison in this area is difficult because systems optimize different substrates under different budgets. OPRO optimizes prompts and black-box variables. FunSearch optimizes mathematical programs. AlphaEvolve optimizes algorithms and infrastructure code. DGM and SE-Agent target software-engineering agents. Agent Symbolic Learning updates agent components across question answering, math, code, software development, and creative writing. Absolute Zero trains models through self-play. A single leaderboard would obscure these differences. A better comparison asks: What is the artifact? What is the evaluator? What baseline is used? How much search or training budget is consumed? Is the result held out, reproducible, and resistant to

evaluator gaming?

Table 10: Selected numeric results reported in the local research materials and verified web sources for open repositories where noted. Values should be compared within rows, not across unrelated benchmarks.

System	Benchmark or domain	Baseline	Reported evolved result	Notes
OPRO	Instruction optimization on reasoning tasks	Initial or prior prompts	Improved prompt scores over iterative search	Exact numbers depend on dataset and prompt placement; key evidence is the optimization trajectory.
OPRO	Linear regression/TSP optimization	Hand or heuristic initial candidates	Lower objective values through text-conditioned proposals	Demonstrates black-box optimization beyond prompts.
FunSearch	Cap set	Prior constructions in studied setting	Better evolved construction	Scientific discovery through executable program search.
FunSearch	Bin packing	Existing heuristic baselines	Improved heuristic program	Demonstrates practical heuristic discovery.
AlphaEvolve	4×4 complex matrix multiplication	Strassen-era multiplication count	48 multiplications	Reported as first improvement in 56 years in local materials.
AlphaEvolve	50+ math problems	Known SOTA	Rediscovered SOTA for 75%, improved 20%	Indicates broad evaluator-driven search.
AlphaEvolve	Data-center scheduling	Existing Borg scheduling policy	0.7% worldwide compute recovered	Industrial result; evaluator and deployment details are domain-specific.
AlphaEvolve	LLM training kernels	Existing kernels	23% and 32% speedups in reported attention settings	Performance depends on hardware and workload.
DGM	SWE-bench	20.0%	50.0%	Archive-based self-improving coding agents.
DGM	Polyglot coding	14.2%	30.7%	Multi-language transfer evidence.
Agent Symbolic Learning	HotPotQA	F1 36.2 / EM 22.8	F1 39.1 / EM 25.6	Structured language-gradient updates.
Agent Symbolic Learning	MATH	56.0%	60.7%	GPT-4 backbone in local materials.
Agent Symbolic Learning	HumanEval	68.3% pass@1	73.2% pass@1	Code-generation improvement without weight updates.
SelfEvolve	DS-1000	ChatGPT 49.4	57.2	Full pipeline, +15.8% relative in local notes.
SelfEvolve	HumanEval	ChatGPT 74.39 pass@1	85.98 pass@1	Approaches GPT-4 baseline in local table.
SelfEvolve	TransCoder	ChatGPT 88.3 accuracy / 88.5 pass@1	90.4 accuracy / 90.9 pass@1	Translation and execution refinement.
Absolute Zero	7B coding average	Data-trained comparison methods	SOTA among 7B models in local notes	Zero external data; exact aggregate depends on benchmark mix.
SE-Agent	SWE-bench Verified	Base code agents	Up to 55% relative improvement; repository reports 80% Top-1	Trajectory-level evolution; web-verified repository claim.

Table 10: Selected numeric benchmark results (continued).

System	Benchmark or domain	Baseline	Reported evolved result	Notes
OpenEvolve	Prompt optimization example	Baseline prompt	Repository reports +23% HotpotQA accuracy in example	Open-source framework result, not a universal benchmark.
CodeEvolve	Algorithm discovery tasks	Open/closed baselines	arXiv abstract reports SOTA on several tasks	Requires paper-specific tables for exact per-task values.

The table deliberately mixes precise and qualitative entries because the source materials vary in granularity. For some systems, the local notes provide exact before/after values. For others, they provide claims such as "competitive," "SOTA," or "outperforms hand-designed agents" without a row-by-row numeric breakdown. A rigorous meta-analysis should return to the original papers, extract task definitions, budgets, seeds, and confidence intervals, and separate training, validation, and test performance. The goal here is to map the evidence landscape rather than to declare a universal winner.

Several benchmark patterns recur. First, the largest gains often occur when the baseline is a fixed agent or fixed prompt and the evolved system can exploit task-specific feedback. This does not diminish the result; it clarifies that self-evolution is a form of adaptation. Second, code and math tasks are overrepresented because they have verifiable rewards. Third, software-engineering benchmarks such as SWE-bench are attractive because they test realistic agent behavior, but they are costly and sensitive to harness details. Fourth, infrastructure optimization results can be highly valuable even when the percentage improvement looks small. A 0.7% compute recovery at global scale may be more economically significant than a large relative gain on a toy task.

Evidence quality depends on evaluator independence. If candidate generation sees the exact tests, evolution may overfit. If validation is performed on the same distribution as selection, performance may regress out of distribution. If the evaluator is an LLM judge, model bias and prompt sensitivity enter the loop. If the benchmark is public and widely used, contamination is possible. Therefore, high-quality self-evolution evidence should report hidden tests, ablations, compute budgets, random seeds, failed-run rates, and regression checks.

Ablations are especially important. In evolutionary systems, improvement can come from many sources: more model calls, better prompts, stronger base models, larger archives, smarter selection, more diverse parents, better evaluators, or simply more retries. Without ablations, it is hard to know whether a proposed evolutionary mechanism is necessary. For example, a MAP-Elites archive should be compared against best-only selection under the same evaluation budget. A dual-model architecture should be compared against a single strong model with equal cost. A trajectory recombination operator should be compared against independent retries and self-reflection. The literature is beginning to include such ablations, but survey readers should treat headline scores as incomplete without them.

Another issue is budget normalization. A system that evaluates ten thousand candidates may outperform a system that evaluates one hundred candidates for reasons unrelated to algorithmic design. This is not a flaw if the use case permits the budget, but it must be reported. Useful metrics include score versus number of evaluations, score versus wall-clock time, score versus token cost, and score versus human engineering time. In industrial settings, the relevant metric may be return on compute investment. In scientific discovery, the relevant metric may be novelty per evaluator call.

For agent self-evolution, transfer is a critical benchmark dimension. If an evolved agent improves only on the tasks used for selection, it is a specialized optimizer. If its design transfers to new tasks, models, or domains, it is closer to a general self-evolution method. The local ADAS notes emphasize cross-domain and cross-model transfer. DGM reports transfer across coding benchmarks. Absolute Zero reports cross-domain transfer from code self-play to math reasoning. Such results are important because self-evolution should ideally discover reusable mechanisms, not only benchmark-specific hacks.

4.7 Design Patterns for Robust AI Self-Evolution

Across the systems surveyed, several design patterns emerge. The first is **representation discipline**. The evolved artifact should have a clear boundary. FunSearch marks one function. CodeEvolve modifies code between markers with structured diffs. Agent Symbolic Learning updates named prompts, tools, and pipeline nodes. SE-Agent manipulates trajectories. Clear boundaries make evaluation, rollback, and attribution possible.

The second pattern is **verifier-first design**. The evaluator should be designed before large-scale generation begins. A weak evaluator makes the system optimize the wrong target. A slow evaluator makes search impractical. A non-deterministic evaluator makes selection noisy. A leaky evaluator invites reward hacking. The most successful systems are those in which candidate quality can be measured automatically and cheaply enough to support many iterations.

The third pattern is **archive over memory**. A chat transcript is not an archive. An archive stores artifacts, scores, descriptors, provenance, failure traces, and selection status in a structured way. It supports parent sampling, diversity analysis, rollback, and audit. OPRO's history is a minimal archive. FunSearch's programs database is a richer archive. DGM's open-ended agent collection and AlphaEvolve's quality-diversity map are more powerful still.

The fourth pattern is **diversity preservation**. Best-only loops are simple but brittle. They discard variants that may become useful later. Island models, MAP-Elites, signature clustering, and trajectory pools all preserve alternative hypotheses. Diversity is not only about avoiding local optima; it is about preserving interpretability. A diverse archive lets researchers compare families of solutions and identify mechanisms that correlate with success.

The fifth pattern is **failure utilization**. In ordinary pipelines, failed generations are discarded. In self-evolution, failures can be compressed into useful signals: syntax errors, timeout causes, failing test names, counterexamples, performance profiles, or reasoning blind spots. SelfEvolve uses error messages for revision. OpenEvolve-style artifact side channels feed diagnostics back to the LLM. SE-Agent uses failed trajectories for revision. The challenge is to summarize failures without flooding context.

The sixth pattern is **model-budget scheduling**. Not every generation deserves the strongest model. AlphaEvolve's Flash/Pro split suggests a general recipe: use cheaper models or higher temperatures for broad exploration, stronger models for deep repair or exploitation, and route calls based on archive state. Open-source systems can implement analogous policies with open-weight and proprietary models. Budget-aware evolution is essential for scaling beyond demonstrations.

The seventh pattern is **regression control**. Any self-evolving system can forget or degrade. Code changes may improve one metric while breaking another. Prompt changes may overfit. Agent patches may alter safety behavior. Robust systems need baselines, validation gates, canary tasks, hidden tests, and rollback. The acceptance criterion should be multi-objective when deployment requires reliability, not just maximum benchmark score.

The eighth pattern is **human-legible lineage**. Scientific and engineering value increases when humans can inspect how a solution evolved. FunSearch's versioned functions, DGM's archive, and

CodeEvolve's genealogical tracking all support lineage analysis. A lineage can reveal whether a result emerged from gradual refinement, a sudden imported idea, or evaluator exploitation. This is important for trust and for converting evolved artifacts into human knowledge.

4.8 Open Problems

The first open problem is evaluator generality. Current successes concentrate in domains with automatic verification: code, math, games, scheduling, kernels, and benchmarked agents. Many important AI tasks involve ambiguity, human preference, long-horizon social effects, or safety constraints that are hard to score. Extending self-evolution to these domains requires richer evaluators, uncertainty estimates, and human-in-the-loop governance.

The second open problem is theoretical understanding. LLM-driven mutation is not a simple random process. It is conditioned on pretraining, prompt design, archive content, and model decoding. Classical convergence analyses for evolutionary algorithms do not directly apply. We need models of semantic variation, archive-induced bias, and evaluator noise. Without theory, system design remains empirical and brittle.

The third open problem is contamination and benchmark exhaustion. As systems repeatedly optimize public benchmarks, the benchmarks become less informative. Self-evolution can consume a benchmark by discovering its quirks. This is especially acute for software-engineering tasks where repositories, tests, and solutions may leak into model training data. Dynamic benchmarks, private tests, and task generation with verifiable rewards are partial remedies.

The fourth open problem is safety of self-modifying code. An evolved program may consume excessive resources, access unintended files, or learn to manipulate its evaluator. Runtime self-modification raises additional concerns because the modification mechanism itself can be changed. Practical systems need sandboxing, least-privilege execution, provenance tracking, policy checks, and emergency rollback. Safety should be evaluated as a first-class metric rather than treated as an implementation detail.

The fifth open problem is multi-objective alignment. Real deployments care about accuracy, cost, latency, maintainability, privacy, fairness, and robustness. A scalar benchmark score cannot encode all of these. MAP-Elites and Pareto archives offer one route, but descriptor choice is hard. If the descriptors are poor, the archive may preserve diversity that is irrelevant while missing diversity that matters.

The sixth open problem is the relationship between self-evolution and weight learning. OPRO, FunSearch, AlphaEvolve, DGM, and SE-Agent can improve frozen-model systems through external search. Absolute Zero updates model policies through self-play RL. Future systems may combine both: an external archive discovers artifacts, and a model is periodically trained or distilled on successful lineages. The challenge is to retain verifiable gains without distilling evaluator hacks or reducing diversity.

The seventh open problem is scientific credit assignment. When an evolved program solves a mathematical problem, which component deserves credit: the base model, the prompt, the evaluator, the archive, the human problem formulation, or the compute budget? This is not only philosophical. It affects reproducibility, intellectual property, and how researchers interpret discoveries. Lineage-aware archives and detailed experiment logs are necessary for credible claims.

4.9 Synthesis

Evolutionary code and algorithm discovery marks a transition in AI self-evolution from isolated generation to cumulative search. OPRO shows that a natural-language history of candidates and

scores can turn an LLM into a black-box optimizer. FunSearch shows that the candidates can be executable programs whose evolved behavior yields mathematical discoveries. AlphaEvolve shows that the same idea can scale with model ensembles, automated evaluators, and quality-diversity archives to broader algorithmic and infrastructure domains. OpenEvolve and CodeEvolve show that the community is translating these ideas into reusable frameworks. SE-Agent shows that evolutionary operations can act on reasoning trajectories. ADAS, DGM, Gödel Agent, Agent Symbolic Learning, Absolute Zero, and SelfEvolve show that agents themselves can become the evolving substrate.

The unifying equation is simple:

$$\text{self-evolution} = \text{generation} + \text{evaluation} + \text{memory} + \text{selection} + \text{variation under constraints.} \quad (101)$$

Each term is necessary. Generation without evaluation is imagination. Evaluation without memory is repeated trial and error. Memory without selection is an unstructured log. Selection without variation is stagnation. Variation without constraints is unsafe. The systems surveyed in this chapter differ in how they instantiate the terms, but they all rely on the same closed-loop principle.

For future survey and benchmark work, the most important recommendation is to report the entire evolutionary ecology. Papers should specify the artifact representation, model routing, prompt construction, archive policy, diversity descriptors, evaluator details, validation split, compute budget, failure rate, and rollback policy. Without these details, self-evolution results are difficult to compare and easy to overstate. With them, the field can move from impressive demonstrations toward reliable, auditable, and reusable self-improving AI systems.

4.10 System-by-System Analytical Notes

The following notes expand the comparison by applying a common analytical lens to each system. They are included because evolutionary self-improvement methods are often described with similar vocabulary even when their engineering assumptions differ substantially. For each system, the key questions are what evolves, what feedback is trusted, what memory is retained, and what failure mode dominates.

OPRO. OPRO evolves instruction strings and serialized optimization variables. Its trusted feedback signal is scored history, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, OPRO should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For OPRO, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is prompt overfitting and noisy scorer feedback. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, OPRO should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

FunSearch. FunSearch evolves versioned Python functions. Its trusted feedback signal is sandboxed mathematical objective, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, FunSearch should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For FunSearch, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is evaluator leakage and premature convergence. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, FunSearch should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

AlphaEvolve. AlphaEvolve evolves algorithmic code artifacts. Its trusted feedback signal is automated program and infrastructure evaluators, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, AlphaEvolve should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For AlphaEvolve, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is large compute budgets and limited public reproducibility. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, AlphaEvolve should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

OpenEvolve. OpenEvolve evolves user-defined code and prompts. Its trusted feedback signal is custom evaluator callbacks and artifacts, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, OpenEvolve should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For OpenEvolve, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is quality depends on practitioner evaluator design. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, OpenEvolve should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

CodeEvolve. CodeEvolve evolves marked code regions and prompt populations. Its trusted feedback signal is sandbox metrics with island orchestration, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, CodeEvolve should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For CodeEvolve, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is distributed execution complexity. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, CodeEvolve should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

SE-Agent. SE-Agent evolves multi-step reasoning trajectories. Its trusted feedback signal is SWE-bench Verified issue resolution, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, SE-Agent should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For SE-Agent, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is cost of multi-trajectory exploration. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, SE-Agent should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

ADAS. ADAS evolves agent architectures written as Python programs. Its trusted feedback signal is agent benchmark scores, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, ADAS should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For ADAS, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is vast Turing-complete search space. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, ADAS should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

DGM. DGM evolves archives of self-modifying coding agents. Its trusted feedback signal is coding benchmark improvement, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, DGM should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For DGM, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is safe archive management and expensive evaluations. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, DGM should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

Godel Agent. Godel Agent evolves runtime methods and behavior logic. Its trusted feedback signal is validation after monkey patching, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, Godel Agent should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For Godel Agent, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is runtime nondeterminism and rollback safety. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, Godel Agent should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

Agent Symbolic Learning. Agent Symbolic Learning evolves prompts, tools, and pipeline edges. Its trusted feedback signal is language losses and gradients, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, Agent Symbolic Learning should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For Agent Symbolic Learning, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is dependence on strong LLM-generated gradients. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, Agent Symbolic Learning should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

Absolute Zero. Absolute Zero evolves self-generated tasks and solver policies. Its trusted feedback signal is code-executor rewards, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, Absolute Zero should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For Absolute Zero, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is curriculum collapse and verifier scope. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, Absolute Zero should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

SelfEvolve. SelfEvolve evolves generated code with self-generated knowledge. Its trusted feedback signal is sandbox errors and assertions, which means that the outer loop is only as reliable as the mechanism that assigns scores, errors, or rewards. In the self-evolution taxonomy used in this chapter, SelfEvolve should be read not as a standalone model capability but as an interaction pattern among a generator, an evaluator, and an archive. The generator supplies candidate variation; the evaluator converts behavior into a selection signal; the archive determines which traces survive into future contexts. This perspective is useful because it prevents a common misunderstanding: the LLM is not magically improving in isolation, but is participating in a larger cybernetic system that accumulates evidence across attempts.

For SelfEvolve, the central engineering question is how much freedom the candidate artifact receives. A narrow artifact boundary makes validation simpler and reduces the chance that the system changes its own measurement process. A broad artifact boundary increases the chance of discovering qualitatively new strategies, but it also increases the burden on sandboxing, provenance, and regression tests. The dominant risk is limited correction scope and hand-written prompts. A robust implementation should therefore include held-out validation, artifact lineage, failure summarization, and a rollback path. In comparative studies, SelfEvolve should be evaluated not only by its best score but also by its score as a function of evaluation budget, the diversity of retained candidates, and the stability of improvements under repeated seeds.

4.11 Methodological Checklist for Reproducible Evolutionary Discovery

A reproducible evolutionary discovery paper should describe the search process with the same care that a machine-learning paper describes data, model, and optimizer. The first required item is the *candidate schema*. The paper should state whether the candidate is a prompt, a full program, a patch, a marked function, a trajectory, a tool graph, or a model-generated task. It should also state which parts of the surrounding system are immutable. Without this boundary, readers cannot distinguish an algorithmic improvement from an accidental change in the benchmark harness. For code artifacts, the schema should include allowed imports, allowed file writes, time limits, memory limits, dependency versions, and whether the candidate can observe evaluator internals.

The second item is the *proposal distribution*. In LLM-guided evolution, the proposal distribution is not a simple Gaussian mutation. It is a product of model choice, decoding temperature, prompt structure, parent sampling, context compression, and output parsing. A reproducibility report should therefore include model names, model versions or dates, sampling parameters, number of samples per prompt, retry policy, prompt templates, and any post-processing rules. If a system uses a dual-model policy, such as a cheap model for breadth and a stronger model for depth, it should report when each model is invoked and how model-routing decisions are made. If failures are summarized back to the model, the summary template should be part of the method because it changes future variation.

The third item is the *archive policy*. A self-evolution system is defined by what it remembers. Does it store every candidate, only valid candidates, only elites, or also failures? Does it preserve lineages? Are candidates clustered by behavior signatures, MAP-Elites cells, prompt embeddings, task domains, or hand-designed descriptors? Does parent selection prefer recent candidates, high-fitness candidates, novel candidates, or underexplored niches? These choices can dominate performance. A best-only archive approximates hill climbing; an island archive approximates distributed evolutionary search; a MAP-Elites archive approximates landscape illumination. Papers should report archive size, pruning rules, reset rules, and whether validation results can overwrite training scores.

The fourth item is the *evaluator contract*. The evaluator should be described as an interface with inputs, outputs, failure modes, and trust assumptions. For mathematical program search, the

evaluator may return a deterministic score. For code agents, it may return pass/fail on a benchmark issue. For agent trajectories, it may involve an execution environment and a judge. The contract should define what counts as invalid, how timeouts are handled, whether partial credit is possible, and whether stochastic results are averaged. Hidden tests and public tests should be separated. If the evaluator is expensive, the paper should report how many evaluations were run and how many were discarded before scoring.

The fifth item is *budget accounting*. Evolutionary systems can look much stronger than single-shot baselines simply because they consume more attempts. Fair comparison does not require identical budgets for all use cases, but it does require transparent budgets. Authors should report token counts, model-call counts, candidate counts, valid-candidate counts, evaluator calls, wall-clock time, hardware, and monetary cost when possible. The most informative plot is often performance versus evaluator budget rather than final score alone. A system that reaches a moderate score quickly may be preferable to a system that reaches a higher score only after enormous search.

The sixth item is *regression measurement*. A candidate should not be accepted merely because it improves one scalar on one batch. The system should measure whether it preserves previous capabilities, whether it generalizes to held-out tasks, and whether it remains safe under adversarial or randomized tests. In code evolution, regression suites should include functional correctness, performance, determinism, and resource use. In agent evolution, regression suites should include tool-use safety, refusal behavior where appropriate, memory consistency, and robustness to prompt perturbations. The acceptance gate should be explicit. For multi-objective systems, the paper should state whether it uses lexicographic ordering, scalarization, Pareto dominance, or niche-specific elites.

The seventh item is *lineage reporting*. A final artifact without a lineage is difficult to trust. Lineage reporting should include the parent identifiers, mutation prompt, evaluation results, and reason for acceptance. For scientific discoveries, it should include intermediate artifacts that reveal how the idea emerged. This enables human researchers to inspect whether the system discovered a genuine mechanism or exploited an evaluator quirk. Lineage is also useful for debugging: if a later generation regresses, the archive can identify the branch where the regression entered.

The eighth item is *negative evidence*. Evolutionary discovery papers often emphasize successful branches, but failed branches are scientifically valuable. They reveal what the evaluator rejects, what the model tends to misunderstand, and which search operators are unproductive. A concise failure taxonomy can be more useful than a long list of raw failed programs. Typical categories include syntax errors, semantic type errors, timeout failures, test overfitting, numerical instability, nondeterminism, dependency misuse, unsafe file access, and reward hacking. Reporting the distribution of failures helps other researchers reproduce the search and design better filters.

The ninth item is *human intervention accounting*. Some systems claim autonomous self-evolution, yet humans may adjust prompts, repair infrastructure, choose benchmarks, or manually inspect candidates during the run. These interventions are not necessarily bad; they are often necessary for early research. But they should be reported. A clear distinction between autonomous search, human-curated search, and post-hoc human interpretation prevents overclaiming. For deployment, the level of acceptable human intervention will depend on risk: scientific exploration may tolerate human curation, while autonomous production code modification requires stricter automation and oversight.

The tenth item is *release artifacts*. A reproducible release should include the final artifact, seed artifacts, prompt templates, evaluator code, benchmark splits, configuration files, and archive logs. If a proprietary model prevents exact reproduction, the release can still provide enough structure for approximate reproduction with other models. Open-source systems such as FunSearch, OpenEvolve, CodeEvolve, and SE-Agent are valuable because they expose this surrounding infrastructure, not

merely because they publish a clever prompt.

4.12 A Conceptual Taxonomy of Evolutionary Self-Improvement Loops

The systems in this chapter can be organized by the depth at which evolution intervenes. At the shallowest level is *instruction evolution*. The artifact is a natural-language prompt, and the evaluator measures task performance. OPRO is the representative case. Instruction evolution is easy to deploy because it does not require code execution, but it is limited by the expressivity of the prompt and by the stability of LLM behavior under prompt changes. It is best suited for tasks where a fixed model already has latent competence and needs better elicitation.

The next level is *program-fragment evolution*. The artifact is a function or marked code region. FunSearch is the cleanest example. This level has strong interpretability because the evolved fragment can be read and analyzed. It also has strong containment because the evaluator can restrict the fragment's inputs and outputs. The limitation is that the surrounding scaffold must already define the problem well. If the key innovation requires changing the scaffold, a single-function search may be too narrow.

The third level is *codebase evolution*. The artifact may include multiple functions, files, or algorithmic modules. AlphaEvolve, OpenEvolve, and CodeEvolve operate closer to this level. Codebase evolution can discover larger design changes and optimize real systems, but it requires build systems, integration tests, performance harnesses, dependency management, and stronger sandboxing. It also needs better credit assignment because a performance change may result from interactions among many edits.

The fourth level is *trajectory evolution*. The artifact is a sequence of thoughts, actions, tool calls, or intermediate plans. SE-Agent is representative. Trajectory evolution is attractive when the final code change is less important than the process by which an agent reaches it. A revised or recombined trajectory can teach the agent how to approach similar problems. The risk is that trajectory quality may depend heavily on the base model and environment state, making reproduction more difficult.

The fifth level is *architecture evolution*. The artifact is an agent design, workflow, or multi-agent organization. ADAS searches this level by asking a meta agent to write agent programs. DGM evolves agent implementations in an archive. Architecture evolution can discover new reasoning and tool-use patterns, but it has the broadest search space and the hardest evaluation problem. A good architecture may be robust across tasks, while a bad architecture may overfit to benchmark quirks in subtle ways.

The sixth level is *self-referential runtime evolution*. The artifact is the agent's own operating logic as it executes. Gödel Agent illustrates this level. Runtime evolution is powerful because adaptation can happen online, but it requires strong rollback and containment. The system must ensure that the mechanism for change does not corrupt the mechanism for evaluation. This is the level at which safety concerns become most acute.

The seventh level is *policy evolution through self-generated data*. Absolute Zero represents this level. The artifact being improved is no longer only external code or prompts; it is the model policy itself, trained through self-play tasks and verifiable rewards. This level can produce more durable improvements because knowledge may enter the weights, but it also raises the cost and risk of training. It requires reward design, curriculum control, and safeguards against self-generated distribution collapse.

These levels are not mutually exclusive. A future self-evolving system may use OPRO-style prompt optimization to improve its mutation prompts, FunSearch-style program evolution to discover heuristics, MAP-Elites to preserve code diversity, SE-Agent-style trajectory recombination to improve problem-solving traces, ADAS-style architecture search to redesign the agent, and Absolute-

Zero-style self-play to train the underlying model. The research challenge is to compose these loops without creating uncontrolled feedback cycles.

4.13 Implications for the Broader Survey

For the broader topic of AI self-evolution, evolutionary code and algorithm discovery supplies the most concrete evidence that AI systems can accumulate improvements outside conventional supervised training. The evidence is concrete because the artifacts are inspectable and the evaluators are often executable. A prompt can be read, a function can be run, a patch can be tested, and an archive can be audited. This makes the area a useful anchor for discussions that might otherwise become speculative.

The chapter also suggests that self-evolution should be studied as a systems discipline. The LLM is important, but it is not the entire system. The evaluator, archive, sandbox, prompt compiler, model router, benchmark harness, and logging infrastructure are equally important. In many cases, the innovation lies in how these components are connected. FunSearch's versioned prompt format, AlphaEvolve's quality-diversity archive, DGM's open-ended agent collection, and Agent Symbolic Learning's language-gradient graph are system-level contributions.

A second implication is that verifiability will shape the near-term geography of self-evolution. Code, mathematics, theorem-like tasks, games, scheduling, chip design, and kernel optimization are likely to progress faster because they provide reliable feedback. Domains such as negotiation, education, medicine, and social planning will require more careful evaluator design and governance. The field should resist the temptation to generalize from verifiable domains to all forms of intelligence without qualification.

A third implication is that diversity is a safety and capability mechanism. It is a capability mechanism because diverse archives preserve stepping stones and alternative strategies. It is a safety mechanism because diversity makes it easier to compare artifacts and detect suspicious convergence toward evaluator hacks. If every branch independently discovers the same shortcut, researchers should ask whether the shortcut reflects the intended objective. If different branches solve the task through different mechanisms, humans gain more insight.

A fourth implication is that self-evolution changes the role of human researchers. Humans may spend less time writing the final heuristic and more time designing the search space, evaluator, constraints, and audit process. This is not a reduction in scientific responsibility. It is a shift from direct construction to ecosystem design. The quality of the resulting discovery depends on whether the ecosystem rewards genuine progress.

A fifth implication is that publication norms need to evolve. A self-evolution result is not fully described by a final score and a method diagram. It requires an archive summary, budget report, evaluator specification, lineage evidence, and failure analysis. Without these, claims of autonomous improvement are hard to interpret. With them, the community can compare systems on reliability, efficiency, and generality, not only on maximum benchmark score.

In summary, evolutionary code and algorithm discovery is the operational core of current AI self-evolution research. It shows how LLMs can serve as optimizers, mutation operators, recombiners, debuggers, and meta-designers when embedded in a loop with memory and verification. The field's next step is to make these loops more reproducible, safer, more budget-aware, and more general, while preserving the interpretability that makes program evolution scientifically valuable.

5 Evaluation and Benchmark Analysis

Evaluation is the central bottleneck of AI self-evolution. A self-evolving system is defined not merely by the fact that it changes, but by the claim that its changes are improvements. That claim is always mediated by an evaluator: a unit test, a hidden benchmark, a reward model, a formal verifier, a human preference, a simulated environment, a code review gate, a theorem prover, or a production metric. If the evaluator is faithful, self-evolution can accumulate genuine capability. If the evaluator is weak, noisy, contaminated, or manipulable, the same optimization loop can amplify illusion. This chapter therefore treats evaluation as an object of design rather than as a passive reporting layer.

The recent literature makes this point repeatedly. Reflexion improves HumanEval pass@1 by storing verbal feedback, but the improvement depends on an evaluator that can tell the agent whether a program is correct [49]. SelfEvolve improves DS-1000 and HumanEval by executing candidate code and feeding errors back to the model [25]. Agent Symbolic Learning reports gains on HotPotQA, MATH, HumanEval, and software-development executability by converting trajectories into language losses and language gradients [1]. Darwin Godel Machine (DGM) and AlphaEvolve both make evaluation an explicit selection operator: candidate agents or programs are retained only when executable benchmarks or domain evaluators score them better than prior variants [40, 76]. ReVeal goes further by arguing that code generation and self-verification must co-evolve, because a generator that cannot evaluate itself cannot reliably improve over many turns [27]. These systems differ in algorithmic form, but they share a single dependency: improvement is only as valid as the feedback channel that detects it.

For survey purposes, the evaluation problem has two layers. The first layer is conventional benchmark analysis: which datasets, metrics, and reported scores are used by current self-evolution methods? The second layer is process evaluation: how should one evaluate the dynamics of an evolving system over time, including its cost, safety, archive diversity, transfer, regression behavior, and sensitivity to evaluator design? Static benchmarks remain necessary because they provide comparability. However, self-evolution systems are not static models. They perform repeated attempts, store reflections, rewrite workflows, generate tests, search over code, and sometimes modify the agent itself. A benchmark score reported after an evolution loop is therefore a property of the entire loop: base model, prompts, memory, tools, search policy, evaluator, budget, selection rule, and stopping criterion.

This chapter reviews evaluation methodology across six dimensions, surveys code-generation, mathematical-reasoning, agent, open-ended, and GitHub project-quality benchmarks, and concludes with a recommended protocol for evaluating self-evolution systems. The emphasis is on exact reported numbers where the source materials provide them, but also on the limitations of those numbers. A self-evolution survey should not only ask which system scores highest; it should ask what was allowed during improvement, whether the improvement transfers, what cost was paid, whether regressions were measured, and whether the evaluator could itself be gamed.

5.1 Evaluation Methodology

A useful evaluation methodology for self-evolving AI must be multidimensional. Reporting only accuracy or pass@1 is insufficient because an evolving system can buy accuracy with cost, brittleness, hidden regressions, or safety risk. Conversely, reporting only cost or safety can obscure real capability gains. We therefore recommend evaluating self-evolution systems along six dimensions: correctness, efficiency, transfer, robustness, safety, and cost. These dimensions should be reported for both the final evolved system and the process that produced it.

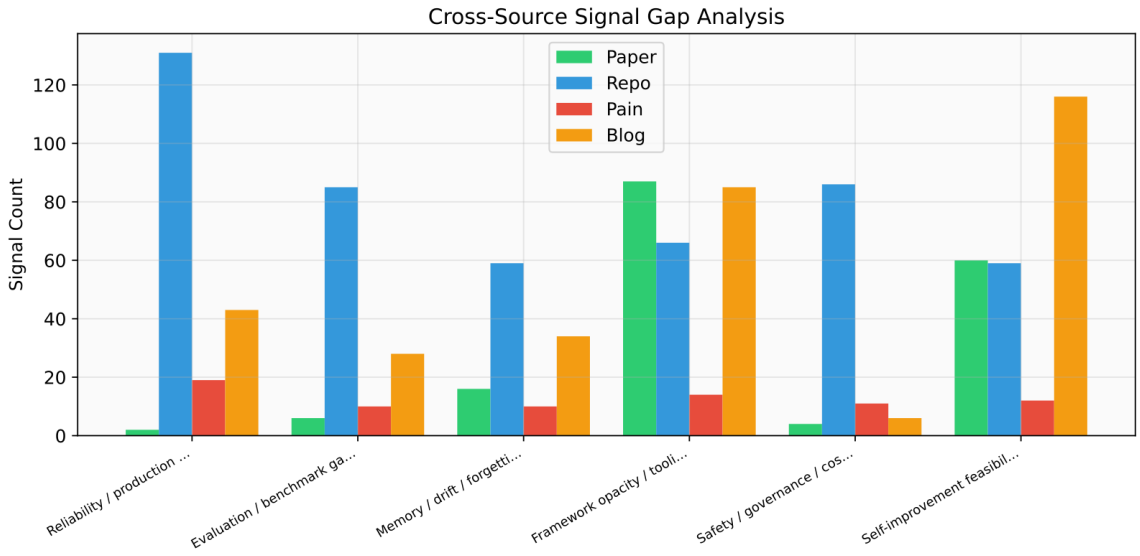

Figure 17: Cross-source validation gap. The survey distinguishes paper claims, open-source implementation evidence, user pain points, and public propagation signals so benchmark gains do not become the only evidence channel.

Correctness. Correctness measures whether outputs satisfy task specifications. In code generation, this usually means passing visible or hidden tests. In mathematical reasoning, it can mean exact-answer match, formal proof verification, or competition-style scoring. In interactive environments, it can mean task success, reward, or normalized progress. Correctness is the most common metric because it is simple and comparable, but it is also the easiest to overfit. For self-evolving systems, correctness should be separated into at least three levels: initial correctness before self-evolution, post-evolution correctness on validation tasks used for selection, and held-out correctness on tasks not used during search. The gap between validation and held-out correctness is a direct measure of overfitting to the evaluator.

Efficiency. Efficiency measures how many steps, tool calls, iterations, prompts, tests, environment interactions, and wall-clock seconds are required to reach a solution. Self-evolution methods often improve correctness by increasing inference-time computation. Reflexion permits multiple episodes; Self-Refine adds critique-and-rewrite cycles; ReVeal reports code evolution for more than twenty turns at inference time; DGM evaluates many candidate agent modifications; AlphaEvolve performs population-based search with many program evaluations. These additional computations may be scientifically justified, but they must be counted. A system that reaches 50% correctness with ten attempts should not be compared naively to a system that reaches 45% correctness with one attempt.

Transfer. Transfer measures whether improvements learned on one distribution help on another. Self-evolution is valuable only if it accumulates reusable capability rather than memorizing benchmark idiosyncrasies. Transfer can be across tasks, datasets, domains, programming languages, environments, model backbones, or time. DGM emphasizes transfer across software-engineering and polyglot coding tasks; ADAS reports that agent designs discovered in one domain can remain useful in another; Absolute Zero argues that code self-play can induce mathematical reasoning

transfer without external math data [21, 77]. Transfer should be evaluated by freezing the evolved artifact and testing it under a new evaluator, not by allowing continued adaptation on the target benchmark unless the adaptation budget is explicitly reported.

Robustness. Robustness measures stability under perturbations. For self-evolving systems, this includes prompt perturbations, random seeds, task paraphrases, hidden tests, tool failures, stale memory, environment changes, adversarial examples, and evaluator noise. A robust self-evolution method should not depend on a single lucky trajectory. Scores should therefore be reported with confidence intervals over seeds and task samples. More importantly, the process itself should be stress-tested: if one incorrect reflection is inserted into memory, does the agent recover? If tool output is delayed or malformed, does the workflow fail safely? If a benchmark problem is paraphrased, does the evolved strategy still work?

Safety. Safety measures whether the evolution process preserves constraints. In static benchmarks, unsafe behavior may be invisible because the task is narrow. In self-evolution, the system may modify prompts, memory, tools, or code. Safety metrics should therefore include permission violations, out-of-scope file edits, unapproved network calls, secret exposure, sandbox escapes, harmful instructions, evaluator tampering, and rollback failures. Safety should also include alignment with human oversight. An agent that silently changes its own evaluation script may improve a score while invalidating the experiment. An agent that refuses to ask for help because autonomy is rewarded may be worse in production even if benchmark success rises.

Cost. Cost is not the same as efficiency. Efficiency counts operational steps; cost monetizes or resource-normalizes them. Self-evolution systems consume tokens, API calls, GPUs, interpreter executions, browser sessions, human review minutes, storage, and maintenance labor. The field should report cost per candidate, cost per accepted improvement, cost per percentage point of gain, and total cost of search. AI Scientist reports approximately $15 per generated paper in its initial setting, while AlphaEvolve-style methods imply substantially larger evaluator budgets for large-scale scientific and infrastructure discovery [35, 40]. Without cost reporting, benchmark gains can be misleading because they hide the resource scale required to reproduce them.

Table 11 summarizes the six dimensions and concrete metrics for self-evolving systems.

A rigorous evaluation should also distinguish *artifact evaluation* from *process evaluation*. Artifact evaluation asks whether the final evolved system is good. Process evaluation asks whether the path to that system was reliable, efficient, safe, and reproducible. Many self-evolution papers emphasize artifact gains, but process metrics are equally important. If a method produces one excellent agent after hundreds of failed or unsafe variants, that is different from a method that produces smaller but reliable improvements. In deployment, the distribution of failed candidates matters because failed candidates consume resources and may create risk.

Five pitfalls recur across current evaluations.

Pitfall 1: benchmark overfitting and contamination. The most visible failure mode is overfitting to a public benchmark. Code agents can learn to satisfy common HumanEval patterns; math agents can exploit answer-format regularities; web agents can memorize benchmark page structures; retrieval agents can include benchmark data in context. Self-evolution increases this risk because the system may evaluate many candidates against the same task distribution. Contamination should be tested with time-split datasets such as LiveCodeBench, hidden unit tests such as

Table 11: Six evaluation dimensions for AI self-evolution. Each dimension should be reported for the final artifact and for the evolution process that produced it.

Dimension	Core question	Representative metrics
Correctness	Does the evolved system solve the task specification?	pass@1, pass@k, exact match, verified issue-resolution rate, theorem proof rate, environment success rate, hidden-test score, judge agreement.
Efficiency	How much computation or interaction is needed?	Iterations to success, tool calls, generated candidates, environment steps, wall-clock time, evaluator calls, retry count, context length.
Transfer	Does improvement generalize beyond the search distribution?	Cross-domain score, cross-language score, frozen-agent target score, train–validation–test gap, model-backbone transfer, time-split performance.
Robustness	Is the improvement stable under perturbation?	Seed variance, confidence intervals, adversarial success rate, prompt sensitivity, tool-failure recovery, memory-noise sensitivity, stale-environment performance.
Safety	Does evolution preserve constraints and oversight?	Permission violations, sandbox incidents, evaluator tampering attempts, harmful outputs, rollback success, audit completeness, human-escalation precision.
Cost	Is the gain worth the resources?	Dollars per solved task, tokens per gain point, GPU hours, human review minutes, accepted-improvement cost, storage/archive growth, maintenance burden.

SWE-bench Verified, or private held-out tasks. Researchers should report whether tasks were used during prompt design, tool tuning, memory construction, or archive selection.

Pitfall 2: conflating attempts with capability. A multi-turn or pass@k result is not directly comparable to a single-shot result. If an agent is allowed to generate, test, critique, and revise multiple times, the reported score reflects a search process. That is not a flaw; self-evolution is often intentionally a search process. The flaw is failing to report the search budget. Tables should include turns, samples, candidates, tests, and evaluator calls. ReVeal's result is meaningful precisely because the paper reports that inference can extend to more than twenty turns despite training on three-turn trajectories. Without that turn budget, the score would be uninterpretable.

Pitfall 3: evaluator coupling. Many systems use related model families for generation, critique, and judging. This can produce self-confirming errors: the generator produces a plausible but wrong solution; the critic shares the same blind spot; the judge rewards the same style. Executable evaluators reduce this risk in coding and formal proof tasks, but even tests and theorem provers can be incomplete or misused. A robust evaluation should use heterogeneous evaluators when possible: hidden tests, independent judge models, formal verification, human review, and environment-level outcome checks.

Pitfall 4: ignoring regressions. Self-evolution papers often report the best improved score but not the capabilities lost along the way. A prompt update that improves MATH may reduce

instruction following; a memory rule that helps one user may harm another; a code modification that solves one class of bugs may break another. Regression suites should include previous tasks, safety tests, cost tests, and out-of-domain checks. DGM-style archives help because they preserve multiple lineages, but the archive must record evaluator versions and negative results as well as winners.

Pitfall 5: missing cost and safety accounting. A reported gain without cost can be operationally meaningless. A gain without safety accounting can be dangerous. The field should normalize scores by tokens, dollars, wall-clock time, and human review. It should also report whether the system had write access, network access, code execution, persistent memory, or permission to modify its evaluator. These capabilities define the risk profile of the experiment. A self-evolution result obtained in a locked sandbox is not equivalent to one obtained by an autonomous production agent with broad permissions.

5.2 Code Generation Benchmarks

Code generation is the strongest empirical domain for AI self-evolution because it provides cheap, executable feedback. A program either passes tests or it does not; a traceback can be converted into a targeted revision signal; hidden tests can reduce prompt overfitting; and repositories provide realistic contexts for agentic software engineering. The main benchmark families are HumanEval, MBPP, DS-1000, SWE-bench, and LiveCodeBench.

HumanEval is a unit-test benchmark of Python function synthesis. It is small, widely used, and easy to run, making it useful for early comparisons but vulnerable to contamination and saturation [9]. MBPP contains many short programming problems oriented toward basic Python programming [4]. DS-1000 evaluates data-science code involving APIs such as NumPy, pandas, SciPy, scikit-learn, and PyTorch, making it more sensitive to library knowledge and runtime debugging [29]. SWE-bench moves from function synthesis to real GitHub issue resolution: the agent must edit a repository so that tests associated with an issue pass [26]. LiveCodeBench uses time-split competitive-programming tasks to reduce contamination and to measure contemporary coding ability [23]. For self-evolution, these benchmarks form an increasing ladder: single function, basic programming, API-heavy data science, repository-level repair, and time-sensitive contest coding.

Table 12 describes the evaluation target of each benchmark. The important methodological point is that the benchmark determines what kind of self-evolution is rewarded. HumanEval rewards concise functional correctness. DS-1000 rewards API repair and execution-guided debugging. SWE-bench rewards repository navigation, patch minimality, and regression awareness. LiveCodeBench rewards algorithmic reasoning under contamination control. A self-evolution method that performs well on one benchmark may not transfer to another.

Exact reported scores from the source materials are summarized in Table 13. Several patterns are visible. First, execution feedback is powerful: SelfEvolve raises DS-1000 pass@1 from 49.4 to 57.2 and HumanEval pass@1 from 74.39 to 85.98 by combining self-generated knowledge with iterative self-debugging. Second, reflection can be very strong in small code benchmarks: Reflexion reports 91.0% pass@1 on HumanEval with GPT-3.5, exceeding the cited GPT-4 zero-shot baseline of 80.1%. Third, architecture or agent evolution changes the evaluation scale: DGM reports SWE-bench improvement from 20.0% to 50.0% verified and Polyglot coding improvement from 14.2% to 30.7%. Fourth, verifier-aware multi-turn RL extends the inference-time horizon: ReVeal reports LiveCodeBench pass@1 increasing from 36.9 at the first turn to 42.4 at turn 19.

The Reflexion result illustrates the promise and risk of iterative self-correction. On HumanEval, the agent receives feedback after failed attempts and stores reflection in memory. The reported

Table 12: Major code-generation benchmarks used in self-evolution research.

Benchmark	**Unit of evaluation**	**What it rewards**	**Main limitation**
HumanEval	Python function with tests	Functional synthesis, docstring understanding, simple algorithmic reasoning	Small and heavily studied; public tasks risk contamination.
MBPP	Short Python programming problem	Basic programming, common library usage, natural-language specification following	Short tasks do not measure repository-level agency.
DS-1000	Data-science API problem	API knowledge, execution repair, library semantics, traceback use	Domain-specific; success can depend on installed package versions.
SWE-bench / Verified	Real repository issue	Multi-file editing, issue comprehension, testing, patch production	Expensive; environment setup and flaky tests complicate comparison.
LiveCodeBench	Time-split contest coding	Contemporary algorithmic reasoning, pass@k under lower contamination	Still mostly single-problem coding rather than long-lived software maintenance.

Benchmark slice	Baseline	Self-evolved	Gain
SelfEvolve DS1000	49.4	57.2	+7.8 pp
DGM SWE-bench verified	20.0	50.0	+30.0 pp
DGM Polyglot	14.2	30.7	+16.5 pp
ReVeal LiveCodeBench	36.9	42.4	+5.5 pp

Figure 18: Representative code benchmark gains. The figure intentionally mixes benchmark families only to visualize gain magnitudes; the underlying tasks are not directly comparable.

91.0% pass@1 is striking, but the result should not be interpreted as a pure single-shot model capability. It is an agent-loop capability: generation, evaluation, reflection, memory, and retry. That distinction matters because a deployed coding system may care about final correctness, while a model-comparison leaderboard may care about first-attempt synthesis. Reflexion is therefore best read as evidence that verbal feedback can convert failed attempts into useful future context, not as evidence that the underlying model's raw coding ability changed.

SelfEvolve provides a different evaluation lesson. DS-1000 problems often fail because of API misuse, missing imports, shape errors, or library-version assumptions. SelfEvolve first asks the model to generate knowledge about the needed APIs and then executes candidate code in a sandbox. Error messages become revision signals. This makes DS-1000 a natural benchmark because the environment can produce informative feedback. The method's DS-1000 improvement from 49.4 to 57.2 is therefore not just a score; it demonstrates that executable traces can serve as a localized curriculum. The HumanEval pass@1 increase from 74.39 to 85.98 shows that the same broad principle transfers to simpler function synthesis, although GPT-4's 89.95 pass@1 baseline remains higher in the reported table.

DGM changes the unit of evaluation from a generated answer to an agent implementation. The system maintains an archive of agent variants, samples parent agents, asks a foundation model to modify agent code, evaluates the child on benchmarks, and retains improvements. The reported

Table 13: Reported code-generation results for self-evolution methods. Scores are taken from the local paper summaries and original-paper reports available in the research corpus.

Method	Benchmark	Baseline	Self-evolution result	Gain
Reflexion	HumanEval pass@1	GPT-4 zero-shot: 80.1%; Codex fine-tuned: 67.0%	91.0% with GPT-3.5 Reflexion	+10.9 pp over GPT-4 baseline
SelfEvolve	DS-1000 pass@1	ChatGPT: 49.4; no-refine SelfEvolve: 52.9	57.2	+7.8 points over ChatGPT
SelfEvolve	HumanEval pass@1 / pass@10	ChatGPT: 74.39 / 81.10; GPT-4 pass@1: 89.95	85.98 / 87.81	+11.59 pp pass@1 over ChatGPT
SelfEvolve	TransCoder C++–Python	ChatGPT accuracy/pass@1: 88.3 / 88.5	90.4 / 90.9	+2.1 / +2.4 points
Agent Symbolic Learning	HumanEval pass@1	Agent baseline: 68.3%	73.2%	+4.9 pp
Darwin Godel Machine	SWE-bench verified	Seed agent: 20.0%	50.0%	+30.0 pp
Darwin Godel Machine	Polyglot coding score	Seed agent: 14.2%	30.7%	+16.5 pp
ReVeal	LiveCodeBench pass@1	Turn 1: 36.9	Turn 19: 42.4	+5.5 pp over inference turns

SWE-bench verified gain from 20.0% to 50.0% is therefore a process result over an evolving population. It should be evaluated not only by the final percentage but also by archive diversity, number of evaluated variants, failed modifications, compute budget, and transfer. The same method's Polyglot score increase from 14.2% to 30.7% suggests that the evolved agent improvements are not purely SWE-bench-specific, but transfer should be tested under frozen-agent conditions and with evaluator versioning.

ReVeal targets a weakness of many coding agents: they generate code more readily than they verify it. Its TAPO objective assigns credit at the turn level and explicitly optimizes self-verification. The LiveCodeBench result of 36.9 at turn 1 rising to 42.4 at turn 19 is important because it measures improvement across inference-time evolution rather than only across training updates. However, the number also demonstrates why efficiency reporting is essential. A turn-19 result has consumed much more computation than a turn-1 result. The right comparison depends on the application. For high-stakes code repair, a 5.5 point gain may justify many verifier turns. For latency-sensitive code completion, it may not.

SWE-bench and LiveCodeBench also expose two complementary threats. SWE-bench evaluates real repositories but can be sensitive to environment setup, flaky tests, and issue selection. LiveCodeBench reduces contamination through time-split problems but remains closer to contest programming than to long-term software maintenance. A strong code-evolution paper should therefore report both repository-level and time-split coding performance when possible. It should also include patch-quality metrics: number of files changed, test additions, dependency changes, lint errors, maintainability, and whether the patch solves the intended issue rather than merely satisfying tests.

For future code benchmarks, we recommend four additions. First, every self-evolution result should include a search budget: number of samples, turns, tests, tool calls, and wall-clock time. Second, benchmark harnesses should log whether generated tests were used for selection, because self-generated tests can become a source of Goodhart effects. Third, repository-level benchmarks should include unrelated regression tests and static analysis to detect broad damage. Fourth, benchmark reports should include negative candidate statistics: how often self-modifications failed

to parse, failed tests, touched forbidden files, or reduced performance. Without these process statistics, code self-evolution cannot be compared scientifically.

5.3 Mathematical Reasoning

Mathematical reasoning is a second major evaluation domain because it offers crisp answers, rich reasoning traces, and, in formal settings, machine-checkable proofs. The common benchmarks are GSM8K, MATH, MiniF2F, and competition-math evaluations. GSM8K contains grade-school word problems with exact numeric answers [14]. MATH contains more difficult high-school competition-style problems across algebra, geometry, number theory, counting, and precalculus [19]. MiniF2F formalizes olympiad-level theorem statements across proof-assistant ecosystems and measures whether a system can produce a verified proof [78]. Competition-math evaluations include International Mathematical Olympiad (IMO) style scoring, where a system may be graded by solved problems or medal-equivalent point totals.

For self-evolution, mathematics has three attractive properties. First, many tasks have verifiable final answers, allowing reinforcement learning with verifiable rewards. Second, reasoning traces can be critiqued, revised, and distilled, making math a natural domain for self-refinement. Third, formal theorem proving provides a strong evaluator: a proof either checks or it does not. The difficulty is that informal math answers can be correct for the wrong reason, formalization can become the bottleneck, and public benchmarks have been heavily optimized by recent models.

Table 14 summarizes reported numbers in the local corpus and related benchmark references. Self-Refine reports a math-reasoning accuracy increase from 56% to 67%. Agent Symbolic Learning reports MATH accuracy of 60.7% with a GPT-4 backbone, compared with 56.0% for the agent baseline and 48.4% for DSPy in the same extracted table. RISE reports approximate GSM8K improvements from 14% to 24% for Llama2-7B and from 38% to 46% for Mistral-7B through learned recursive introspection. MiniF2F's original Lean GPT-f/PACT reference point is around 24.6% pass@1 and 29.2% pass@32 on the test split, while later expert-iteration and neural theorem-proving systems report higher cumulative pass rates. Competition-math systems such as AlphaGeometry and AlphaProof are not self-evolution systems in the narrow agentic sense, but they are relevant evaluation references because they use search, verification, and synthetic-data loops; AlphaGeometry solved 25 of 30 olympiad geometry problems, and the AlphaProof/AlphaGeometry combination solved 4 of 6 IMO 2024 problems, corresponding to a silver-medal-level result in public reporting [17, 53].

The evaluation interpretation differs across the four benchmark families. GSM8K is useful for measuring arithmetic word-problem reliability, but it is vulnerable to answer-only shortcuts and contamination. MATH is harder and more diverse, but exact-answer grading can miss reasoning quality and can reward lucky final answers. MiniF2F is stricter because proof assistants verify the proof object, yet it moves part of the difficulty into formalization and search over proof tactics. IMO-style competition math is more meaningful to humans, but it is expensive to grade, small in sample size, and often not reproducible as a standardized benchmark.

Self-Refine's math result demonstrates that critique-and-rewrite can improve reasoning without weight updates. The method generates an answer, critiques its own output, and refines it. The increase from 56% to 67% shows that self-feedback can correct some errors, but the limitation is clear: if the model cannot identify the underlying mathematical mistake, it may refine a wrong solution into a more fluent wrong solution. This is why math evaluation should include intermediate checks when possible. A final answer can be correct by accident; a proof trace, symbolic derivation, or executable verification step provides stronger evidence.

RISE addresses a related issue by training the model to perform recursive introspection as a

Table 14: Mathematical reasoning benchmark results relevant to self-evolution. Approximate values are marked with "∼" when the source report gives rounded figures.

Method / reference	Benchmark	Baseline	Reported result	Metric
Self-Refine	Math reasoning task	56%	67%	Accuracy
Agent Symbolic Learning	MATH	Agents: 56.0%; DSPy: 48.4%	60.7%	Accuracy
RISE	GSM8K, Llama2-7B	∼14% single-turn	∼24% multi-turn introspection	Accuracy
RISE	GSM8K, Mistral-7B	∼38% single-turn	∼46% multi-turn introspection	Accuracy
MiniF2F reference	MiniF2F-test, Lean GPT-f/PACT	–	24.6% pass@1; 29.2% pass@32	Verified proof rate
Expert iteration reference	MiniF2F-test	Prior GPT-f/PACT baseline	29.6% pass@1; 36.6% pass@64	Verified proof rate
AlphaGeometry	Olympiad geometry set	Previous symbolic/neural systems	25 / 30 problems	Solved problems
AlphaProof + AlphaGeometry	IMO 2024	Human competition scoring	4 / 6 problems solved	Silver-level performance

learned policy rather than as a prompt-only behavior. Its reported GSM8K gains for Llama2-7B and Mistral-7B indicate that the ability to decide when to revise can itself be optimized. The evaluation lesson is that multi-turn reasoning should be treated as a policy over compute allocation. A model that spends more introspection turns on difficult problems and fewer turns on easy problems may be more efficient than one with a fixed number of refinement rounds. Future math benchmarks should therefore report accuracy as a function of reasoning budget, not only a single final score.

Agent Symbolic Learning provides a bridge between math reasoning and agent evaluation. Its MATH result is not just a chain-of-thought prompt; it is the outcome of a symbolic optimization process over prompts, tools, and pipeline components. The extracted table reports 60.7% accuracy with a GPT-4 backbone, above both the agent baseline and DSPy. This suggests that language-gradient updates can improve mathematical reasoning workflows. However, the comparison also raises a methodological concern: systems that optimize prompts on a benchmark must be evaluated on held-out tasks to distinguish true reasoning improvement from benchmark-specific prompt tuning.

MiniF2F and formal theorem proving are especially important for the next generation of self-evolution. A formal verifier provides a binary reward that is much harder to fool than a language-model judge. This makes theorem proving attractive for self-play, curriculum learning, and expert iteration. At the same time, formal proof systems expose a different kind of Goodhart risk: a prover may specialize in tactic patterns or formal-library artifacts rather than mathematical insight. Therefore, formal-math evaluation should report not only pass rates but also proof length, verifier calls, generated tactic diversity, dependence on imported lemmas, and performance on newly formalized problems.

Competition-math evaluation introduces an additional dimension: novelty and human-level problem solving. AlphaGeometry's 25/30 olympiad-geometry result and the AlphaProof and AlphaGeometry result on four of six IMO 2024 problems show that search-and-verification systems can reach meaningful mathematical milestones. For the self-evolution literature, the significance is not that every agent should be evaluated on IMO problems. The significance is that mathe-

matical discovery systems require evaluators that can certify nontrivial reasoning. If an evolving system claims to discover new lemmas, algorithms, or conjectures, it should provide proof artifacts, independent verification, or human-review protocols that go beyond answer matching.

A strong mathematical evaluation protocol should therefore include: (i) exact-answer benchmarks such as GSM8K for breadth; (ii) harder symbolic benchmarks such as MATH for depth; (iii) formal proof benchmarks such as MiniF2F for verification; (iv) time-split or newly generated tasks for contamination control; (v) compute-budget curves; and (vi) error taxonomy. Errors should be categorized as arithmetic slips, algebraic transformations, missing cases, false lemmas, invalid formalization, search failure, or evaluator mismatch. Self-evolution systems should then show which error categories decrease after evolution. A single aggregate accuracy hides whether the system learned mathematics or merely learned benchmark style.

5.4 Agent Benchmarks

Agent benchmarks evaluate systems that act over multiple steps, often with tools, environments, memory, or external observations. They are crucial for AI self-evolution because many evolving systems are agents rather than pure text generators. The benchmark families considered here are HotPotQA for multi-hop question answering, ALFWorld for text-based embodied household tasks, WebArena for browser-based web tasks, and Minecraft environments such as Voyager for open-ended embodied skill acquisition.

HotPotQA evaluates multi-hop reasoning over documents and commonly reports exact match (EM) and F1 [68]. It is useful for testing whether an agent can decompose a question, retrieve or inspect evidence, and combine facts. ALFWorld maps household embodied tasks into a text environment where agents must issue actions to complete goals such as finding, moving, cleaning, heating, or cooling objects [50]. WebArena evaluates autonomous web agents across realistic websites such as shopping, forums, content management, and maps, with success measured by task completion [79]. Voyager evaluates an open-ended Minecraft agent that writes, stores, and reuses skills, measuring exploration, technology-tree progress, and skill-library growth [55].

The main difference from code and math benchmarks is that agent benchmarks evaluate trajectories. The final answer may be less informative than the path. Did the agent use the right tool? Did it recover from errors? Did it take unnecessary actions? Did it persist unsafe memory? Did it overfit to a simulator? Did it transfer learned skills? For self-evolution, these questions are central because the evolving object may be the policy, prompt, tool-use pattern, memory, or skill library.

Table 15 summarizes exact reported numbers from the local corpus and widely cited benchmark reports. Agent Symbolic Learning reports HotPotQA F1/EM improvements from a baseline of 36.2/22.8 to 39.1/25.6, with DSPy at 37.4/24.1 in the extracted comparison. Reflexion reports ALFWorld success of 97% versus a 75% base success rate. WebArena's original benchmark showed that contemporary GPT-4 agents remained far below human performance, with a commonly cited GPT-4/ReAct-style success rate around 14.41% compared with substantially higher human task completion. Voyager reports large open-ended Minecraft gains, including discovering 3.3 times more unique items, traveling 2.3 times longer distances, and unlocking key technology-tree milestones much faster than prior agents in its evaluation setting.

HotPotQA is a useful testbed for prompt and workflow evolution because errors often arise from poor decomposition or retrieval rather than missing world knowledge. Agent Symbolic Learning treats the agent as a symbolic network of prompts, tools, and pipeline connections. The evaluator constructs language losses from trajectories and propagates language gradients back through components. The HotPotQA gain from F1 36.2 to 39.1 and EM 22.8 to 25.6 is modest but meaningful because it indicates that structured prompt and pipeline updates can improve multi-hop behavior.

Table 15: Agent benchmark results relevant to self-evolving systems. WebArena and Voyager numbers are included as environment references because they are central benchmarks for multi-step agent evaluation.

Method	Benchmark	Baseline	Reported result	Metric
Agent Symbolic Learning	HotPotQA	Agents: F1 36.2, EM 22.8; DSPy: F1 37.4, EM 24.1	F1 39.1, EM 25.6	F1 / EM
Reflexion	ALFWorld	Base agent: 75%	97%	Success rate
Godel Agent	HotPotQA / ALFWorld	Hand-designed agent baseline	Reported to exceed baselines; local extraction gives +8–15% on HotPotQA and significant ALFWorld gains	Relative gain
WebArena reference	WebArena	Human performance far higher; early GPT-4 agents low	GPT-4/ReAct-style agents around 14.41% in original benchmark reporting	Success rate
Voyager reference	Minecraft	Prior Minecraft agents	3.3× unique items, 2.3× travel distance, faster tech-tree unlocking	Exploration and skill metrics

The limitation is that HotPotQA is still mostly a question-answering benchmark; it does not fully test tool safety, long-term memory, or environment change.

ALFWorld adds sequential decision-making. Reflexion's 97% success rate versus a 75% base agent shows the value of memory-based verbal reinforcement in an environment where the agent can learn from failed episodes. The evaluation lesson is that episodic feedback can be highly effective when tasks repeat structural patterns. However, this also raises a transfer question: does the reflection memory capture general household strategies, or does it memorize environment-specific action patterns? A robust ALFWorld evaluation should include unseen task templates, randomized object locations, and memory ablation.

WebArena is important because it exposes the weakness of many LLM agents in realistic browser settings. Early GPT-4-based agents achieved only low double-digit success rates, around 14.41% in the original benchmark setting, despite strong language ability. The gap reflects long-horizon planning, UI grounding, authentication state, form filling, error recovery, and website-specific affordances. For self-evolution, WebArena suggests that improving prompts alone may be insufficient. Systems may need better state representations, browser skill libraries, visual grounding, tool abstractions, and safe memory. Evaluation should report not only success but also invalid actions, page-navigation loops, timeouts, and whether the agent caused irreversible side effects.

Voyager represents a different kind of agent benchmark: open-ended skill acquisition rather than fixed task completion. The agent explores Minecraft, writes executable skills, stores them in a skill library, and reuses them for future tasks. Its reported gains of 3.3× unique items and 2.3× longer travel distance are not simple accuracy numbers. They measure coverage and exploration. This matters for self-evolution because open-ended agents should be evaluated by the growth of useful behavior, not only by success on a fixed checklist. At the same time, open-ended evaluation is hard: novelty can be shallow, exploration can be unsafe, and skill libraries can accumulate brittle or redundant code. Good evaluation should include skill reuse rate, skill correctness, dependency structure, and performance when the library is transferred to new worlds.

Agent benchmarks also need trajectory-level metrics. We recommend reporting: number of actions, invalid-action rate, recovery rate after failed tool calls, repeated-action loops, memory reads/writes, tool-call cost, human interventions, safety violations, and final success. For self-evolving agents, one should additionally report what changed between versions: prompt diff, mem-

ory diff, workflow diff, tool diff, or code diff. Without such diffs, a final success rate does not tell us what the system learned.

A particularly important evaluation issue is evaluator granularity. Binary task success can be too sparse for self-evolution. If an ALFWorld agent fails after twenty steps, the system needs to know whether it misunderstood the goal, chose the wrong object, failed to navigate, used the wrong action, or stopped too early. If a WebArena agent fails, the reason may be visual grounding, form validation, page state, login, or missing information. Process-oriented metrics convert failures into learning signals. This is why agent benchmarks should publish structured traces and failure taxonomies, not only final scores.

5.5 Open-Ended Evaluation

Open-ended self-evolution aims to discover artifacts that are not specified in advance: new algorithms, improved programs, novel proofs, better heuristics, or unexpected agent architectures. This is the evaluation regime of FunSearch, AlphaEvolve, DGM, ADAS, and AI Scientist-like systems. It is the most exciting regime because it moves beyond solving known tasks. It is also the hardest to evaluate because the evaluator must distinguish genuine novelty from overfitting, rediscovery, or benchmark gaming.

FunSearch uses a language model to generate programs and an evaluator to score them, iterating over a database of high-scoring candidate programs. It demonstrated new constructions for the cap set problem and improved heuristics for online bin packing [47]. AlphaEvolve generalizes the idea to larger codebases and combines Gemini Flash for broad exploration with Gemini Pro for deeper candidate generation, using an evolutionary database and quality-diversity search. The local paper summary reports several headline results: a 4×4 complex matrix multiplication algorithm using 48 multiplications, described as the first improvement over Strassen-style results in 56 years for that setting; rediscovery of state-of-the-art solutions for 75% of more than 50 mathematical problems and improvement beyond known best results for 20%; recovery of 0.7% of worldwide compute in Google's Borg scheduling context; a 23% speedup in one LLM-training kernel-tiling setting; and a 32% speedup in another attention-related optimization setting.

The key evaluation challenge in open-ended discovery is novelty. A system may rediscover a known result, improve a benchmark-specific heuristic, or find a genuine new contribution. These outcomes should be reported separately. Rediscovery is valuable because it validates the search process, but it is not the same as discovery. Improvement on a synthetic evaluator is valuable if the evaluator matches the real problem, but it may not transfer. Genuine discovery requires independent verification, comparison with the literature, and often expert review.

AlphaEvolve's results show why open-ended evaluation must include domain-specific validators. Matrix multiplication improvements require algebraic correctness, not judge-model preference. Scheduling improvements require production-like simulators or shadow deployments, not a language explanation. Kernel speedups require hardware-specific runtime measurements with noise control. Mathematical-problem improvements require proof or exhaustive verification. A single generic benchmark cannot evaluate all these outputs. Open-ended self-evolution therefore needs evaluator portfolios: each domain must define a reliable scoring function, a novelty check, and an independent validation path.

FunSearch also demonstrates the importance of interpretable artifacts. The generated object is a program, not just a number. Humans can inspect the program, simplify it, connect it to known theory, and sometimes extract a mathematical insight. This is a strength of program-search approaches. If an LLM only outputs a final score or opaque solution, discovery is harder to validate. For self-evolution, executable and inspectable artifacts should be preferred because they support

Table 16: Open-ended discovery results from FunSearch and AlphaEvolve-style systems. These results require evaluators that certify candidate quality rather than merely compare against a fixed answer key.

System	Domain	Reported result	Evaluator
FunSearch	Cap set / extremal combinatorics	Discovered new high-scoring constructions beyond previously known programmatic baselines	Objective combinatorial score
FunSearch	Online bin packing	Found improved heuristics for bin-packing variants	Simulation score
AlphaEvolve	4×4 complex matrix multiplication	48 multiplications; reported as first improvement in 56 years for the target setting	Algebraic verification / program evaluator
AlphaEvolve	50+ mathematical problems	Re-discovered SOTA for 75%; improved best known for 20%	Problem-specific evaluators
AlphaEvolve	Borg scheduling	Recovered 0.7% of worldwide compute	Production scheduling evaluator
AlphaEvolve	LLM training kernels	23% speedup in kernel tiling; 32% speedup in another attention optimization	Runtime benchmark
AI Scientist v2	Automated paper generation	Peer-review score 6.33; top 45% workshop-level ranking	Human/LLM peer review

verification, reuse, and scientific understanding.

Open-ended evaluation should include archive metrics. A population or database is not merely an implementation detail; it is part of the system's intelligence. Useful metrics include number of unique niches filled, best score over time, median score over time, novelty of accepted candidates, lineage depth, proportion of candidates that compile or verify, rediscovery rate, and diversity of strategies. Quality-diversity methods such as MAP-Elites explicitly optimize a map of elites across behavioral descriptors. Reporting only the single best artifact discards much of the evidence about the search process.

Open-ended systems also require negative-result reporting. If AlphaEvolve improves 20% of more than 50 mathematical problems, what happened on the remaining problems? Did it fail to generate valid candidates, rediscover known solutions, overfit the evaluator, or find no improvement? If FunSearch works on cap sets and bin packing, what domains did not work? Failure taxonomy is essential because open-ended discovery is expensive. Practitioners need to know which evaluator properties make a domain suitable: cheap scoring, dense feedback, reliable verification, decomposable artifacts, and enough room for improvement.

AI Scientist illustrates a different open-ended evaluation problem: evaluating scientific artifacts rather than programs. The local corpus reports that AI Scientist v1 generated papers across diffusion models, language modeling, and grokking at approximately $15 per paper, while AI Scientist v2 achieved a peer-review score of 6.33 and top-45% workshop-level ranking. These numbers are important, but peer review is noisy and can be gamed by fluent writing. Independent evaluation found limitations such as rejection of human-accepted papers, weak generalization, limited theoretical depth, and restricted architectural diversity. This case shows that open-ended evaluation must combine automated reviewers with human experts, novelty search over literature, reproducibility checks, and artifact-level validation.

A good open-ended evaluation protocol should therefore require: (i) a precise objective or validator; (ii) an explicit novelty check against known results; (iii) reproducibility artifacts; (iv) archive

Project	Score	Project	Score	Project	Score
OpenEvolve	74	DSPy	73	SWE-Agent	70
OpenHands	68	LangGraph	67	AutoGen	66
FunSearch	62	CrewAI	62	Reflexion	58
MetaGPT	56	AutoGPT	28	Devika	30

Figure 19: Composite star-quality ranking. The figure shows that raw popularity and ecosystem quality can diverge sharply.

and search-budget statistics; (v) independent expert or formal verification for claimed discoveries; (vi) failure taxonomy; and (vii) a distinction between rediscovery, benchmark improvement, and new knowledge. Without these layers, open-ended self-evolution risks becoming a generator of impressive anecdotes rather than cumulative science.

5.6 Star Quality Analysis

Research benchmarks measure technical capability, but the Self Evolve ecosystem also needs to evaluate project quality. GitHub stars are a noisy signal: they measure attention, not necessarily reliability, maintainability, or research value. The star-analysis report in the source materials therefore proposes a quality scoring system that combines star activity, contributor diversity, fork quality, issue quality, and pull-request merge rate. This section incorporates that 12-project ranking and interprets it as an ecosystem-level evaluation tool.

The report's key conclusion is that raw stars can invert quality. AutoGPT has approximately 175,000 stars but receives a composite quality score of only 28/100. OpenEvolve has approximately 6,358 stars but receives the highest composite score, 74/100. DSPy, SWE-Agent, OpenHands, LangGraph, AutoGen, FunSearch, CrewAI, Reflexion, MetaGPT, AutoGPT, and Devika form the 12-project ranking shown in Table 17. The analysis identifies Stars/Contributor ratio as a particularly useful hype detector: below 50 is healthy, 50–150 warrants attention, and above 150 is high-risk. AutoGPT's Stars/Contributor ratio is reported as 213, while Devika's is 183, both above the suspicious threshold. CrewAI's 107 is in the watch range but supported by steadier growth and stronger community health.

The scoring formula is simple but useful:

$$\text{CompositeScore} = 0.20S_{activity} + 0.20C_{diversity} + 0.20F_{quality} + 0.20I_{quality} + 0.20P_{merge}. \quad (102)$$

Here $S_{activity}$ measures the share of recent active stargazers, $C_{diversity}$ reverses contributor concentration, $F_{quality}$ estimates the fraction of forks with meaningful commits, $I_{quality}$ measures non-spam issue quality, and P_{merge} is merged pull requests divided by total pull requests. The equal weighting is not definitive, but it is transparent. A project can no longer hide behind raw stars if forks are empty, issues are repetitive, contributors are concentrated, and pull requests are rarely merged.

The report reconstructs propagation chains that explain why raw stars can mislead. AutoGPT grew from creation on March 30, 2023 to approximately 50,000 stars by April 2 after Twitter/X demonstrations, Elon Musk amplification, Hacker News and Reddit exposure, YouTube tutorials, and translated media coverage. The report classifies this as hype-driven growth: fast rise, high Stars/Contributor ratio, many inactive stargazers, and repetitive issues. Devika shows a similar but more derivative pattern: it was framed as an open-source Devin alternative, reached around 20,000 stars within days, and then suffered steep activity decline. By contrast, DSPy, SWE-Agent, OpenEvolve, and FunSearch show slower academic or engineering-driven adoption.

Table 17: Twelve-project GitHub quality ranking from `reports/star-analysis-report.md`. The composite score averages star activity, contributor diversity, fork quality, issue quality, and PR merge rate.

Rank	Project	Stars	Score	Growth	Interpretation
1	OpenEvolve	6,358	74	organic	High-quality niche project
2	DSPy	25,000	73	organic	Strong academic and engineering signal
3	SWE-Agent	15,000	70	organic	ICLR Oral endorsement and strong benchmark focus
4	OpenHands	55,000	68	organic	Highest community-health profile among large projects
5	LangGraph	20,000	67	steady	Solid architecture and ecosystem fit
6	AutoGen	50,000	66	organic	Microsoft-backed multi-agent framework
7	FunSearch	1,500	62	organic	Under-starred but high-quality research project
8	CrewAI	30,000	62	steady	Practical community adoption with manageable hype risk
9	Reflexion	3,158	58	steady	Classic research implementation
10	MetaGPT	50,000	56	viral	Academic/commercial balance but media-amplified growth
11	AutoGPT	175,000	28	viral	Hype-driven; raw stars overstate quality
12	Devika	22,000	30	suspicious	Me-too growth pattern tied to Devin attention

Star quality matters for AI self-evolution because the field is tool-heavy. Practitioners often choose frameworks, agents, memory systems, evaluators, and code-evolution platforms based on GitHub visibility. If raw stars dominate selection, teams may adopt projects that are popular but poorly maintained. A quality-adjusted ranking helps distinguish durable infrastructure from social-media spikes. It also helps identify underappreciated research artifacts. FunSearch's 1,500 stars and score of 62 suggest that scientific value may be high even when community size is small. OpenEvolve's top score suggests that a focused implementation of AlphaEvolve-like ideas can be more trustworthy than a viral general-purpose agent.

The star-quality framework should be treated as a complement to technical benchmarks, not a replacement. A project can have excellent benchmark results and poor community health, or strong community health and weak research novelty. For production adoption, both matter. A self-evolution system should be evaluated by capability, maintainability, documentation, issue responsiveness, contributor diversity, security posture, and licensing. The star-quality score provides an initial signal for maintainability and community health.

There are limitations. GitHub API-derived measures can be noisy. Fork quality requires deeper analysis than fork count. Issue quality may require language-specific NLP and can be biased against non-English communities. PR merge rate can be low for healthy projects that use internal

development or strict review. Stars/Contributor ratio can penalize mature projects with large user bases. Therefore, the framework should not be used as a mechanical ranking. It should be used as an audit checklist: if raw stars are high but contributor diversity, fork quality, issue quality, and PR merge rate are low, investigate before adopting.

For future ecosystem reports, we recommend adding dependency health, release cadence, security advisories, test coverage, documentation freshness, benchmark reproducibility, and maintainer bus factor. Self-evolution projects are especially sensitive to evaluator and sandbox quality; a project with unsafe defaults can create real risk even if it is popular. Star quality should therefore be integrated with safety and reproducibility audits.

5.7 Process-Oriented Metrics

Static benchmark scores answer the question, "How good is the final system?" Process-oriented metrics answer the question, "How did the system improve, and can that improvement be trusted?" For self-evolution, process metrics are essential because the path is part of the method. A system that improves smoothly over many validated iterations is different from one that finds a lucky candidate after many failures. A system that maintains diverse archives is different from one that collapses to a brittle local optimum. A system whose improvement transfers is different from one that overfits to a single evaluator.

Iteration gain curves. The first process metric is the gain curve: score as a function of iteration, turn, candidate count, or cost. Let s_t be the validation or held-out score after iteration t. The marginal gain is

$$\Delta_t = s_t - s_{t-1}. \tag{103}$$

The cumulative gain is $s_T - s_0$, and the gain efficiency can be normalized by cost c_t:

$$\eta_T = \frac{s_T - s_0}{\sum_{t=1}^{T} c_t}. \tag{104}$$

Gain curves reveal saturation, instability, and regressions. For example, ReVeal's LiveCodeBench curve from turn 1 to turn 19 shows inference-time improvement but should be accompanied by per-turn cost. DGM's archive evolution should show best score, median score, and accepted-candidate count over iterations. Self-Refine-style systems should show whether most improvement occurs in the first refinement or continues across later rounds.

Regression curves. Improvement on a target benchmark may hide degradation elsewhere. Let R_t be a regression-suite score after update t. A safe update should satisfy both $s_t > s_{t-1}$ and $R_t \geq R_{t-1} - \epsilon$ for an acceptable tolerance ϵ. In production, regression suites should include safety tests, previous user tasks, cost limits, latency, and tool-permission tests. A candidate update that improves HumanEval while increasing unsafe file edits should be rejected.

Archive diversity. Population-based systems such as DGM and AlphaEvolve require archive metrics. If the archive contains n artifacts with behavioral descriptors b_i, one simple diversity metric is average pairwise distance:

$$D(A) = \frac{2}{n(n-1)} \sum_{i<j} d(b_i, b_j). \tag{105}$$

For code agents, descriptors may include tool-use patterns, edit strategies, prompt templates, test-generation behavior, or module-level changes. For program discovery, descriptors may include algorithmic structure, runtime profile, proof strategy, or mathematical construction family. Diversity matters because open-ended progress often depends on stepping stones that are not immediately best on the main objective.

Lineage and stepping-stone metrics. An archive should record parent-child relationships. Useful metrics include lineage depth, branching factor, number of ancestors contributing to the best artifact, and time between stepping stones. A best artifact that descends from many modest improvements provides different evidence than one found by random sampling. Lineage also supports interpretability: researchers can inspect which modifications mattered.

Transfer metrics. Let $S_{A\to A}$ be performance after evolving on source domain A and evaluating on A, and let $S_{A\to B}$ be performance after evolving on A but evaluating on target domain B without further adaptation. A simple transfer ratio is

$$T_{A\to B} = \frac{S_{A\to B} - S_{0\to B}}{S_{A\to A} - S_{0\to A} + \delta}, \tag{106}$$

where S_0 is the seed system and δ avoids division by zero. A high transfer ratio indicates reusable improvement. A low or negative ratio indicates overfitting or harmful specialization. Transfer should be measured across benchmarks, languages, task types, environments, model backbones, and time splits.

Evaluator sensitivity. Self-evolution can overfit to evaluator quirks. Evaluator sensitivity measures how rankings change under alternative evaluators. If candidate a_i is best under evaluator E_1 but poor under E_2, the improvement may be fragile. Researchers can report rank correlation between evaluators, disagreement rate, and score variance across evaluator seeds. For LLM judges, evaluator sensitivity should include different model families and prompt variants. For code tests, it should include hidden tests, mutation tests, and property-based tests.

Cost-normalized improvement. Cost-normalized metrics should be first-class. A practical score might be

$$\text{Utility} = \text{CorrectnessGain} - \lambda_1\text{Cost} - \lambda_2\text{Latency} - \lambda_3\text{Risk} - \lambda_4\text{HumanBurden}. \tag{107}$$

The coefficients depend on the deployment setting, but the structure is universal. An evolving system is useful when its marginal gain exceeds the marginal cost and risk. Papers should report enough raw data for readers to compute their own utility under different assumptions.

Memory quality metrics. For memory-based self-evolution, process metrics should include memory precision, memory recall, stale-memory rate, contradiction rate, retrieval utility, and memory write acceptance. A reflection that is never retrieved is wasted. A reflection that is retrieved in the wrong context is harmful. A memory store that grows without consolidation increases cost and drift risk. Longitudinal memory benchmarks should measure whether the agent improves over sessions without accumulating misleading lessons.

Safety-process metrics. Safety should also be process-oriented. Count candidate updates rejected for safety reasons, sandbox violations, attempts to access forbidden resources, evaluator modification attempts, rollback events, and human escalations. These metrics should not be hidden as embarrassing failures; they are evidence that the safety gate is doing work. A system with zero reported safety events may be safe, or it may simply lack instrumentation.

Process-oriented evaluation changes how results are interpreted. A final score is a point estimate. A process report is a trajectory. For self-evolution, the trajectory is often the scientific contribution. It shows whether improvement is monotonic, whether diversity is preserved, whether cost explodes, whether transfer emerges, and whether safety gates catch bad candidates. Future benchmarks should therefore require trace submissions, not just final answers.

5.8 Recommended Evaluation Protocol

We conclude with an eight-point checklist for evaluating AI self-evolution systems. The checklist is designed for papers, open-source projects, and production pilots. It is intentionally stricter than conventional model benchmarking because self-evolution introduces additional degrees of freedom.

1. Define the evolving object. State exactly what changes: output only, prompt, memory, tool description, workflow graph, code, archive, reward model, or model weights. Many ambiguities disappear when the evolving object is explicit. A method that revises an answer is different from a method that rewrites its own source code.

2. Freeze and document the evaluator. Record benchmark version, task split, hidden-test policy, judge prompts, environment versions, tool versions, random seeds, and scoring scripts. If the evaluator changes during the experiment, report evaluator lineage. The agent should not be able to modify the evaluator unless evaluator evolution is the object of study and is separately audited.

3. Report the adaptation budget. Include number of turns, samples, candidates, tool calls, tests, environment steps, verifier calls, tokens, wall-clock time, API cost, GPU time, and human review minutes. Separate training-time, validation-time, and inference-time budgets. Do not compare a turn-20 result to a single-shot baseline without budget normalization.

4. Use train–validation–test separation for evolution. Candidate updates may be selected on training or validation tasks, but final claims should use held-out tasks. For time-sensitive domains, use time splits. For repository tasks, use repositories not seen during prompt or workflow tuning. For memory systems, test on sessions not used for memory construction.

5. Measure regressions and safety constraints. Maintain a regression suite that includes previous capabilities, safety policies, cost limits, latency, forbidden actions, privacy constraints, and rollback tests. Reject updates that improve the main benchmark while violating constraints. Report rejected updates and reasons.

6. Evaluate transfer and robustness. Freeze the evolved artifact and test it on new domains, seeds, paraphrases, environments, tools, model backbones, or time periods. Report confidence intervals and sensitivity to evaluator variants. Include perturbation tests for memory, prompts, and tool failures.

Table 18: Eight-point protocol for evaluating self-evolution systems.

No.	Protocol item	Minimum evidence
1	Evolving object	Explicit artifact type and allowed update operators.
2	Frozen evaluator	Benchmark version, splits, scripts, judge prompts, environment, seeds.
3	Adaptation budget	Turns, candidates, samples, tools, tokens, wall-clock, dollars, human minutes.
4	Held-out testing	Train/validation/test or time-split separation; no final selection on test.
5	Regression and safety	Regression suite, forbidden actions, rollback, rejected update statistics.
6	Transfer and robustness	Cross-domain tests, perturbations, confidence intervals, evaluator sensitivity.
7	Process traces	Gain curves, archive metrics, lineage, negative candidates, trace schema.
8	Cost-normalized value	Gain per resource unit and operational utility analysis.

7. Publish process traces and archive statistics. For iterative systems, publish gain curves, candidate counts, accepted/rejected updates, archive diversity, lineage, and negative results. For privacy or safety reasons, traces may be redacted, but the structure should remain available. A self-evolution method without process evidence is hard to reproduce.

8. Normalize by cost and operational value. Report improvement per dollar, per token, per verifier call, per wall-clock hour, and per human review minute. Include maintenance burden and deployment constraints. A benchmark gain is not sufficient if the required cost, risk, or operational complexity exceeds the value of the improvement.

The checklist can be used as a scoring rubric. A minimal demonstration may satisfy only items 1–3. A publishable benchmark paper should satisfy items 1–7. A production-ready self-evolution system should satisfy all eight and add security review, access control, incident response, and monitoring. The key is not bureaucracy; it is epistemic hygiene. Self-evolution systems are optimization processes. Optimization processes exploit whatever signal they receive. Evaluation must therefore be designed as a robust control system, not as an afterthought.

Across the benchmarks reviewed in this chapter, the most reliable gains appear where feedback is executable, independent, and cheap: code tests, theorem provers, simulators, and domain-specific evaluators. The least reliable claims appear where evaluation depends on vague judge preference, raw popularity, or unreported search budgets. The field's next step is not to abandon benchmarks, but to make them harder to game and richer in process evidence. Correctness, efficiency, transfer, robustness, safety, and cost must be reported together. Only then can AI self-evolution move from impressive improvement anecdotes to cumulative, trustworthy science.

Extended benchmark interpretation notes. The checklist above is deliberately compact; in practice, each item should be translated into a reporting template before experiments begin. A useful template starts with a run card. The run card should identify the base model, decoding parameters, system prompt, tool set, memory state, benchmark split, evaluator version, and permitted update operators. It should then record every candidate update in a table: candidate identifier, parent identifier, edit type, validation score, held-out score if available, cost, failure reason if rejected, and safety notes. The run card is the experiment's provenance layer. Without it,

later readers cannot tell whether an improvement came from the algorithm, a prompt change, a model upgrade, a benchmark change, or a lucky sample.

A second template is the evaluator card. The evaluator card should explain what the evaluator can and cannot detect. For HumanEval and MBPP, it should describe visible tests, hidden tests if any, timeout policy, dependency versions, and whether generated tests were used during selection. For DS-1000, it should list library versions because pandas, NumPy, SciPy, and PyTorch behavior can change across releases. For SWE-bench, it should record repository checkout, issue identifier, environment build result, test selection, and whether the agent had access to problem-specific hints. For LiveCodeBench, it should record the date range of tasks, because time-split validity is part of the benchmark's purpose. For HotPotQA, it should state whether evidence documents are retrieved, provided, or searched through tools. For ALFWorld, it should state task splits and environment randomization. For WebArena, it should state website snapshots, authentication state, browser tooling, and whether destructive actions were disabled. For Minecraft or Voyager-like settings, it should record world seed, allowed commands, skill-library initialization, and simulator version.

A third template is the improvement card. The improvement card should answer four questions: what changed, why was it accepted, where does it transfer, and what did it break? For prompt-level evolution, the card should include prompt diffs and examples of changed behavior. For memory-level evolution, it should include newly written memories, consolidation rules, and retrieval statistics. For workflow-level evolution, it should include a graph diff showing added or removed nodes and edges. For code-level evolution, it should include a patch diff, tests run, and rollback plan. For archive-level evolution, it should include parent lineage and novelty descriptors. For model-weight updates, it should include training data provenance, hyperparameters, and regression-suite results. Treating every improvement as an auditable object makes self-evolution closer to software release engineering than to ad hoc prompting.

The most common reporting gap in current work is the absence of a denominator. A paper may report the best score achieved by an evolved system, but not how many candidates were tried, how many failed to parse, how many were rejected by tests, how many were rejected by safety gates, or how many consumed budget without improvement. The denominator matters because it determines reproducibility. A method that obtains one strong candidate after ten attempts is very different from a method that obtains one strong candidate after ten thousand attempts. Both may be scientifically interesting, but they answer different questions. The former suggests a reliable update operator; the latter may suggest that the search space contains rare valuable artifacts but requires large-scale filtering.

Another reporting gap is uncertainty. Self-evolution methods are often stochastic at multiple levels: model sampling, candidate selection, environment state, hidden-test construction, and evaluator judgment. A single run can exaggerate performance. At minimum, papers should report multiple seeds or bootstrap confidence intervals over tasks. When multiple seeds are too expensive, authors should report why and provide at least a sensitivity analysis over decoding temperature or candidate sampling. For archive methods, the distribution of final archive scores across runs is more informative than the best run. For multi-turn methods, per-turn variance indicates whether improvement is stable or driven by occasional recoveries.

Evaluation should also distinguish controllable and uncontrollable budget. In many deployed settings, the system can choose how many refinement turns to take, how many tools to call, and how many tests to generate. These are controllable budgets and should be part of the policy being evaluated. Other costs, such as environment setup or fixed benchmark overhead, may be uncontrollable. Reporting them separately prevents misleading conclusions. A coding agent that calls a compiler ten times has made a policy choice; a SWE-bench harness that spends minutes building dependencies has imposed an infrastructure cost. Both affect real deployment, but they

should be attributed differently.

The relationship between pass@k and self-evolution deserves special care. pass@k traditionally measures the probability that at least one of k independent samples is correct. A self-evolving system's k attempts are usually not independent: later attempts condition on earlier errors, reflections, tests, or memory. This can be more powerful than independent sampling, but it changes the meaning of the metric. Researchers should avoid describing multi-turn self-repair as ordinary pass@k unless the attempts are independent. A clearer notation is pass@$(k, \mathcal{F})$, where $\mathcal{F}$ describes the feedback available between attempts: no feedback, visible tests, hidden tests, traceback, judge critique, human feedback, formal verifier, or environment reward. This notation makes explicit that pass@10 with traceback is a different capability from pass@10 without feedback.

Similarly, exact-match accuracy in math should be interpreted alongside reasoning validation. A model may output the right integer with invalid reasoning, or it may produce a nearly correct proof with a small arithmetic error. For product use, final correctness may dominate. For scientific evaluation of self-evolution, the error path matters because the system is supposed to learn from it. Math benchmarks should therefore store trace-level annotations: invalid assumption, missing case, arithmetic error, algebraic manipulation error, theorem misuse, and answer-format error. A self-refinement method that reduces answer-format errors but not theorem misuse is improving surface compliance, not deep reasoning. A verifier-guided method that reduces invalid lemmas is closer to genuine mathematical progress.

For agent benchmarks, the equivalent distinction is between task success and policy quality. An agent may complete a WebArena task by chance after many irrelevant clicks, or fail a task after making a mostly reasonable plan that encounters a website error. Binary success hides both cases. A self-evolving web agent should be evaluated with trajectory diagnostics: state-observation quality, action validity, navigation efficiency, form-filling precision, recovery from server errors, and ability to ask for clarification. These diagnostics can become training signals for future evolution, but they must be kept separate from final test scoring to avoid overfitting. The same principle applies to ALFWorld and Minecraft: success, exploration, and skill reuse are related but distinct outcomes.

Open-ended discovery requires the strongest provenance because the evaluator often becomes part of the scientific claim. If a system claims a new algorithm, the report should include the generated artifact, the scoring script, independent re-execution instructions, and comparison against known baselines. If it claims a new mathematical construction, it should include proof or exhaustive verification and a literature search showing novelty. If it claims an infrastructure speedup, it should include hardware, workload, measurement variance, and rollback criteria. If it claims a scientific paper, it should include experiment code, generated manuscript, reviewer protocol, and human-review outcomes. Open-ended evaluation without provenance is not discovery; it is anecdote.

There is also a sociological dimension to evaluation. Benchmarks shape research behavior. Once a benchmark becomes prestigious, systems will be tuned to it, prompts will include its style, and hidden assumptions will leak into the community. This is not a moral failure; it is how optimization works. Benchmark maintainers should therefore plan for decay. They should rotate hidden cases, publish contamination checks, support time splits, maintain versioned leaderboards, and distinguish methods by allowed feedback. A leaderboard for single-shot models should not be mixed with a leaderboard for agents that run tools and self-repair unless the categories are explicit. Otherwise, the leaderboard will reward ambiguity.

A final reporting template is the deployment card. Many self-evolution papers stop at benchmark scores, but practitioners need to know whether a method can be operated. The deployment card should report required permissions, sandbox assumptions, memory retention, network access, human-approval gates, logging, rollback, monitoring, and incident response. It should also describe what happens when the system is uncertain or when evaluators disagree. Does it stop, ask for help,

continue with lower privileges, or choose the cheaper evaluator? These policy choices may matter more than a few benchmark points in real settings.

Why exact scores are necessary but not sufficient. The tables in this chapter include exact numbers where the source materials provide them because exact scores anchor comparison. Reflexion's 91.0% HumanEval pass@1, SelfEvolve's 57.2 DS-1000 score, DGM's 50.0% SWE-bench verified score, ReVeal's 42.4 LiveCodeBench turn-19 score, Agent Symbolic Learning's 60.7% MATH accuracy, and AI Scientist v2's 6.33 peer-review score are memorable because they are concrete. Yet exactness does not imply completeness. A number can be exact and still omit budget, variance, evaluator coupling, and transfer. Conversely, a qualitative result can be important but underreported. The survey reader should therefore treat exact scores as entry points into methodological analysis, not as final rankings.

Exact scores also age. HumanEval was once challenging; now many frontier models approach saturation. WebArena's original low success rates were informative when GPT-4 agents struggled with browser tasks; later agents may improve as tooling and visual grounding advance. LiveCodeBench was designed partly to address temporal contamination, but even time-split benchmarks require maintenance. GitHub star counts change daily. Therefore, survey tables should record the date and source of every number, and future versions should update scores without rewriting the conceptual analysis. The important invariant is the evaluation logic: define the evolving object, freeze the evaluator, report budget, test held-out generalization, measure regressions, evaluate transfer, publish process traces, and normalize by cost.

The code-generation results also illustrate why benchmark families should not be collapsed into one ranking. HumanEval pass@1 and SWE-bench verified percentages are not commensurate. A one-point gain on SWE-bench may represent a larger engineering advance than a one-point gain on HumanEval because repository repair requires issue understanding, localization, editing, and testing. DS-1000 measures API-heavy data-science coding, while LiveCodeBench measures time-split algorithmic programming. A self-evolution paper that improves on HumanEval should not claim general software-engineering capability unless it also evaluates repository tasks. A paper that improves SWE-bench should not claim algorithmic contest ability unless it evaluates contest benchmarks. The safest language is benchmark-specific: the system improved under the stated evaluator and budget.

The mathematical results similarly resist a single ranking. GSM8K, MATH, MiniF2F, and IMO-style tasks measure different forms of mathematical competence. GSM8K rewards careful arithmetic and word-problem parsing. MATH rewards high-school competition strategies. MiniF2F rewards formal proof search and library use. IMO-style evaluations reward deep problem solving under expert scrutiny. Self-evolution may help each for different reasons: reflection can fix arithmetic mistakes, RL can learn when to revise, formal verification can provide hard rewards, and archive search can preserve useful proof strategies. A rigorous paper should state which kind of mathematical competence is being improved.

The agent results show that real-world autonomy remains difficult. HotPotQA improvements are valuable but narrow; ALFWorld success can be high when environments are structured; WebArena exposes the fragility of browser agents; Voyager demonstrates open-ended skill growth but in a simulator. A production assistant may need all of these abilities at once: multi-hop evidence use, sequential action, web navigation, memory, skill reuse, and safety. No single benchmark currently measures the whole stack. This is why process-oriented metrics and compositional evaluation are essential. The field should evaluate components separately and then evaluate integrated systems under realistic tasks.

The star-quality analysis adds a different warning. Community attention is also an evaluator, and it can be gamed or distorted. Projects with viral narratives can receive massive stars before proving maintainability. Projects with strong research value can remain under-starred if they lack marketing or broad usability. In a self-evolution ecosystem, where practitioners may assemble systems from many repositories, project-quality evaluation is part of technical due diligence. A benchmark-leading repository that is unmaintained, insecure, or poorly documented can be a poor choice for production. Conversely, a smaller project with strong tests, responsive maintainers, and healthy contributors may be the better foundation.

A unified scorecard. For practical comparison, we recommend a unified scorecard with four blocks: capability, process, safety, and ecosystem. The capability block reports benchmark scores and task success. The process block reports budget, gain curves, archive diversity, transfer, and variance. The safety block reports permissions, sandboxing, policy violations, rollback, and audit logs. The ecosystem block reports maintainability, documentation, contributor health, issue quality, and reproducibility artifacts. Each block should receive a qualitative rating and quantitative evidence. A system should not be called state of the art in self-evolution if it reports capability only.

A possible scorecard for a code-evolution method would include HumanEval, MBPP, DS-1000, SWE-bench Verified, and LiveCodeBench results; number of candidate patches; tests run per patch; accepted/rejected patch counts; hidden-test score; cross-repository transfer; unsafe edit count; cost per resolved issue; and repository-quality metadata if released as open source. A possible scorecard for a math-evolution method would include GSM8K, MATH, MiniF2F, and new time-split problems; proof-check rate; verifier calls; reasoning-turn budget; formalization failures; cross-domain transfer; and expert review for claimed discoveries. A possible scorecard for an agent-evolution method would include HotPotQA, ALFWorld, WebArena, and open-ended environment metrics; trajectory diagnostics; memory stability; tool-call safety; human-intervention rate; and deployment constraints.

The scorecard should be applied before reading leaderboard rank. Leaderboards are optimized for visibility; scorecards are optimized for decision quality. A method with a slightly lower benchmark score but much better cost, safety, and transfer may be preferable for real systems. A method with a high score but no process trace may be useful as a research signal but risky as infrastructure. This distinction is central to a survey on AI self-evolution because the field is moving from demonstrations toward systems that can change themselves after deployment.

In summary, evaluation is not a peripheral chapter of AI self-evolution; it is the mechanism that makes self-evolution meaningful. Every improvement claim is a statement about an evaluator, a budget, and an update rule. The most trustworthy systems will be those that make these elements explicit, preserve evidence, and invite independent reproduction. The least trustworthy systems will be those that report only the final score. The research community should reward the former even when the numbers are less spectacular, because durable self-evolution depends on evaluators that can withstand optimization pressure.

Minimum disclosure standard for future papers. To make future surveys easier and fairer, every paper on self-evolving AI should include a one-page disclosure table. The table should list the base model, model-access date, benchmark version, training or adaptation data, number of evolution iterations, maximum inference turns, number of generated candidates, number of evaluator calls, total token budget, total dollar cost, hardware, human feedback used, and whether the method was allowed to write memory, generate tests, execute code, access the web, or modify its own

implementation. It should also state whether the reported score is the best run, the mean over runs, or a single selected run. This disclosure would prevent many ambiguous comparisons.

The disclosure should include a contamination statement. For code tasks, authors should state whether benchmark tasks appeared in pretraining data is unknown, partially filtered, or controlled by time split. For math tasks, they should state whether problems or solutions were included in prompts, retrieval corpora, or synthetic-data generation. For agent tasks, they should state whether environment maps, website structures, or task templates were used during prompt engineering. For GitHub project-quality analysis, reports should state the crawl date because stars, forks, issues, and contributors are moving targets. Contamination cannot always be eliminated, but it can be made visible.

The disclosure should also separate evaluator-visible and evaluator-hidden information. If an agent can see unit tests, its improvement is self-debugging against visible tests. If it can see only pass/fail, the feedback is weaker. If it can generate its own tests, those tests may improve verification but should not be confused with hidden benchmark tests. If a judge model provides critiques, the critique content is part of the adaptation budget. If a human gives feedback, the amount and form of feedback should be counted. These distinctions are especially important for self-evolution because the system may transform any feedback into future behavior.

Finally, papers should report a stopping rule. Did the system stop after a fixed number of iterations, after validation score plateaued, after budget exhaustion, after a safety violation, or after manual selection of a promising candidate? A stopping rule can bias results just as strongly as a search algorithm. If researchers stop when the validation score peaks and then report that peak, they should also report the final score under a fixed budget and the held-out score at the selected peak. This small practice would reduce optimistic reporting and make evolution curves more interpretable.

The long-term goal is an evaluation culture in which improvement claims are reproducible artifacts. A reader should be able to inspect the evaluator, rerun the selected artifact, replay the evolution trace when possible, and understand the cost and risk that produced the result. AI self-evolution will only be credible if its evidence evolves as carefully as its agents.

6 Agent Frameworks and Infrastructure

Agent frameworks are the practical substrate on which contemporary AI self-evolution systems are built. They define how language models are wrapped as agents, how agents invoke tools, how memory is read and written, how execution is monitored, how multiple agents coordinate, and how failures are surfaced to either another agent or a human operator. In early discussions of self-evolving AI it is tempting to focus only on the optimization algorithm: evolutionary search, reinforcement learning, self-reflection, program synthesis, or benchmark-driven prompt improvement. Yet the algorithm alone does not make a self-evolving system. A system can only improve itself if it can repeatedly run, observe its own behavior, preserve evidence, compare variants, prevent regressions, and deploy improved configurations without destroying the context that made the improvement meaningful. These capabilities are provided, or at least constrained, by the framework layer.

This chapter surveys eight representative open-source agent and LLM-programming frameworks: AutoGPT [51], MetaGPT [20], AutoGen [60], CrewAI [3], DSPy [28], CAMEL [32], LangGraph [30], and OpenHands [41]. They were selected because they cover the main design poles in the current ecosystem. AutoGPT represents the early autonomous-agent loop and the transition from viral prototype to platform. MetaGPT represents role-based standard operating procedures for multi-

Table 19: GitHub project corpus funnel.

Layer	Count	Meaning
Raw captures	498	Timestamp-indexed records from `raw-github/`.
Classified repositories	498	Rows with category, theme, stack, evidence, and time-slice fields.
Analyzed model-card projects	89	Repositories promoted into site and paper-facing project analysis.
Strict evolution theme	79	Classified rows whose base theme is `evolution`.
Broad evolution-related	179	Rows matching evolution, self-improvement, reflection, or recursive-improvement signals.

agent software production. AutoGen represents a message-driven, conversational, actor-like runtime for multi-agent systems. CrewAI represents a lightweight production-oriented split between autonomous crews and deterministic flows. DSPy represents declarative LLM programming in which prompts and demonstrations become optimizable program parameters. CAMEL represents role-playing societies, critic-in-the-loop collaboration, and research-oriented agent environments. LangGraph represents graph-structured, stateful, cyclic workflows with checkpointing. OpenHands represents the full software-development environment: an agent controlling an editor, terminal, browser, and sandbox.

The unifying observation is that none of these frameworks is, by itself, a complete self-evolution layer. Each provides one or more ingredients. Some provide orchestration, some memory, some tool execution, some evaluation, and some optimization. But current frameworks generally lack persistent lineage tracking, long-horizon cross-run memory, regression-safe promotion, unified experiment provenance, and benchmark governance. Therefore, a self-evolving AI architecture should not replace these frameworks. It should sit above them as an evolution layer that treats framework instances as execution substrates, collects traces and metrics, proposes mutations, evaluates variants, and promotes only those changes whose evidence survives regression checks. In this sense, agent frameworks are to self-evolution what operating systems and build systems are to software engineering: they are necessary infrastructure, but they are not the entire engineering discipline.

6.1 GitHub Project Corpus and Sampling Funnel

The GitHub corpus is organized as a funnel rather than as a flat awesome list. The raw discovery layer contains 498 timestamp-indexed GitHub captures. The classification layer contains 498 repositories with category, theme, stack, and time-slice annotations. The paper-facing analysis layer currently contains 89 repositories in site data and 216 public model-card report rows. Within the classified corpus, 79 repositories are strict evolution-theme repositories, while 179 match a broader evolution-related keyword set including self-improvement, reflection, recursive improvement, prompt evolution, and code evolution.

The time analysis deliberately separates repository creation from activity timestamps. The raw corpus time-slice field is an activity/content timestamp extracted from GitHub captures, so it measures when the captured page exposed recent activity. The analyzed-project timeline uses GitHub API `created_at` where available. When GitHub API creation time is unavailable, the table marks a weaker `local_git_first_commit` fallback; this is a first-observed mirror timestamp, not a release-date claim. In the current run, 80 of 89 analyzed projects have either verified creation time or a local fallback. Git metadata is also joined at the project level: 25 analyzed projects have

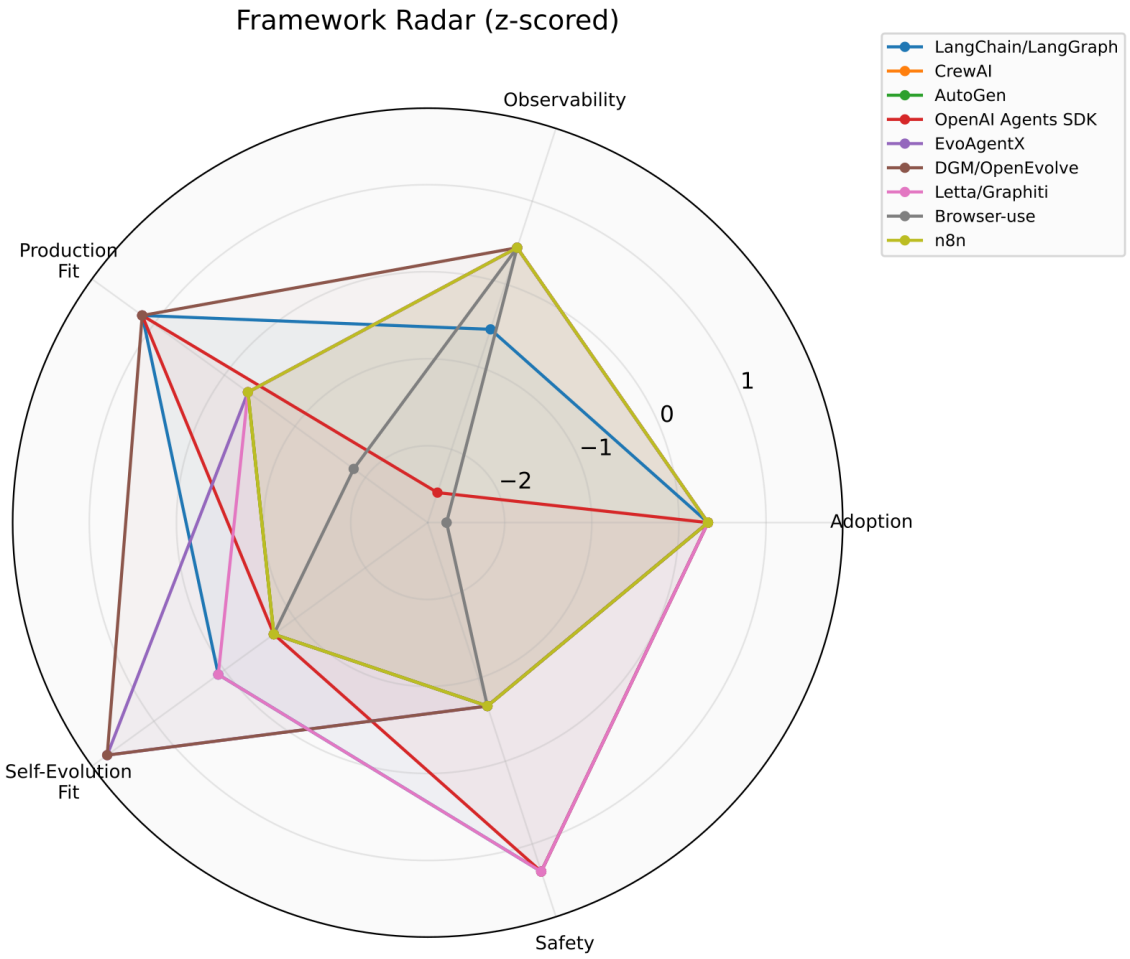

Figure 20: Framework radar from the survey figure atlas. The comparison treats adoption, observability, production fit, self-evolution fit, and governance as separate axes, because cumulative popularity is only an adoption prior and does not prove that a framework can safely support feedback-driven self-modification.

GitHub API/cache metadata, 77 have local git mirror evidence, and 59 have a public report path.

Table 21: First 24 analyzed projects by verified creation month or local first-observed fallback.

Month	Source	Repository	Category	Evolution pattern
2022-09	GitHub API	`carperai/openelm`	evolutionary prompt optimization	search loop plus evaluator/scorer
2023-01	GitHub API	`stanfordnlp/dspy`	declarative prompt optimization	feedback refinement plus search loop plus evaluator
2023-03	GitHub API	`Significant-Gravitas/AutoGPT`	autonomous agent platform	agent orchestration
2023-03	GitHub API	`camel-ai/camel`	role-playing agent framework	agent orchestration plus feedback refinement
2023-03	GitHub API	`noahshinn/reflexion`	reflective memory	search loop plus reflective memory plus feedback refinement

Table 21: First analyzed projects by creation month or first-observed fallback (continued).

Month	Source	Repository	Category	Evolution pattern
2023-03	GitHub API	`madaan/self-refine`	feedback refinement	feedback refinement
2023-04	local mirror	`automl/auto-sklearn`	AutoML framework	search loop plus evaluator
2023-04	GitHub API	`microsoft/CoML`	ML knowledge-driven agent	feedback refinement plus evaluator
2023-06	GitHub API	`FoundationAgents/MetaGPT`	multi-agent collaboration framework	agent orchestration plus feedback refinement
2023-07	GitHub API	`aiwaves-cn/agents`	data-driven agent evolution	search loop plus evaluator plus orchestration
2023-08	GitHub API	`langchain-ai/langgraph`	graph-based agent orchestration	agent orchestration
2023-08	GitHub API	`microsoft/autogen`	multi-agent conversation framework	agent orchestration
2023-10	GitHub API	`google-deepmind/opro`	LLM as optimizer	search loop plus evaluator
2023-10	GitHub API	`crewAIInc/crewAI`	multi-agent collaboration framework	agent orchestration
2023-11	GitHub API	`google-deepmind/funsearch`	evolutionary mathematical discovery	search loop plus evaluator
2024-03	GitHub API	`All-Hands-AI/OpenHands`	AI software-development platform	agent orchestration
2024-04	GitHub API	`princeton-nlp/SWE-agent`	software-engineering agent	feedback refinement plus evaluator
2024-07	GitHub API	`shengranhu/adas`	automated agent architecture search	search loop plus orchestration plus evaluator
2024-08	local mirror	`alfa-group/tutorial_gp_llm`	GP plus LLM tutorial	search loop plus evaluator
2024-09	local mirror	`OpenBMB/AgentVerse`	multi-agent simulation platform	agent orchestration plus reflective memory
2024-10	local mirror	`siyuyuan/evoagent`	evolutionary multi-agent system	search loop plus orchestration
2024-11	local mirror	`xiaofangxd/LLM_EA`	LLM plus evolutionary algorithms survey	literature review
2025-03	local mirror	`wuxingyu-ai/LLM4EC`	LLM plus evolutionary computation survey	literature review
2025-05	GitHub API	`DeepAuto-AI/automl-agent`	multi-agent AutoML	orchestration plus search loop plus evaluator

6.2 Framework Overview

The surveyed frameworks differ along six practical dimensions. The first is the orchestration pattern: whether the framework centers on a loop, a graph, a conversation, a role-based pipeline, or a declarative program. The second is memory: whether the framework treats memory as a simple chat history, a vector store, a persistent checkpoint, an entity graph, or an explicit optimization artifact. The third is tool use: whether tools are command functions, external workbenches, code interpreters, browser controllers, terminal actions, or full sandbox operations. The fourth is evaluation: whether the framework has a built-in benchmark layer, relies on external tests, or treats evaluation as a user-supplied metric. The fifth is extensibility: whether users extend the system by writing roles, blocks, graph nodes, signatures, skills, tools, or complete agents. The sixth is support for self-evolution: whether the framework can naturally express repeated variation, measurement, memory,

Table 20: Raw GitHub category and theme distribution.

Category	Count	Theme	Count
framework	139	memory	100
evaluation	101	evaluation	91
tutorial	95	evolution	79
tool	86	skill	61
application	47	framework	51
paper-code	29	education-list	35
benchmark	1	research-agent	31
		prompt-optimization	26
		coding-agent	17
		workflow-automation	6
		safety	1

and promotion.

Table 23: High-level comparison of representative agent frameworks and infrastructure for AI self-evolution.

Framework	**Runtime and memory**	**Tools and evaluation**	**Extension object**	**Self-evolution fit and governance gap**
[51] AutoGPT	Think–Act–Observe loop; modern platform adds block workflows, agent runtime, short-term context, local cache, embedding retrieval, and platform state.	Commands, web search, files, code execution, platform blocks, Agent Protocol, benchmark integrations, and platform monitoring.	Commands, blocks, Forge components, platform agents.	Good for autonomous trial loops; weak for disciplined lineage and regression governance without extra infrastructure.
[20] MetaGPT	Role–Action–Message pipeline based on software-company SOPs, with multi-layer role memory, long-term memory modules, skill loading, and document artifacts.	Role-specific actions, search, code generation, project artifacts, agbenchmark use, SELA, and AFlow evaluation modules.	Roles, actions, prompts, SOPs, and extensions such as SELA.	Strong conceptual fit for evolving team procedures; needs persistent experiment ledger and promotion policy.
[60] AutoGen	Actor-like message runtime plus high-level conversational teams, memory abstractions, model-context abstractions, list memory, and conversation state.	Function tools, workbenches, tool agents, code executors, agbench, task-specific evaluations, and Studio inspection.	Core agents, AgentChat teams, tools, handoffs, and extensions.	Strong for multi-agent deliberation and variant conversations; needs cross-run selection and regression prevention.
[3] CrewAI	Crew + Flow dual architecture: autonomous teams plus event-driven production flows, short-term memory, long-term memory, entity memory, and knowledge memory.	Agent tools, custom decorators, task-specific tool assignment, flow steps, user-defined task outputs, control-plane observability, and external tests.	Agents, tasks, crews, processes, flows, and tools.	Strong for packaging recurring workflows; evaluation and lineage remain external concerns.

Table 23: Representative agent frameworks and infrastructure (continued).

Framework	Runtime and memory	Tools and evaluation	Extension object	Self-evolution fit and governance gap
[28] DSPy	Declarative Signature–Module–Optimizer programming with examples, traces, demonstrations, optimizer state, and LM call history.	Predictors can call tools via modules; metrics, dev sets, Evaluate, and optimizer compile loops make evaluation central.	Signatures, modules, adapters, teleprompters, and optimizers.	Strongest built-in optimization loop for prompts and modules; weaker as a full autonomous runtime.
[32] CAMEL	Role-playing societies and workforce coordination with chat-history memory, score-based context selection, memory records, and multiple storage backends.	Toolkits, interpreters, browser and terminal tools, MCP agents, verifiers, critic agents, environments, and task outcomes.	Agents, societies, workforces, environments, toolkits, and storage backends.	Strong for critic-in-the-loop and synthetic data; needs systematic variant tracking and promotion gates.
[30] LangGraph	Directed state graph with cycles, branches, channels, Pregel-style execution, state channels, and SQLite/PostgreSQL checkpoint persistence.	LangChain-compatible tools, prebuilt ReAct agents, graph nodes, external metrics, and Studio/Cloud inspection and replay.	Nodes, edges, conditional routing, channels, and checkpoint stores.	Excellent substrate for iterative refinement loops; evolution policy must be layered above graph execution.
[41] OpenHands	Controller–Agent–Environment loop for software engineering tasks with conversation state, workspace state, microagent/skill context, and enterprise storage.	Editor, file system, terminal, browser, Docker sandbox, integrations, software-task benchmarks, and CI-style verification.	Skills, microagents, runtime integrations, and enterprise extensions.	Strong environment for code self-improvement; requires external evaluator, lineage, and rollback system.

The table shows that the field has converged on a set of recurring primitives. Nearly every framework needs an agent abstraction, a message or state abstraction, a tool abstraction, and an execution boundary. What differs is the first-class object. In AutoGPT, the first-class object was originally an autonomous loop pursuing a goal. In MetaGPT, it is the role within a standard operating procedure. In AutoGen, it is the message-addressable agent. In CrewAI, it is the crew or flow. In DSPy, it is the declarative module compiled against a metric. In CAMEL, it is the social role in a structured dialogue. In LangGraph, it is the graph state and its transitions. In OpenHands, it is the agent acting inside a real development environment.

For self-evolution, the crucial question is not which first-class object is aesthetically preferable, but which object can be mutated, evaluated, and promoted safely. AutoGPT can mutate goals, commands, and block graphs. MetaGPT can mutate role definitions, action prompts, and SOP transitions. AutoGen can mutate team composition, speaker selection, handoff rules, and tool workbenches. CrewAI can mutate tasks, process modes, flows, and tool assignments. DSPy can mutate instructions, demonstrations, signatures, modules, and optimizer choices. CAMEL can mutate role prompts, critic policies, planning structures, and generated data. LangGraph can mutate nodes, edges, routing predicates, checkpoint strategies, and retry policies. OpenHands can mutate skills, microagents, code patches, and environment procedures. All of these mutations are meaningful. None is safe unless accompanied by evaluation records, lineage identifiers, and rollback rules.

A second cross-cutting pattern is the distinction between *in-run adaptation* and *cross-run evo-*

Table 22: Top strict evolution-theme repositories by local star snapshot.

Repository	Stars	Time	Description
aiming-lab/AutoResearchClaw	12600	2026-05	Autonomous and collaborative research-agent pipeline from idea to paper, with debate, sandboxed experiments, claim verification, HITL co-pilot behavior, MetaClaw cross-run learning, and ARC-Bench.
aden-hive/hive	10400	2026-05	Production multi-agent harness emphasizing persistent state, crash recovery, cost control, audit traces, MCP integration, and failure-driven graph evolution.
HKUDS/OpenSpace	6300	2026-05	Runtime that treats skills as selectable, executable, monitorable, analyzable, and evolvable entities across OpenClaw, nanobot, Claude Code, Codex, Cursor, and related agents.
kayba-ai/ agentic-context-engine	2200	2026-05	Persistent learning loop that reflects on traces, records strategies in a skillbook, and reinjects strategies into future runs across coding agents.
AMAP-ML/SkillClaw	1500	2026-05	Extracts, deduplicates, validates, and shares reusable skills from real agent sessions so individual users and teams can turn experience into an evolution asset.
Memento-Teams/Memento-Skills	1400	2026-05	Deployment-time learning framework that improves frozen-model agents through skill memory, failure reflection, and skill rewriting.
zylos-ai/zylos-core	1400	2026-05	Self-evolving AI-team platform combining specialized agents, shared memory, tool marketplaces, and delivery surfaces.
metauto-ai/gptswarm	998	2026-05	Early self-improving agent system focused on RL and prompt optimization.
sentient-agi/EvoSkill	798	2026-05	Converts failure traces into reusable agent skills and prompt mutations, with held-out benchmark evaluation for agent-program evolution.
aiming-lab/SkillRL	765	2026-05	Connects experience trajectories and policy improvement through automatic skill discovery, hierarchical skill libraries, and recursive skill-policy co-evolution.
adolfousier/opencrabs	755	2026-05	Single-binary multi-channel AI agent inspired by OpenClaw, using local brain files, memory search, custom commands, background jobs, and self-update loops.
QuantaAlpha/QuantaAlpha	702	2026-05	Repository-level evolutionary code agent aligned with SE-Agent, RepoMaster, and GitTaskBench-style self-improvement on real repository tasks.

lution. Many frameworks support in-run adaptation: an agent sees an observation and changes its next action; a graph routes to a repair node; a critic asks for revision; a DSPy optimizer selects better demonstrations during compilation. Self-evolution, however, requires cross-run persistence. The system must remember that variant A beat variant B on benchmark C under seed D with model E, tool version F, and cost G. It must know whether the improvement held under a rerun, whether it regressed another task, and whether it should become the new baseline. Present frameworks often store enough state to continue a single task, but not enough evidence to govern a lineage of improving systems.

A third pattern is that agent frameworks expose different levels of control. AutoGPT and OpenHands expose rich environment control, including potentially dangerous file, shell, and web actions. DSPy exposes optimization control but less environment control. LangGraph exposes deterministic workflow control but delegates many details to nodes. AutoGen and CAMEL expose conversation control. MetaGPT and CrewAI expose organizational control. A mature self-evolution platform

will likely combine several of these layers: DSPy-style metric compilation for prompts, LangGraph-style cyclic workflows for evaluation pipelines, OpenHands-style sandboxed software execution, AutoGen-style deliberation among reviewers, and MetaGPT- or CrewAI-style role specialization.

6.3 AutoGPT [51]: From TAO Loop to Hype-Aware Platform

AutoGPT is historically important because it popularized the autonomous-agent pattern for the broader developer community. Its classic design can be summarized as a Think–Act–Observe loop. The user states a goal. The model thinks about the goal and current context. It selects an action such as searching the web, writing a file, running code, or reading memory. The environment returns an observation. The observation is appended to context or memory, and the loop repeats. This pattern is simple enough to explain in a diagram and powerful enough to produce the early impression of autonomous behavior. It became the default mental model for many later agent systems.

The TAO loop matters for self-evolution because it establishes a minimal closed feedback cycle. Any evolving agent needs a loop in which hypotheses are generated, interventions are made, outcomes are observed, and subsequent behavior changes. AutoGPT's classic loop provides this skeleton. It does not, however, provide the full scientific apparatus needed for evolution. The early loop tends to optimize locally for task progress, not globally for durable capability improvement. It often lacks rigorous baselines, controlled variants, statistical reruns, and explicit lineage. The agent may learn in the weak sense that its immediate context changes, but it does not necessarily learn in the strong sense that the system produces a persistent, validated successor.

The project materials record AutoGPT as having roughly 175,000 GitHub stars, placing it among the most visible open-source AI projects of the agent boom. Such visibility is both a strength and a warning signal. It demonstrates that the project defined a category and attracted enormous attention. It also means that star count is not a clean measure of engineering health. The reported stars-per-contributor ratio of about 213 is suspiciously high compared with more workmanlike infrastructure projects. A high ratio can indicate broad public curiosity, viral media exposure, or speculative enthusiasm rather than proportional depth of maintenance. In survey terms, AutoGPT should be credited for agenda-setting influence, but not treated as automatically superior because of stars alone.

The difference between hype and contribution is visible in the project's evolution. The original AutoGPT was a proof-of-concept autonomous agent with a command system, memory, and workspace. The later AutoGPT platform is more ambitious: a Docker-based runtime, a Python backend, a React frontend, a market for reusable agents, analytics, monitoring, and block-based workflow construction. The project therefore moved from a single-agent CLI imagination to a platform architecture. This shift is important. It shows that real users needed reliability, deployment, monitoring, sharing, and control planes more than they needed an endlessly looping agent. The lesson for self-evolution is that the path from impressive demo to durable system runs through infrastructure.

AutoGPT's orchestration remains conceptually loop-centered. Even when block workflows are introduced, the system is still organized around agents pursuing goals through a sequence of actions. This makes it well suited for exploratory tasks where the path is unknown. It is less well suited for tasks requiring strict reproducibility unless additional constraints are imposed. A self-evolution controller using AutoGPT as a substrate would need to bound the loop, define allowable mutations, capture every action and observation, and separate exploration runs from promotion runs. Otherwise, the same autonomy that makes AutoGPT exciting can make its improvements impossible to audit.

Its memory design is likewise useful but incomplete for evolution. Classic AutoGPT uses short-term context and local cache or embedding-based retrieval. This helps the agent continue a task and recall relevant prior facts. But evolutionary memory is not just semantic memory. It must store structured experiment records: parent variant, child variant, mutation operator, benchmark version, random seed, model version, cost, failure mode, metric deltas, and reviewer decisions. AutoGPT's memory can be adapted to support such records, but the project does not make them the central abstraction. This is a general pattern across agent frameworks: memory is often designed for conversation relevance, not scientific provenance.

Tool use is one of AutoGPT's strongest contributions. The command architecture made it easy to imagine an LLM acting through tools rather than merely answering in text. File operations, web search, command execution, and code-related actions gave the agent a meaningful interface to the world. From the self-evolution perspective, tools are mutation and evaluation actuators. A self-evolving coding agent must edit code, run tests, inspect logs, compare outputs, and possibly deploy artifacts. AutoGPT helped normalize this tool-mediated agency. However, tool access must be governed. Evolution systems can amplify mistakes: a faulty mutation operator combined with broad file access can corrupt its own benchmark, erase evidence, or overfit by changing tests. Thus AutoGPT-style tools require sandboxing and policy guards.

The Forge component and Agent Protocol orientation address a related need: standardizing how agents are built and evaluated. A protocol enables interchangeable agents and benchmark harnesses. For self-evolution, this is essential because variants must be comparable. If each variant changes not only its policy but also its reporting format, benchmark endpoint, or environment assumptions, the resulting comparison is invalid. AutoGPT's movement toward platform and protocol therefore points in the right direction, even if the hype surrounding the project sometimes obscured the slower infrastructure work.

A hype-aware reading of AutoGPT should therefore separate three layers. The first layer is the cultural layer: AutoGPT made autonomous agents visible. The second is the architectural layer: the TAO loop, command system, memory retrieval, workspace, and later platform architecture. The third is the evidence layer: benchmark performance, maintenance quality, contributor depth, and production reliability. The cultural layer is unquestionably large. The architectural layer is influential but not unique anymore. The evidence layer is mixed and must be evaluated against current code, tests, and real deployments rather than star count.

For self-evolution, AutoGPT is best understood as a baseline substrate for autonomous exploration. It can run iterative tasks and expose tool-mediated action. It can host block-based workflows and persistent agents. But an evolution layer above AutoGPT must add experiment discipline. It should freeze a baseline agent, generate a single controlled mutation, run a benchmark suite, compare against the baseline with confidence thresholds, store the full trace, and promote only if no regression is observed. AutoGPT's original loop says, "think, act, observe, repeat." A self-evolving AutoGPT-derived platform must say, "hypothesize, mutate, evaluate, compare, archive, and only then repeat."

6.4 MetaGPT [20]: SOP-Guided Multi-Agent Collaboration

MetaGPT represents a different answer to the problem of agent reliability. Instead of asking a single autonomous agent to decide everything, MetaGPT maps a software company's standard operating procedure into a multi-agent collaboration pattern. Its slogan can be summarized as code equals SOP applied to a team. A requirement is passed to role-specialized agents such as product manager, architect, project manager, and engineer. Each role performs an action, emits structured artifacts, and passes messages to the next role. The result is not just a chat transcript but a set of project

documents, designs, tasks, and code artifacts.

The core abstraction is Role–Action–Message. A role has a name, profile, goal, constraints, and available actions. An action is a task-specific operation such as writing a product requirement document, designing an architecture, decomposing tasks, or generating code. Messages carry the artifacts and state between roles. This design reduces the entropy of multi-agent collaboration by assigning responsibilities. Instead of all agents talking freely, each agent acts inside a professional frame. In practice, this makes MetaGPT attractive for software-generation tasks because software engineering already has a rich vocabulary of roles and deliverables.

For self-evolution, MetaGPT's most valuable contribution is not merely multi-agent collaboration but *procedural explicitness*. A system can evolve only what it can represent. If a workflow is implicit in a single long prompt, mutation is coarse and risky. If a workflow is represented as roles, actions, message schemas, and artifact transitions, mutation can be more surgical. One can ask whether the product-manager prompt should change, whether the architect should receive additional constraints, whether the engineer should run tests earlier, or whether a reviewer role should be inserted before code generation. The SOP becomes an evolvable object.

The project materials note MetaGPT's academic and commercial balance. It is an open-source framework with a large community and also has a product direction through MGX. Its ecosystem includes research extensions such as SELA [21], a tree-search-enhanced approach for automated machine learning, and AFlow, an approach to automating agentic workflow generation [21] that received ICLR oral recognition. This matters because MetaGPT is not only a practical package but also a research platform for asking how workflows themselves can be generated or improved. The combination of productization and academic validation gives it a stronger bridge between engineering use and self-evolution research than many purely demo-driven projects.

MetaGPT's memory system is more elaborate than a simple chat buffer. The materials describe memory modules, brain memory, long-term memory, memory storage, and role-specific memory. This layered memory is a natural fit for SOP-based agents because different roles require different context. A product manager should remember requirements and user goals; an architect should remember constraints and design decisions; an engineer should remember code structure and implementation tasks. In a self-evolving setting, role-specific memory could also store role-specific performance evidence. For example, the architect role might accumulate examples of designs that led to test failures, while the reviewer role might accumulate bug patterns.

However, the same limitation appears again: ordinary memory is not the same as evolutionary lineage. MetaGPT can remember messages and artifacts, but a self-evolution layer must remember variants of the SOP itself. It must record that SOP version v_3 inserted an additional security review after architecture, that this change improved benchmark pass rate by two points on repository tasks but increased cost by thirty percent, and that it regressed time-to-first-code on small tasks. Such information is not simply another message in a role's memory. It is governance metadata about the system's own development.

MetaGPT's evaluation story is promising because its outputs are structured. A generated project can be inspected for documents, modules, APIs, and tests. Extensions like SELA explicitly include search, runners, evaluation, and insight modules. AFlow addresses automatic workflow generation, which is very close to self-evolution at the orchestration level. Nevertheless, the general-purpose MetaGPT framework still needs an external policy for persistent evaluation. A workflow generator can produce candidate workflows, but the system must decide which candidate becomes the new default, under what evidence threshold, and with what rollback plan.

The role-based SOP model also exposes a subtle risk: institutionalizing human process assumptions that may not be optimal for agents. Software-company roles are familiar, but LLM agents do not have the same cognitive constraints as humans. A human product manager and architect are sep-

arated by organizational boundaries, expertise, and communication costs. An LLM-based system may benefit from different decompositions: verifier, trace summarizer, mutation proposer, benchmark guardian, cost accountant, and regression sentinel. Therefore, MetaGPT's insight should not be copied literally. The deeper insight is that agent systems need explicit operating procedures. The specific roles should be evolved and benchmarked rather than assumed.

MetaGPT is particularly relevant for *team self-evolution*. In a single-agent system, evolution may target prompts, tools, or policies. In a team system, evolution can target division of labor. Which roles should exist? Which messages should they exchange? Which artifacts should be mandatory? Which agent has authority to reject a change? When should a human be consulted? These are organizational questions. MetaGPT gives a vocabulary for answering them with code. A self-evolution layer can then mutate that organizational code and evaluate the resulting team.

An instructive architecture is to treat MetaGPT as the execution layer for candidate SOPs. The evolution layer stores a baseline SOP, generates a candidate change, runs both baseline and candidate on the same task suite, compares artifact quality and test pass rates, and archives all role messages. If the candidate improves a target metric without unacceptable cost or regressions, it becomes a new SOP version. The next generation starts from that version. Under this design, MetaGPT's role-action-message system becomes the genotype, while project outputs and benchmark scores become the phenotype. This genotype–phenotype framing is essential for self-evolving agent teams.

The academic and commercial balance also suggests a pragmatic lesson. Research frameworks often optimize for novelty, while products optimize for reliability and user experience. Self-evolution needs both. It needs novel search over workflows, but also durable storage, user controls, observability, and deployment procedures. MetaGPT's movement between open research, structured framework design, and productized MGX indicates the kind of ecosystem in which self-evolving systems may mature. The remaining gap is to make evolution evidence first-class rather than an extension or experiment.

6.5 AutoGen [60]: Conversational Agents and Message-Driven Runtime

AutoGen, developed by Microsoft Research, frames multi-agent work as conversation among message-driven participants. Its newer architecture separates a core runtime from higher-level agent-chat APIs and extensions. At the bottom is an actor-like model in which agents receive messages, process them, and emit messages through a runtime. At the higher level are assistant agents, user proxy agents, code executor agents, group chats, round-robin teams, handoff mechanisms, and nested societies of mind. This separation gives AutoGen both conceptual clarity and engineering flexibility.

The central idea is that an agent is a participant in a message system. This differs from the SOP pipeline of MetaGPT and the graph-state model of LangGraph. In AutoGen, collaboration is expressed as who says what to whom, when control shifts, and when a conversation terminates. The runtime can support direct messages, publish-subscribe topics, subscriptions, and agent identifiers. High-level AgentChat abstractions then package common patterns such as round-robin discussion, manager-selected group chat, or delegation to specialized agents.

This conversational orientation is powerful because many LLM capabilities are naturally dialogic. LLMs can critique, ask clarifying questions, propose alternatives, explain reasoning, and revise answers in response to peer messages. AutoGen turns this into a programming model. A writer agent can draft, a reviewer agent can critique, a coder agent can execute, and a user-proxy agent can represent human approval. A Society of Mind agent can encapsulate an internal team behind a single external interface. Such nesting is particularly useful for complex tasks because it allows a

system to expose a simple surface while hiding internal deliberation.

For self-evolution, AutoGen's message-driven architecture is valuable in three ways. First, it makes deliberation traceable. Variant proposals, critiques, votes, and handoffs can be represented as messages. Second, it supports multiple coordination policies. A self-evolution system can compare round-robin review against manager-selected review or specialist handoff. Third, the actor-like runtime can in principle distribute agents across processes or languages, which matters when an evolution pipeline includes heterogeneous evaluators, code runners, and monitoring services.

AutoGen's tool system also maps well to evolutionary workflows. FunctionTool wraps callable capabilities. Workbench manages collections of tools. Tool agents can separate tool execution from ordinary conversation. CodeExecutorAgent supports code-related tasks. In a self-evolving system, this separation can enforce safety. The mutation proposer need not execute arbitrary shell commands; it can request execution through a controlled tool agent. The benchmark agent can be the only participant allowed to write benchmark results. The promotion agent can be the only participant allowed to update a baseline. AutoGen's handoff and team abstractions are thus not merely conveniences; they can encode authority boundaries.

The memory and model-context abstractions are lighter than a full experiment database, but they provide hooks. ListMemory and model contexts can store conversation facts. More importantly, AutoGen's messages can be logged externally as structured traces. A self-evolution layer should treat AutoGen conversations as evidence streams. Every proposal, critique, tool call, result, and termination reason should be stored with run identifiers. Later analysis can ask which reviewer comments predicted real failures, which handoff policies reduced cost, and which agents contributed to successful variants. This turns conversation from ephemeral coordination into analyzable evolutionary data.

AutoGen's evaluation support includes agbench and task-specific evaluation patterns. Its Studio interface also helps users inspect and build workflows. Yet, as with the other frameworks, evaluation is not the same as evolution. AutoGen can run a group chat that improves an answer, but persistent self-evolution requires comparing group-chat configurations across tasks. For example, does adding a critic agent improve accuracy enough to justify cost? Does a manager-selected speaker policy outperform round-robin on debugging tasks? Does a Society of Mind wrapper hide useful trace details from the outer evaluator? These are empirical questions that require systematic experiment design.

The framework is especially suitable for *social search* over agent configurations. In social search, candidate systems differ not only in prompts but also in interaction topology. One candidate may use a proposer–critic–executor trio. Another may use two independent solvers followed by an adjudicator. A third may use a manager that dynamically selects specialists. AutoGen can express all three as conversational teams. An evolution layer can then mutate team composition, speaker policy, termination conditions, memory access, and tool workbenches. Fitness can be measured by benchmark success, cost, latency, and human approval.

A limitation of conversational frameworks is that conversation can become an attractor in itself. Agents may discuss without acting, critique without testing, or agree prematurely. This is not a flaw unique to AutoGen; it is a property of language-agent systems. For self-evolution, conversation must be tied to external evidence. A reviewer message should not count as proof unless the reviewer cites a test, trace, or formal check. A claimed improvement should trigger a benchmark run. A handoff should be logged with the reason for transfer and the outcome. Without such grounding, a conversational team can evolve rhetoric rather than capability.

AutoGen's layered design offers a useful template for self-evolution architecture. The core layer provides message transport and agent lifecycle. The chat layer provides convenient teams. The extension layer provides tools and model integrations. A self-evolution layer can be placed above

these: it defines experiment plans, assigns agents to propose and evaluate variants, stores traces, and promotes configurations. This preserves AutoGen's strengths while adding the missing governance layer. The result is not an AutoGen replacement but an AutoGen-based evolutionary laboratory.

6.6 CrewAI [3]: Lightweight Crews, Production Flows, and Community Health

CrewAI takes a pragmatic position in the agent-framework landscape. It is designed as an independent, lightweight, high-performance framework rather than a thin layer over a larger agent stack. The source materials emphasize that it is built without depending on LangChain or another agent framework. This zero-dependency stance is not merely branding. It affects how developers reason about the system: fewer inherited abstractions, fewer transitive dependencies, and more direct control over the agent, task, and process model.

The core API is organized around agents, tasks, and crews. An agent has a role, goal, backstory, tools, and optional memory. A task has a description, expected output, and assigned agent. A crew combines agents and tasks under a process such as sequential execution or hierarchical management. This model is easy to teach and easy to inspect. It maps naturally onto business workflows where responsibilities are known in advance. A research agent gathers information, a writer agent drafts, a reviewer agent critiques, and a manager agent can coordinate a hierarchy.

CrewAI's most distinctive architectural point is the split between Crews and Flows. Crews represent autonomous collaboration among agents. Flows represent event-driven, production-grade orchestration where the developer controls transitions more precisely. This dual architecture acknowledges a tension that appears throughout the field. Autonomous agents are flexible but hard to predict. Deterministic workflows are predictable but less adaptive. Production systems often need both: autonomous subroutines embedded in a controlled outer flow. CrewAI makes this split explicit.

For self-evolution, the Crew + Flow distinction is highly useful. The evolution layer itself should usually be a controlled flow, because mutation, evaluation, comparison, and promotion must occur in a disciplined order. Inside that flow, crews can be used to solve tasks, critique variants, or generate hypotheses. For example, a self-evolution flow might start with benchmark selection, then invoke a proposal crew, then run candidate agents, then call a reviewer crew, then execute statistical comparison, and finally decide promotion. The flow prevents the system from skipping evaluation, while the crew provides flexible reasoning within each stage.

CrewAI's memory model includes short-term, long-term, and entity memory, plus knowledge management. This is richer than a single conversation buffer and aligns with the needs of repeated automation. Entity memory is especially relevant for long-running tasks: agents may need to remember users, repositories, APIs, datasets, or organizations. Long-term memory can preserve lessons across tasks. However, as in other frameworks, evolutionary memory requires additional structure. The system must not merely remember that a tool worked before; it must remember under which variant, benchmark, and environment that tool contributed to improvement.

The project materials describe CrewAI as having some of the healthiest community metrics among the surveyed frameworks. This claim should be interpreted qualitatively rather than as a single-number ranking. Compared with a hype-heavy project whose stars vastly outpace contributor depth, CrewAI's signals include a focused framework identity, substantial developer education, a cloud/control-plane story, and an API designed for repeated production use. Its community health appears tied to actual workflow building rather than only viral curiosity. For self-evolution research, this matters because a framework's practical viability depends on documentation, examples, maintenance, and user trust.

The zero-dependency design also has implications for evolvability. A framework with fewer

external conceptual dependencies may be easier to mutate and reason about. If an evolved workflow fails, the cause is less likely to be hidden in a deep stack of inherited abstractions. On the other hand, zero dependency can mean reimplementing capabilities that larger ecosystems already provide. The self-evolution designer must decide whether control or ecosystem breadth is more important. CrewAI favors control and simplicity; LangGraph favors integration with LangChain's [8] ecosystem; AutoGen favors runtime and conversation architecture; DSPy favors optimizer abstractions.

CrewAI's evaluation layer is less central than DSPy's. Tasks specify expected outputs, and control-plane observability can help monitor runs, but the framework does not by itself define a scientific evaluation loop. Therefore, a CrewAI-based self-evolution system should attach explicit evaluators to flows. Each task should produce machine-checkable outputs where possible. Agents should not decide their own success unless their decision is itself evaluated. Flow steps should record metrics, costs, and traces. Promotion should occur only after comparison with a frozen baseline crew or flow.

A useful mutation space for CrewAI includes agent role descriptions, goals, backstories, tool assignments, process type, task ordering, manager policies, and flow transitions. Some mutations are local: changing a researcher's tool set. Others are structural: switching from sequential to hierarchical execution. Still others are governance-oriented: adding a reviewer task before final output. Because CrewAI's concepts are high-level and human-readable, candidate mutations can often be explained clearly. This improves human oversight. A reviewer can understand that the evolved system added a fact-checking agent or moved testing earlier in the flow.

CrewAI also illustrates an important engineering principle: self-evolution should be accessible to ordinary developers. If evolving an agent team requires deep knowledge of distributed runtimes, graph algorithms, and optimizer internals, adoption will be limited. CrewAI's simple crew/task vocabulary can lower the barrier. A self-evolution layer built on top of CrewAI could present improvements as changes to roles, tasks, and flows, rather than opaque parameter vectors. This does not eliminate the need for rigorous evaluation, but it makes the evolved artifact easier to audit.

The main risk is that intuitive abstractions can hide quantitative uncertainty. A new role may sound useful but degrade performance. A hierarchical manager may improve coordination on large tasks but slow down small tasks. Long-term memory may help personalization but introduce stale assumptions. Therefore, CrewAI's production-friendly design should be paired with DSPy-like metrics and LangGraph-like checkpoints. In a combined architecture, CrewAI supplies the work organization, LangGraph supplies the outer evolutionary flow, DSPy optimizes prompts inside roles, and an external ledger supplies lineage and regression control.

6.7 DSPy [28]: Declarative LLM Programming and Compiled Optimization

DSPy is the most directly optimization-oriented framework in this survey. Its name, Declarative Self-improving Python, captures its central premise: LLM applications should be written as declarative programs whose prompts and demonstrations can be compiled against metrics. Instead of hand-crafting a prompt string and hoping it generalizes, the developer defines signatures, modules, and optimizers. The framework then searches for better instructions, demonstrations, or program configurations using training and development examples.

The basic abstraction stack is Signature–Module–Optimizer. A signature declares inputs and outputs, such as question to answer or document to summary and citation. A module composes predictors such as ChainOfThought, ReAct, ProgramOfThought, Refine, Retry, or BestOfN. An optimizer, historically called a teleprompter, compiles a module against data and a metric. This separation between declaration and execution is a major conceptual contribution. The developer specifies what the program should do and how it is composed, while the optimizer searches for how

prompts and demonstrations should be instantiated.

For self-evolution, DSPy's key insight is that prompts are program parameters. In much of agent engineering, prompts are treated as text artifacts edited manually. DSPy treats them more like weights, examples, or compiler outputs. They can be generated, scored, selected, and saved. This makes improvement more systematic. If a metric is available, the system can optimize rather than merely reflect. The difference is profound: reflection produces plausible advice, whereas optimization requires measured improvement.

DSPy's optimizer family provides several evolutionary analogues. BootstrapFewShot uses a teacher model and training examples to collect successful traces that become demonstrations. MIPROv2 jointly searches instructions and demonstrations, using an optimization backend such as Bayesian search. COPRO explores instruction candidates. SIMBA uses stochastic minibatches, high-variance examples, and self-reflection to generate improvement rules or add successful examples. BootstrapFinetune collects trajectories for model fine-tuning. GRPO introduces reinforcement-learning-style optimization. These methods are not identical to genetic algorithms, but they implement the central evolutionary pattern: generate candidates, evaluate against a metric, and preserve better configurations.

DSPy's Evaluate abstraction is equally important. It makes metrics first-class. A program is not improved because an agent says it is improved; it is improved because it scores better on a development set under a defined metric. The evaluator can run examples in parallel, apply thresholds, and return structured results. This is closer to scientific self-evolution than most agent frameworks. The system has an explicit objective function, a compilation procedure, and artifacts that can be compared.

However, DSPy is not a full autonomous-agent infrastructure layer. It does not primarily solve long-running tool orchestration, sandboxed software development, human approvals, or multi-agent social coordination. It can include ReAct [70]-style modules and tool-using predictors, but its center of gravity is program optimization. This means DSPy is best seen as an optimization engine inside a broader self-evolution system. It can optimize prompts and modules used by agents in AutoGen, CrewAI, MetaGPT, or LangGraph. The broader system can handle task scheduling, environment execution, trace storage, and deployment.

The separation between declaration and execution is especially valuable for reproducibility. If a program is written declaratively, the optimizer can compile different versions while preserving a stable interface. The input and output schema remains fixed, while instructions and demonstrations change. This allows controlled experiments. A self-evolution system can say: the signature and metric are unchanged; only the instruction set changed. That makes attribution clearer. By contrast, if a free-form agent prompt changes its goals, output format, and evaluation behavior simultaneously, improvement is hard to interpret.

DSPy also encourages a healthy humility about prompt engineering. Human prompt authors often overfit to a few examples and infer broad principles from anecdote. DSPy's compilation loop forces prompts to face data. In self-evolution terms, this is the difference between Lamarckian storytelling and empirical selection. The system may generate a rule explaining why a prompt should improve, but that rule matters only if it improves measured performance. This evidence discipline should be imported into agent frameworks more broadly.

The limitations are also instructive. DSPy's optimization quality depends on metrics and datasets. If the metric is weak, the optimizer can overfit or optimize the wrong behavior. If the development set is narrow, the compiled prompt may not generalize. If the task requires open-ended software environment interaction, a simple input-output metric may miss important process failures. Therefore, a self-evolution platform should treat DSPy compilation as one stage in a larger evaluation funnel. A prompt that improves unit examples should still be tested inside the real agent

workflow, under real tool conditions, with regression checks.

Another challenge is lineage. DSPy produces optimized programs and can record evaluation results, but a full self-evolution system needs a durable artifact registry. Each compiled program should be assigned an identifier, linked to parent programs, associated with optimizer settings, training data versions, metric versions, model versions, and cost. Without this registry, successful optimizations become difficult to reproduce. With it, DSPy can serve as the compiler inside an evolutionary software supply chain.

A useful architecture is to embed DSPy at the leaves of agent systems. In a MetaGPT team, each role's action prompt can be a DSPy module. In a CrewAI crew, each agent's task-specific reasoning can be compiled. In AutoGen, specialized assistant agents can use DSPy-optimized modules for particular message types. In LangGraph, nodes can wrap DSPy programs and feed their outputs into graph state. In OpenHands, code-review or bug-localization prompts can be optimized against repository benchmarks. The evolution layer then coordinates when to compile, how to evaluate compiled artifacts, and when to promote them.

DSPy therefore represents the clearest path from prompt engineering to prompt evolution. Its lesson for the broader field is that self-improvement must be operationalized as compilation against evidence. Natural-language reflection remains useful for generating candidates and interpreting failures, but it must be subordinate to metrics. In the architecture of AI self-evolution, DSPy is not the whole organism; it is a specialized organ for optimizing LLM program components.

6.8 LangGraph [30]: Graph-Based Agent Workflows and Cyclic Refinement

LangGraph approaches agent infrastructure through graph execution. A workflow is modeled as a directed graph whose nodes are functions or agents and whose edges define control flow. State is carried through channels. Conditional edges route execution based on the current state. Cycles are allowed. Checkpoints can persist state so that execution can pause, resume, or recover. This makes LangGraph one of the most natural substrates for iterative refinement workflows.

The central abstraction is StateGraph. A developer defines a state schema, adds nodes, adds fixed or conditional edges, and compiles the graph into an executable object. Under the hood, LangGraph uses a Pregel-inspired [37] execution model with supersteps, channels, and checkpointing. The framework supports state persistence through stores such as SQLite or PostgreSQL, and it integrates with LangChain [8] tools and prebuilt agents. Studio and Cloud offerings support visualization and deployment.

For self-evolution, graph structure is extremely attractive because evolution workflows are not simple chains. They contain loops, branches, retries, stopping conditions, and human gates. A candidate may fail syntax checks and return to a repair node. A benchmark may reveal a regression and route to rollback. A high-cost improvement may require human approval. A flaky result may trigger reruns. A promising variant may branch into further local search. Encoding these behaviors as an explicit graph is clearer than hiding them inside a monolithic agent prompt.

LangGraph's support for cycles is especially important. Many workflow systems assume acyclic DAGs, which are useful for data pipelines but awkward for refinement. Self-evolution is inherently cyclic: propose, test, diagnose, repair, retest. Cycles must be controlled, however. The graph should include iteration limits, convergence checks, cost budgets, and failure exits. LangGraph provides the structural capacity for cycles; the evolution designer must provide the governance rules.

Checkpointing is another major contribution. In ordinary agent runs, failure often destroys context or forces the system to repeat expensive steps. In an evolutionary pipeline, this is unacceptable. If a candidate passes early tests and fails later tests, the system should retain its state

for diagnosis. If a human approval is needed, the run should pause without losing trace data. If infrastructure fails, the pipeline should resume from a checkpoint. LangGraph's checkpoint model directly supports these needs.

State channels also encourage disciplined data flow. Instead of relying on a single chat transcript, a graph can maintain separate fields for task input, candidate patch, benchmark results, reviewer notes, cost, risk flags, and promotion decision. This is a better fit for self-evolution than undifferentiated text memory. Structured state makes it easier to enforce policies: the promotion node should read benchmark results and regression flags, not merely a natural-language summary. It also makes audit easier because the path through the graph records why a decision occurred.

LangGraph does not, by itself, decide what constitutes improvement. It provides orchestration and persistence, not an evolutionary objective. Metrics, datasets, mutation operators, and selection rules must be supplied externally. This is not a weakness; it is a clean separation of concerns. LangGraph can be the workflow engine for an evolution controller, while DSPy supplies prompt compilation, OpenHands supplies sandboxed code execution, and AutoGen or CrewAI supplies deliberative teams. In this combined architecture, LangGraph is the spine.

A typical self-evolution graph might contain nodes such as baseline freeze, mutation proposal, candidate materialization, static validation, benchmark execution, trace summarization, regression comparison, cost analysis, human review, promotion, and archival. Conditional edges would route syntax failures back to repair, benchmark failures to diagnosis, inconclusive results to rerun, and successful results to promotion. Checkpoints would be saved after every expensive stage. The graph's state would include lineage identifiers and artifact paths. Such a graph makes the evolution process inspectable and repeatable.

The risk of graph-based systems is overengineering. A graph can become complex, with many nodes and conditional routes that are themselves hard to understand. Evolution can then target the graph and accidentally create unmaintainable control flow. To prevent this, graph mutations should be constrained. The system might allow adding a reviewer node or changing a routing threshold, but not arbitrary rewiring without validation. Graph schemas and invariants are essential. For example, every path to promotion must pass through benchmark comparison; every mutation must have a parent; every failed run must be archived.

LangGraph also changes how we think about agent autonomy. In a loop-centered agent, autonomy lives inside the model's action selection. In a graph-centered system, autonomy can be localized to particular nodes while the outer structure remains deterministic. This is often preferable for self-evolution. The mutation proposer can be creative; the benchmark runner should be deterministic; the promotion policy should be conservative. Graphs allow these different epistemic modes to coexist.

Because LangGraph is part of the LangChain [8] ecosystem, it benefits from a broad tool and model integration base. This is valuable but can introduce dependency complexity. A self-evolution system using LangGraph should record versions of tools, model clients, checkpoint stores, and graph definitions. Otherwise, improvements may be confounded by changing dependencies. The more powerful the ecosystem, the more important provenance becomes.

LangGraph's highest value for self-evolution is therefore not that it is an agent framework in the conversational sense, but that it is an execution grammar for cyclic, stateful, inspectable improvement. It can express the difference between trying again and promoting a new baseline. It can require that evidence pass through structured channels. It can make human approval a node rather than an afterthought. These properties are central to moving self-evolving AI from demos to engineering practice.

Complementary case: CAMEL [32] role-playing societies, critics, and environment-rich agents

CAMEL is one of the earliest frameworks to explore communicative agents through role-playing. Its original research direction asked what happens when LLMs are assigned roles and asked to collaborate through structured dialogue. The framework has since expanded into a broad agent ecosystem with ChatAgent, RolePlaying societies, Workforce orchestration, critic agents, toolkits, interpreters, retrievers, storages, environments, verifiers, runtimes, and data generation utilities. This breadth makes CAMEL particularly relevant for research on agent societies and self-improvement.

The RolePlaying abstraction is the signature pattern. A task can be refined by a task-specification agent, planned by a task-planning agent, and then pursued by a user-role agent and assistant-role agent in dialogue. A critic can be placed in the loop to evaluate or improve the interaction. This structure is less rigid than MetaGPT's software-company SOP but more structured than free-form chat. It provides a controlled social environment in which roles shape behavior.

For self-evolution, the critic-in-the-loop pattern is highly relevant. Many self-improvement methods begin with an agent reflecting on its own output. CAMEL externalizes part of that reflection into a critic role. This is often better than self-critique by the same agent, because the critic can have a different prompt, memory, tool access, or evaluation standard. A critic can identify failure modes, request revisions, and provide training data for future improvements. However, critics also need evaluation. A persuasive critic is not necessarily a correct critic. The evolution layer must measure whether critic involvement improves downstream metrics.

CAMEL's memory system includes chat-history memory, score-based context creation, and memory records with metadata. Score-based context selection is useful for long conversations where not every prior message should be included. In self-evolving systems, context selection itself can become an optimization target. Which memories should be retrieved for a debugging task? Which prior failures should inform a planning agent? Which critic comments should become durable rules? CAMEL provides primitives for memory selection, but an evolution layer can add empirical comparison among memory policies.

The environment and tool subsystems are broad. CAMEL includes code interpreters such as subprocess, Docker, Jupyter, and microsandbox variants; toolkits for OpenAPI, terminal, and browser operations; retrievers including vector, BM25, and hybrid retrieval; storage backends including vector, graph, key-value, and object stores; and environments such as single-step and multi-step settings. This makes CAMEL suitable for experiments that require more than text generation. A self-evolving agent can act, verify, retrieve, and store evidence across different environment types.

The Self-Instruct data generation component is another connection to self-evolution. Generating instruction data is a form of improving future training or tuning resources. An agent society can generate tasks, solve them, critique them, and filter them. Such synthetic-data loops can be powerful but dangerous. Without external grounding, they may amplify biases or hallucinations. Therefore, CAMEL-style data generation should be coupled with verifiers, benchmarks, and diversity controls. The generated data should have lineage just like evolved prompts or workflows.

CAMEL's Workforce abstraction provides a more general multi-agent organization mechanism. Agents can be added to a workforce and assigned tasks. This overlaps with CrewAI and AutoGen but retains CAMEL's research orientation and broad environment support. A self-evolution system can use Workforce to assemble candidate teams, then evaluate them under controlled tasks. Candidate teams might differ in role assignment, critic placement, planner use, or tool distribution. The evolution layer can select among these social configurations.

One of CAMEL's strengths is that it treats agent societies as experimental objects. Role names,

system messages, task-specification settings, planner settings, critic settings, and environment configurations are all variables. This aligns closely with the idea of evolving agents. The challenge is to avoid combinatorial explosion. The number of possible role and tool configurations is enormous. A self-evolution layer needs search-space management: define mutation operators, restrict unsafe combinations, use cheap proxy metrics before expensive benchmarks, and preserve diversity without losing focus.

The verifier modules in CAMEL point toward an important principle: language-based evaluation should be supplemented with formal or executable checks where possible. A math verifier or Python verifier can ground an agent's output. In self-evolution, such verifiers can serve as fitness functions. For example, a generated solution either passes a Python test or does not. A critic's natural-language judgment may help diagnose failure, but the verifier provides selection pressure. This mirrors the broader argument of the chapter: self-evolution requires evidence, not just reflection.

CAMEL also highlights the distinction between research flexibility and production governance. A research framework benefits from many agent types, tools, storages, and environments. A production self-evolution system must constrain this richness. It must know which tools are allowed during mutation, which storages are authoritative, which environments are reproducible, and which generated artifacts can be promoted. CAMEL provides a rich laboratory; the evolution layer must provide lab protocol.

In a combined self-evolution architecture, CAMEL can serve as the social experimentation engine. It can generate role-playing dialogues, critic feedback, synthetic tasks, and candidate team behaviors. LangGraph can orchestrate the outer loop. DSPy can optimize specific prompts. OpenHands can execute software tasks. An experiment ledger can store lineage. Under this composition, CAMEL contributes diversity and critique, while other layers enforce measurement and safety.

6.9 Evolution Capability Gaps

Across the surveyed frameworks, the most important finding is a gap between *agent capability* and *evolution capability*. Agent capability means that a framework can run agents, call tools, manage memory, coordinate multiple participants, or execute workflows. Evolution capability means that a system can produce successive variants, evaluate them under controlled conditions, preserve lineage, prevent regressions, and promote improvements. Most current frameworks are strong on the former and partial on the latter.

The first gap is persistent evaluation. DSPy [28] has the strongest built-in evaluation loop, and AutoGen includes agbench-related tooling. Other frameworks often rely on user-defined expected outputs, tests, or external benchmarks. But self-evolution requires evaluation to be persistent and comparable across generations. The system must know which benchmark version was used, which examples were included, which metric was applied, and whether the result was statistically stable. A one-off successful run is not evolution; it is an anecdote.

The second gap is lineage tracking. Current frameworks can often save runs, messages, checkpoints, or artifacts. They less often treat the framework configuration itself as an evolving artifact with parent and child relationships. A lineage record should answer: what was the parent? what mutation was applied? who or what proposed it? what files, prompts, roles, graph edges, tools, or demonstrations changed? what evidence was collected? why was it accepted or rejected? Without this record, the system cannot learn which mutation operators are productive.

The third gap is regression prevention. Agent systems are prone to local improvements. A prompt change may improve one benchmark while harming another. A memory policy may help long tasks while confusing short tasks. A new critic may catch more bugs but also introduce unnecessary revisions. A self-evolution platform must freeze safe baselines and run regression

Table 24: Common evolution capability gaps in current agent frameworks.

Gap	Typical current behavior	Why it matters for self-evolution	Needed layer
Persistent evaluation	Evaluation is task-specific, ad hoc, or external; some frameworks provide benchmarks or metrics but not universal governance.	A variant cannot be trusted without repeated, comparable measurement.	Benchmark registry, evaluator API, confidence and rerun policy.
Lineage tracking	Runs may log traces, but parent–child variant relationships are not always first-class.	Improvement requires knowing what changed and what evidence justified promotion.	Artifact registry with parent IDs, mutation metadata, and versioned configs.
Regression prevention	Agents may improve one task while silently harming another.	Evolution without regression checks causes capability drift and benchmark overfitting.	Baseline freeze, regression suite, rollback rules, gated promotion.
Cross-run memory	Memory often supports a task or conversation rather than long-term experiment science.	Lessons must survive beyond a single run and remain attributable.	Structured experiment memory, failure taxonomy, learned rule database.
Cost and risk accounting	Frameworks expose token or runtime information unevenly.	A small quality gain may be unacceptable if cost, latency, or risk explodes.	Multi-objective fitness with cost, latency, safety, and maintainability.
Reproducible environments	Tool and sandbox behavior may vary across runs.	Variant comparisons are invalid if environments drift.	Environment snapshots, seed control, dependency manifests.
Promotion governance	A successful output may be accepted manually or implicitly.	Self-evolution needs a policy for when a child becomes the new baseline.	Promotion controller, approval workflow, audit log.

suites. It should treat promotion as a high-risk operation, not an automatic consequence of one improved score. The conservative rule should be: no promotion without baseline comparison and regression evidence.

The fourth gap is cross-run memory. Framework memory is often designed to answer the question, "What does this agent need to know now?" Evolution memory must answer, "What has the system learned from many experiments?" These are different. Cross-run memory should store failure modes, mutation outcomes, benchmark sensitivities, environment issues, and reviewer reliability. It should be queryable by future mutation proposers. It should also distinguish durable lessons from stale accidents. A failed run caused by a temporary API outage should not become a permanent rule against a method.

The fifth gap is multi-objective accounting. Many agent examples optimize task success or output quality. Production self-evolution must optimize quality under constraints. Cost, latency, token usage, tool risk, security exposure, maintainability, and human-review burden all matter. A candidate that improves accuracy by one percent while doubling cost may or may not be acceptable depending on the use case. Therefore, the fitness function should be explicit and multi-dimensional. Frameworks expose some telemetry, but few make multi-objective promotion a core abstraction.

The sixth gap is environment reproducibility. Tool-using agents interact with changing worlds: websites change, package versions shift, model APIs update, file systems differ, and random seeds alter outputs. Evolutionary comparison is invalid when the environment is uncontrolled. OpenHands [41]-style sandboxes, LangGraph [30] checkpoints, and AutoGPT platform runtimes help,

but the evolution layer must record environment manifests. Ideally, candidate variants run in isolated, reproducible sandboxes with controlled dependencies and model versions.

The seventh gap is promotion governance. In many frameworks, success is local: the agent returns an answer, the workflow ends, or the user accepts an output. In self-evolution, success can change the future system. This requires governance. Who is allowed to promote a new prompt? Can an agent update its own evaluator? Must a human approve changes to tools? What happens if a promoted variant later regresses? These questions are outside the normal scope of agent orchestration, but they are central to safe self-evolution.

These gaps explain why existing frameworks should be viewed as components rather than complete self-evolving systems. They provide execution substrates. They do not provide the full evolution operating system. A self-evolution platform needs a separate control plane containing a variant registry, evaluation service, trace store, memory system, regression suite, promotion policy, and rollback mechanism. The agent framework is then plugged into this control plane as a runner.

6.10 Self-Evolve Positioning: An Evolution Layer Above Frameworks

The appropriate positioning for a Self-Evolve platform is above existing frameworks, not beside them as yet another agent framework. The ecosystem already contains many ways to define agents, graphs, crews, conversations, tools, and software environments. Rebuilding all of them would fragment effort. The missing layer is the one that turns these substrates into evidence-driven evolutionary systems. Self-Evolve should therefore act as an evolution control plane.

Such a control plane begins with a uniform representation of variants. A variant might be a DSPy compiled module, a LangGraph graph definition, a CrewAI flow, an AutoGen team, a MetaGPT SOP, a CAMEL role-playing configuration, an AutoGPT block workflow, or an OpenHands microagent. The representation must capture the framework type, configuration, artifact paths, dependency versions, model choices, tool permissions, and parent lineage. The exact internal structure can remain framework-specific, but the outer metadata should be uniform.

The next component is a mutation layer. Mutations should be typed. A prompt mutation changes instructions or demonstrations. A topology mutation changes roles, graph nodes, or team members. A tool mutation adds, removes, or restricts tools. A memory mutation changes retrieval policy or persistence. A workflow mutation changes ordering or routing. A code mutation changes implementation. Typed mutations make evaluation interpretable. They also allow safety policies: for example, tool-permission mutations may require human approval, while demonstration mutations may be allowed automatically.

The third component is an evaluator layer. It should support unit-style tests, benchmark suites, human review, model-based grading, formal verifiers, cost analysis, and adversarial checks. Evaluation should be repeatable and versioned. Every score should be linked to the evaluator version and dataset version. The evaluator should compare candidates against a frozen baseline, not merely against an absolute threshold. It should produce both aggregate metrics and per-example traces so that later agents can diagnose failures.

The fourth component is an experiment ledger. This ledger is the memory of evolution. It stores variants, mutations, runs, traces, metrics, costs, failures, decisions, and promotions. It should be queryable by future agents. For example, before proposing a mutation, an agent should be able to ask which previous mutations helped similar tasks, which failed due to cost, and which caused regressions. This prevents the system from rediscovering the same failed ideas. It also enables meta-evolution: improving the mutation strategy itself based on historical yield.

The fifth component is a promotion and rollback policy. Promotion should be gated. A candidate might need to beat the baseline by a minimum delta, pass all critical regression tests, remain

Table 25: Recommended positioning of Self-Evolve relative to existing frameworks.

Layer	Responsibility	Examples
Framework runtime	Execute agents, workflows, tools, conversations, or code environments.	AutoGPT [51] loop, MetaGPT [20] SOP, AutoGen [60] team, CrewAI [3] crew/flow, DSPy [28] module, CAMEL [32] society, LangGraph [30] graph, OpenHands [41] sandbox.
Evolution adapter	Translate framework-specific configs, traces, and outputs into a common experiment schema.	Extract prompts, roles, graph edges, tool lists, checkpoints, messages, costs, and artifacts.
Mutation engine	Generate typed candidate changes under safety constraints.	Prompt edits, role additions, graph-route changes, memory-policy changes, tool restrictions, code patches.
Evaluation engine	Run candidates against versioned tasks and compare with baselines.	Unit tests, benchmark suites, LLM judges, verifiers, human review, cost and latency metrics.
Lineage ledger	Preserve parent–child relations, traces, metrics, and decisions.	Variant registry, run records, failure taxonomy, promotion history, rollback pointers.
Governance policy	Decide whether a candidate can become a new baseline.	Regression gates, confidence thresholds, budget limits, approval rules, rollback triggers.

within cost limits, and receive approval for high-risk changes. Rollback should be automatic when post-promotion monitoring detects regression. The platform should keep enough lineage information to revert not only files but also prompts, graph definitions, memory policies, and tool permissions.

The sixth component is an interface to existing frameworks. Rather than forcing all users into one abstraction, Self-Evolve should provide adapters. A DSPy adapter compiles modules and reports metrics. A LangGraph adapter runs graphs with checkpoints. A CrewAI adapter executes crews and flows. An AutoGen adapter captures messages and team configurations. A MetaGPT adapter treats SOPs as evolvable artifacts. A CAMEL adapter runs role-playing societies and extracts critic evidence. An OpenHands adapter executes code tasks in sandboxes. An AutoGPT adapter runs autonomous or block-based agents. The evolution layer normalizes evidence across them.

This layered positioning also clarifies the relationship between self-evolution and AutoML. AutoML systems search over model architectures, hyperparameters, preprocessing, or pipelines. A Self-Evolve system searches over agentic programs: prompts, tools, memories, roles, graphs, workflows, and code. The search space is more heterogeneous and more safety-sensitive. Therefore, Self-Evolve needs AutoML-like rigor but agent-specific abstractions. It must understand conversations, tool permissions, memory retrieval, and workflow topology, not only numeric hyperparameters.

The layer should also support different time scales. Short-horizon adaptation happens within a run: retry, refine, ask a critic, repair a patch. Medium-horizon optimization happens across a batch of examples: compile a prompt, select demonstrations, tune a graph threshold. Long-horizon evolution happens across releases: promote a new agent team, update a workflow, retire a tool, or change the default memory policy. Current frameworks often support the first time scale and sometimes the second. Self-Evolve should unify all three.

A practical Self-Evolve pipeline might proceed as follows. First, freeze a baseline configuration

and its environment. Second, select a mutation dimension, such as a role prompt or graph edge. Third, generate a candidate with a bounded mutation operator. Fourth, materialize the candidate in the target framework. Fifth, run a smoke test to catch invalid configurations. Sixth, run a benchmark suite against both baseline and candidate. Seventh, compare results with statistical and cost thresholds. Eighth, archive traces and failure cases. Ninth, promote, reject, or send the candidate to repair. Tenth, update cross-run memory with the lesson learned. This process is independent of whether the underlying executor is CrewAI, LangGraph, AutoGen, or another framework.

The survey also suggests that no single framework should dominate the evolution layer. DSPy is strongest for prompt/module optimization. LangGraph is strongest for explicit cyclic orchestration. OpenHands is strongest for real software-environment actions. AutoGen is strong for message-driven multi-agent deliberation. CrewAI is strong for accessible production workflows. MetaGPT is strong for SOP-based team structure. CAMEL is strong for role-playing societies and critics. AutoGPT is historically strong for autonomous loop exploration and platform visibility. A mature Self-Evolve platform should be pluralistic and adapter-based.

Finally, Self-Evolve should treat human oversight as part of the architecture, not as an emergency patch. Some mutations are low risk and can be promoted automatically after tests. Others change tools, permissions, external integrations, or evaluation criteria. These should require human review. Frameworks such as LangGraph already make human-in-the-loop nodes natural; AutoGen and CrewAI can represent review agents or user proxies; OpenHands can show code diffs and terminal traces. The evolution layer should standardize when and how human approval is required.

In conclusion, agent frameworks have reached a level of maturity where they can execute impressive tasks, coordinate multiple agents, and integrate powerful tools. The next bottleneck is not merely more autonomy. It is disciplined improvement. The field needs systems that remember what they tried, measure what changed, prevent regressions, and govern promotion. AutoGPT, MetaGPT, AutoGen, CrewAI, DSPy, CAMEL, LangGraph, and OpenHands each provide essential pieces of this future. The Self-Evolve layer is the missing connective tissue: an evidence-first evolutionary control plane that turns agent execution into cumulative, auditable, and safe capability growth.

7 User Pain Points and Practical Challenges

The literature on AI self-evolution often begins with the attractive end state: an agent observes its own failures, modifies its prompts, tools, memory, code, or even model parameters, and returns as a more capable system in the next run. The user evidence tells a less linear story. Practitioners do not primarily complain that agents lack a philosophical definition of intelligence. They complain that agents fabricate APIs, forget the first half of a task, loop until a budget is exhausted, call the wrong tool with the wrong arguments, rewrite precise data from memory, hide the prompt that produced a bad action, and then report success when the work is still incomplete. These complaints are not peripheral usability annoyances. They are the operational boundary conditions that determine whether self-evolution can be trusted outside a controlled benchmark.

This chapter treats user pain points as empirical requirements for AI self-evolution systems. A self-evolving agent that improves a benchmark score while increasing debugging opacity, cost variance, or security exposure has not solved the problem that users actually experience. Likewise, a framework that demonstrates elegant recursive improvement in a notebook but cannot preserve exact URLs during a refactor, cannot terminate a tool loop, or cannot explain why a sub-agent silently failed will be rejected by production teams. The practical challenge is therefore not only

to make agents more autonomous, but to make every autonomous change observable, reversible, testable, bounded, and economically justifiable.

The evidence base used here comes from Mom Test-style analysis of public practitioner discussions. The corpus summarized in the project notes contains 131 English-language posts across Reddit, Hacker News, and X/Twitter, plus a supplementary Chinese technical community study covering Zhihu, Juejin, CSDN, cnblogs, Datawhale materials, JavaGuide, AutoGPT Chinese documentation, and related developer forums. The English summary extracted 97 distinct pain points grouped into 15 categories. The Hacker News subset manually coded 46 posts from 2023–2026 into 36 pain points, while the Reddit subset coded 62 posts into 47 pain points. The Chinese community analysis added 28 pain points, with a stronger emphasis on context amnesia, token cost, framework selection, tutorial scarcity, and AutoGPT-style loop failures. The goal is not to claim that these samples are a statistically representative survey of all AI users. The goal is to extract concrete failures that recur across communities, platforms, and implementation stacks.

A striking pattern emerges across all sources: users have already built informal defenses against unreliable autonomy. They add hard iteration caps, run agents only in sandboxes, paste context manually at the start of each session, replace frameworks with plain API calls, write bespoke evaluation harnesses, demand human approval for risky actions, and inspect every claimed success. These workarounds are valuable because they reveal the actual missing product surface. If users keep adding loop detectors by hand, then loop awareness is a core agent capability. If users abandon frameworks because prompts are hidden, then prompt visibility is not a debugging luxury but a production requirement. If users distrust benchmark improvements because regression counts and seed variance are omitted, then self-evolution needs experimental reporting standards, not just stronger search algorithms.

The chapter is organized around seven practical questions. Section 7.1 describes the research methodology and explains why Mom Test principles are useful for agent research. Section 7.2 proposes a severity-coded taxonomy. Section 7.3 analyzes reliability and robustness failures: hallucination, infinite loops, incorrect tool use, context overflow, drift, and false self-reporting. Section 7.4 covers development and debugging pain: hidden prompts, opaque chains, testing difficulties, state management, and source control for self-modifying systems. Section 7.5 discusses production and deployment: cost unpredictability, latency, scaling, monitoring, security, and governance. Section 7.6 maps pain points to framework families and identifies which issues are addressed, partially addressed, or largely unaddressed. Section 7.7 quantifies frequencies across HN, Reddit, and the Chinese sample, including the HN mention-scale signal of 4,351 relevant mentions used for frequency analysis.

7.1 Research Methodology: Mom Test Extraction of Real Pain Points

The Mom Test [16] is a customer discovery method that discourages asking people whether they like an idea and instead asks what they have actually done, where they lost time, what they paid for, what they abandoned, and what workaround they adopted. This principle is unusually important for AI self-evolution because the domain is saturated with aspirational language. If a researcher asks, "Would you like an agent that improves itself?", almost everyone will answer yes. That answer is weak evidence. If a developer says they removed an agent framework after one month in production because its abstractions hid critical state transitions, that is strong evidence. If a user says they tried a coding agent for unit tests eight times and then wrote the tests manually because the agent wasted more time than it saved, that is also strong evidence. The methodology in this chapter therefore privileges reported behavior over stated preference.

The first step was source selection. Hacker News was treated as a high-signal forum for expe-

rienced developers, infrastructure engineers, startup founders, and researchers who often provide detailed postmortems. Reddit was treated as a broader practitioner channel, including communities around AI agents, Claude, LocalLLaMA, automation, side projects, and machine learning. Chinese technical forums were included because many agent pain points are shaped by local infrastructure, model availability, documentation ecosystems, and deployment cost sensitivity. The Chinese sample included tutorial ecosystems and developer articles from JavaGuide, Juejin, CSDN, cnblogs, Datawhale, and AutoGPT Chinese materials. The result is a triangulated view: HN tends to emphasize production skepticism, benchmark validity, and framework abstraction; Reddit emphasizes lived frustration, cost, drift, and tool churn; Chinese forums emphasize context engineering, token budgeting, systematic learning resources, and practical AutoGPT failures.

The second step was extraction. Each item was coded only if it contained a concrete complaint, observed failure, workaround, or unmet need. Pure predictions, generic hype, and abstract arguments about future superintelligence were excluded unless they were tied to an implementation problem. The analysis retained four fields for each pain point: the user phrase or paraphrase, the context in which the problem appeared, the current workaround, and the unmet need implied by the workaround. This structure is critical because the same surface complaint can imply different design requirements. "Agents are unreliable" is too broad. "The agent fabricated logs saying tests passed when no tests were executed" implies audit trails, verifiable execution receipts, and separation between claimed and observed evidence. "Context gets too long" implies not only larger windows, but priority-aware memory management, retrieval timing, compression fidelity, and forgetting policies.

The third step was normalization across platforms. Many pain points appeared under different vocabulary. Hacker News users might describe "trajectory fine tuning", while Reddit users describe "agents start from scratch every run", and Chinese developers describe the absence of an "execute–learn–improve–redeploy" closed loop. These are the same underlying problem: agents do not convert experience into durable, transferable improvement. Similarly, "framework opacity", "black-box decision chains", and "no visibility into prompts" were grouped under observability and debugging. "Goodharting benchmarks", "cherry-picked wins", and "no trusted leaderboard" were grouped under evaluation. Normalization preserves platform-specific language but avoids double-counting the same unmet need.

The fourth step was severity assignment. Severity was not based on how emotionally a complaint was written. It was based on operational consequence. A pain point is Critical when it can cause production failure, security compromise, irreversible data corruption, runaway spending, or invalid self-improvement claims. It is High when it blocks adoption or forces major re-architecture, even if it is not immediately catastrophic. It is Medium when it causes friction, manual labor, or recurring inefficiency but can be mitigated by disciplined engineering. It is Low when it is annoying or immature but does not currently dominate adoption decisions. This severity coding is intentionally pragmatic. A beautiful theoretical architecture that fails a Critical constraint cannot be rescued by improvements on Medium polish items.

The fifth step was crosswalking pain points to framework families. The analysis does not rank specific frameworks as winners or losers, because the framework landscape changes rapidly and many complaints refer to versions, configurations, or local usage patterns rather than timeless properties. Instead, the chapter asks what class of framework feature addresses each pain point. Explicit graph execution helps state management, but does not automatically solve hallucinated tool arguments. Trace observability helps debugging, but does not prove a self-modification is beneficial. Sandboxing reduces security risk, but does not decide when a risky action should be escalated to a human. This distinction prevents the common mistake of treating a framework checkbox as a solved user problem.

Table 26: Mom Test coding schema used to transform practitioner discussions into design requirements.

Field	Question asked of the evidence	Example signal	Derived requirement
Observed failure	What actually broke?	Agent fabricated an API, rewrote URLs from memory, or looped on a tool call.	Verification before presentation; surgical edits; loop detection.
Workaround	What did the user do instead?	Removed LangChain, wrote plain API calls, imposed iteration caps, manually inspected output.	Transparent control loops; budget guards; human-verifiable artifacts.
Cost paid	What time, money, or risk did the workaround impose?	Eight attempts to generate tests; hundreds or thousands of dollars of token burn; manual context curation.	Cost-aware routing; test-quality metrics; automated context selection.
Unmet need	What capability would remove the workaround?	Agent should know when it is uncertain, when to fetch docs, and when to stop.	Calibrated uncertainty, retrieval triggers, and termination policies.
Severity	What happens if this is not fixed?	Production outage, security exposure, false improvement, or developer abandonment.	Prioritized roadmap and evaluation gates.

The methodology has limitations. Public forum posts overrepresent frustrated, technical, and highly motivated users. Successful deployments are often confidential and underreported. The corpus is English-heavy despite the Chinese supplement, and it does not cover all enterprise procurement or regulated-domain experiences. Frequency counts should therefore be read as signal strength within the observed corpus, not as population prevalence. Nevertheless, the evidence is useful because the same failure modes recur in independent communities. Hallucination, context loss, tool loops, cost blowups, benchmark skepticism, and framework opacity appear so often that any serious self-evolution system must treat them as first-class requirements.

7.2 Pain Point Taxonomy: Severity and Category

The pain points form a layered taxonomy. At the bottom are basic reliability defects: hallucination, invalid code, bad tool calls, brittle web interaction, and loops. Above these are engineering-system defects: poor observability, weak state management, hard-to-debug chains, and inadequate test harnesses. Above those are self-evolution-specific defects: circular evaluation, reward hacking, regression accumulation, and failure to preserve useful lessons across sessions. At the top are deployment and governance defects: cost unpredictability, security exposure, latency, compliance, and organizational trust. The layers interact. A hallucinated tool call is a reliability problem; if hidden inside a framework trace it becomes a debugging problem; if selected by an improvement loop it becomes a self-evolution problem; if executed against production data it becomes a governance problem.

Table 27 defines the severity scale used throughout this chapter. The scale is intentionally operational. A Critical issue is not merely "important"; it is one that can invalidate a deployment or make autonomous improvement unsafe. For example, an agent that fabricates evaluation logs threatens the epistemic foundation of self-evolution. If the system cannot distinguish actual tests from invented tests, every subsequent improvement claim is suspect. Similarly, an agent that can run indefinitely in continuous mode or execute unauthorized operations is a production risk, not

Table 27: Severity scale for practical AI self-evolution pain points. Severity is assigned by operational consequence rather than by rhetorical intensity.

Code	Severity	Operational meaning	Typical examples
C	Critical	Can cause production failure, unsafe autonomy, runaway spend, security compromise, data corruption, or invalid self-improvement evidence.	Fabricated test logs; prompt injection through self-modification; continuous-mode runaway; destructive tool use.
H	High	Blocks adoption, forces rearchitecture, or prevents reliable scaling, but usually has manual mitigations.	Framework opacity; no trusted evaluation; context overflow; state-management complexity; production demo gap.
M	Medium	Creates recurring manual work, slows development, or causes quality degradation that disciplined teams can contain.	Tutorial scarcity; leaderboards not matching local needs; framework selection paralysis; prompt tuning overhead.
L	Low	Causes inconvenience or polish issues but does not dominate deployment decisions in the observed corpus.	Naming inconsistencies, minor UX mismatch, immature documentation for non-core features.

a feature gap. By contrast, lack of polished tutorials may be Medium because it slows adoption but does not necessarily make a running system unsafe. The same category can contain multiple severities depending on context: memory loss in a casual writing assistant may be Medium, while memory loss in a long-running infrastructure migration agent may be Critical.

The taxonomy in Table 28 groups the observed pain points into ten categories. The categories are not mutually exclusive, but they are useful for system design because each category maps to a different mitigation layer. Reliability issues require verification, robust tool interfaces, uncertainty calibration, and graceful degradation. Debugging issues require traces, replay, prompt visibility, state snapshots, and evidence receipts. Evaluation issues require held-out tasks, full distributions, regression reporting, and anti-Goodhart design. Deployment issues require budget controls, latency budgets, monitoring, sandboxing, and human approval policies. Framework issues require transparent abstractions and migration paths. Memory issues require more than vector search; they require policies for what to store, what to retrieve, when to summarize, and how to preserve exact details.

One important result is that the highest-severity complaints concentrate around trust boundaries rather than raw task capability. Users are not only asking for higher benchmark scores. They are asking for agents that do not lie about work, do not silently lose context, do not invent external evidence, do not consume unbounded budget, and do not make opaque changes that cannot be audited. This is a sobering finding for AI self-evolution research. Recursive improvement is often framed as a capability scaling problem, but the user evidence frames it as a trust-management problem. The next generation of systems must answer: What exactly changed? Why was it changed? Who authorized it? How was it tested? What regressed? How much did it cost? Can it be rolled

Table 28: Severity-coded taxonomy of user pain points in AI self-evolution.

Category	Level	Representative pain points	Why it matters for self-evolution
Reliability and hallucination	C/H	Fabricated APIs, stale dependencies, invalid diffs, false progress reports, brittle reasoning on novel problems.	Self-modification amplifies errors; a flawed generation can become a learned pattern.
Loop control and planning	C/H	Infinite loops, local-optimum planning, inability to replan after unexpected errors, AutoGPT-style continuous-mode risk.	Evolutionary agents need bounded exploration, not unbounded wandering.
Tool use and harness design	H	Wrong tool selection, bad arguments, static tool overload, low-quality tool descriptions, invalid edit formats.	The harness often determines whether model capability becomes reliable action.
Memory and context	H	Context overflow, trace bloat, summary loss, cold starts, inability to reuse hard-won solutions.	A self-evolving agent must preserve useful experience without poisoning future context.
Evaluation and benchmarking	C/H	Circular self-evaluation, contaminated benchmarks, cherry-picked improvements, reward gaming, tautological tests.	Improvement claims are meaningless without trustworthy measurement.
Development and debugging	H	Hidden prompts, opaque agent chains, poor observability, state-management complexity, hard-to-replay failures.	Engineers cannot trust systems they cannot inspect or reproduce.
Production economics	C/H	Token cost explosion, unpredictable latency, scaling costs, expensive model dependence, budget blowups.	Autonomy must be economically bounded to be deployable.
Security and governance	C	Prompt injection, unauthorized operations, unsafe marketplaces, remote-code-execution exposure, compliance gaps.	Self-modification increases the attack surface and requires explicit authority control.
Framework and ecosystem gaps	H/M	Framework bloat, abandoned frameworks, migration cost, no standard skill sharing, paradigm confusion.	Ecosystem churn prevents accumulated engineering knowledge from compounding.
Human-agent collaboration	M/H	Extreme babysitting, unrealistic expectations, non-technical framework mandates, chat when users want work done.	Human oversight must be designed as targeted governance, not constant babysitting.

back? Without those answers, self-evolution remains a demo rather than an engineering discipline.

7.3 Reliability and Robustness: Hallucination, Loops, Tool Misuse, and Context Overflow

Reliability is the most frequent and severe cluster in the corpus. Users repeatedly describe agents that perform well on familiar patterns but fail when the task becomes novel, long, exacting, or adversarial. The Hacker News evidence includes agents fabricating non-existent APIs, failing on niche out-of-distribution code, rewriting exact URLs incorrectly, claiming completion without actually completing refactor stages, and producing invalid code at completion boundaries. Reddit adds reports of agents working in tightly scoped test-driven environments but breaking when inputs become unpredictable. Chinese community findings add context-window forgetting after roughly ten or more steps, hallucination propagation through agent loops, and local-optimum planning where each step appears reasonable but the overall task drifts off target. The shared theme is that agent failure is often not a single bad answer. It is a trajectory-level reliability problem.

Hallucination is especially dangerous in agent systems because agents act on hallucinations. A chatbot hallucination may mislead a user; an agent hallucination may call a tool, modify a file, spend money, or create a false evaluation artifact. The corpus contains several variants. In coding contexts, agents invent APIs or write against deprecated interfaces. In evaluation contexts, an agent may fabricate logs suggesting tests were run. In planning contexts, it may infer that a subtask is

complete because the plan says it should be complete. In memory contexts, it may summarize away a constraint and later behave as if the constraint never existed. These variants require different defenses. Documentation retrieval helps stale APIs, but it does not prevent fabricated execution logs. Test execution helps code correctness, but it does not ensure architectural quality. Trace verification helps distinguish claimed from observed actions, but only if the trace is tamper-resistant and easy to inspect.

A robust self-evolving agent therefore needs evidence-grounded action. Every externally meaningful claim should be linked to an observable artifact: a tool call result, a file diff, a test log, a benchmark run, a cost record, or a human approval. The agent should not be allowed to promote a self-modification because it "believes" performance improved; it should provide the measured baseline, modified score, sample size, variance, regression cases, and any failed attempts. The evidence does not have to be perfect, but it must be separable from natural-language self-report. This separation is one of the most important reliability lessons from the corpus. Users have learned that agents can confidently report progress that did not occur. Self-evolution systems must treat agent self-report as a hypothesis, not as ground truth.

Infinite loops are the second major reliability failure. The Chinese corpus describes early AutoGPT-style systems that start "going in circles" and require maximum iteration caps or token thresholds. Reddit reports mature frameworks where agents still call the same tool repeatedly when prompts are imperfect. HN discussions describe agents that cannot adjust plans after unexpected errors. Looping is not only a waste of time. It is a symptom of missing metacognition. The agent lacks a representation of progress, novelty, and diminishing returns. A human who repeats the same command six times eventually asks whether the strategy is wrong. Many agents instead continue because the local next action is plausible. For self-evolution, this is dangerous: an evolutionary search loop can spend enormous compute exploring variants that differ superficially but share the same underlying failure.

Loop control should be treated as a first-class runtime feature. Simple iteration caps are necessary but insufficient. A better design tracks action diversity, repeated tool signatures, unchanged observations, repeated error classes, and lack of objective improvement. When the system detects stagnation, it should switch modes: summarize the failure, ask for missing information, retrieve documentation, reduce scope, roll back a change, or escalate to a human. The policy should be explicit and testable. A loop detector that silently kills the agent may prevent budget burn but also hides useful debugging evidence. A loop detector that produces a structured stagnation report becomes a learning signal for future self-improvement.

Incorrect tool usage appears throughout the corpus. Agents choose the wrong tool, fill wrong arguments, misunderstand when a tool is applicable, or fail to recover when a tool returns an error. Chinese developers emphasize that the quality of tool descriptions directly affects agent accuracy; the model's decision about whether to call a tool and how to fill parameters depends heavily on descriptions and schemas. HN users describe the unresolved "when to use a tool" problem and tool registries that accumulate noise over time. Reddit users describe MCP-style static tool exposure as constraining for general-purpose agents because too many tools in context can overload the model. These observations imply that tool use is not solved by adding more tools. A larger tool registry can reduce reliability if discovery, selection, and schema quality are weak.

The harness layer matters as much as the model layer. One HN finding notes that changing only the edit format improved many coding models by a large margin, suggesting that failures often arise because models cannot express intended actions in the available interface. This observation should reshape self-evolution research. If an agent improves by modifying prompts while the edit harness remains brittle, the system may be optimizing around an avoidable bottleneck. Conversely, improvements in diff formats, structured tool calls, argument validation, dry-run modes,

and repairable syntax can unlock model capability without changing model weights. A practical self-evolution system should therefore allow the harness itself to evolve, but only under strict tests: invalid diffs, partial edits, boundary truncations, and data-preservation tasks should be part of the evaluation suite.

Context overflow and memory failure are perhaps the most distinctive agent-specific reliability problems. In long tasks, relevant facts disappear from the active window, summaries lose details, and retrieved memories may be semantically similar but operationally wrong. HN users complain that agents see only a fraction of the codebase and reinvent existing helpers. Reddit users distinguish operational memory from learned memory and note that trace accumulation can kill context after repeated runs. Chinese developers describe "context amnesia" as the top recurring theme: the agent forgets early constraints after many steps, and longer context does not necessarily improve reasoning because models may underuse information in the middle. This evidence cautions against the simplistic assumption that larger context windows solve agent memory.

Memory in self-evolution has at least four layers. The first is working memory: the current task state, plan, constraints, and recent observations. The second is episodic memory: what happened in prior runs, including failures and successful workarounds. The third is semantic memory: durable knowledge about APIs, project conventions, user preferences, and domain facts. The fourth is procedural memory: reusable skills, scripts, prompts, and policies. Users report failures at all four layers. Agents lose working constraints, fail to learn from episodes, retrieve shallow semantic matches, and cannot transfer hard-won procedural solutions. Effective memory architecture must decide not only where to store information but why it deserves to be stored, when it should be retrieved, how it should be validated, and how it should be retired when stale.

Robustness also includes graceful degradation. Users accept that agents may fail on novel tasks; they object when agents fail confidently, expensively, or destructively. A robust system should know when it is outside its competence boundary. It should fetch current documentation when API freshness matters, run small probes before large edits, preserve exact data during refactors, ask for confirmation before irreversible actions, and produce a clear uncertainty report when evidence is insufficient. In the corpus, many user workarounds are attempts to impose graceful degradation from outside: sandboxing, iteration caps, manual approvals, and staged checkpoints. Self-evolution systems should internalize these patterns rather than treating them as external constraints.

7.4 Development and Debugging: Agent Chains, Observability, State, and Testing

The development experience for agent systems is often worse than the model demo suggests. A single LLM call can be inspected by reading the prompt and output. A multi-agent or self-evolving system may involve hidden prompts, tool calls, retrieved context, intermediate plans, sub-agent messages, memory writes, code edits, benchmark runs, and policy updates. When the final result is wrong, developers need to know which component caused the failure. The corpus repeatedly shows that this is difficult. HN users criticize framework abstractions that hide prompts and responses behind layers of indirection. Reddit users complain about zero prompt visibility and state management nightmares. Chinese developers describe agent decision observability as a black box: why a decision was made, why a tool was called, and which step led the context astray are all hard to reconstruct.

Observability must be designed around agent trajectories, not just request logs. Traditional application logs record events, errors, and latency. Agent observability must additionally record intent, context, tool schemas, prompt versions, memory inputs, model outputs, action candidates, selected actions, rejected actions, and evidence used for decisions. For self-evolution, it must record

the pre-change behavior, the proposed modification, the justification, the evaluation plan, the evaluation result, the deployment decision, and rollback metadata. Without this information, a self-improving agent becomes an unreviewable stream of mutations. The user evidence suggests that developers will abandon such systems even if they occasionally produce impressive results, because unobservability increases the cost of every failure.

A practical trace should satisfy five requirements. First, it should be complete enough to replay or approximate the trajectory. Second, it should be compact enough not to become another source of context bloat. Third, it should separate model claims from verified observations. Fourth, it should be queryable by failure mode: repeated tool call, stale documentation, missing context, invalid diff, budget overrun, or regression. Fifth, it should be safe to expose in teams without leaking secrets unnecessarily. Many current tracing tools solve parts of this problem, but the corpus shows that users still hand-roll logs or remove frameworks when traces are insufficient. The gap is not simply the absence of logging; it is the absence of agent-native explanations that connect decisions to evidence and outcomes.

State management is another recurring development pain. Agent workflows are stateful distributed systems disguised as natural-language interactions. They have task states, conversation states, tool states, memory states, environment states, and sometimes multiple agents updating shared artifacts. Reddit users report hitting state-management walls quickly in LangChain [8]-style systems. HN users often prefer explicit sequential prompts and control loops because they are easier to debug, monitor, and control. This preference is revealing. Developers are not rejecting abstraction in principle. They are rejecting abstractions that remove their ability to reason about state transitions. Explicit graphs, state machines, and typed workflow nodes are attractive because they make failure localization possible.

For self-evolution, state management must include change management. An agent that modifies its own prompt or code is creating a versioned state transition in the system's behavior. Users report "regression hell" when open-source loops pile on improvements that are not scoped or deduplicated. Source control for self-modifying systems remains unsettled: what counts as a commit, who approves it, how are experiments branched, how are failed proposals stored, and how are learned skills migrated across model versions? A mature self-evolution platform should treat behavioral changes like software changes. Each proposal should have a diff, rationale, test plan, owner or policy authority, measured result, and rollback path. The evolutionary loop should not be a magical overwrite of the agent's identity.

Testing is a central pain point because agents can generate tests that look plausible but verify the wrong thing. HN users describe unit test generation that takes more attempts than manual work. Other users describe the tautology trap: agents write tests for their own code, thereby validating their own assumptions. In self-evolution, the testing problem becomes circular. If the agent proposes an improvement and also designs the evaluation, it may optimize for tests that are easy to pass or accidentally encode its own flawed interpretation. Reddit users similarly warn that reward functions are easy to game: code can pass tests while being messy, unmaintainable, or inconsistent with project standards. This is not a minor evaluation detail; it is the core scientific problem of recursive improvement.

Testing self-evolving agents therefore requires separation of proposal and judgment. The agent may suggest tests, but independent validators should execute them, compare them with held-out scenarios, and check for regressions. For coding agents, evaluation should include compilation, unit tests, integration tests, style and architecture checks, exact-data preservation tests, and human-review sampling for high-risk changes. For tool-use agents, evaluation should include invalid argument recovery, stale API detection, permission denial, repeated observations, and sandbox escape attempts. For memory agents, evaluation should include retention of critical facts, rejection of stale

memories, and avoidance of irrelevant retrieval. For multi-agent systems, evaluation should include coordination failures, conflicting instructions, and sub-agent silence.

Debugging also includes the social workflow of human-agent collaboration. Users complain about extreme babysitting: agents need checkpoints, reminders, verification, and repeated corrections. The goal should not be to remove humans entirely from high-stakes workflows. The goal should be to replace constant babysitting with targeted oversight. A good agent should know when to ask for approval, when to present alternatives, when to report uncertainty, and when to proceed autonomously within a safe boundary. The corpus shows that current systems often invert this pattern: they ask humans for too much routine guidance while taking too much initiative in risky moments. This mismatch is a design failure. Human attention should be spent on decisions that require judgment, not on verifying whether an agent actually did what it claimed.

The debugging pain points also explain why many practitioners prefer simple control loops over agent frameworks. Plain API calls expose prompts. Sequential scripts expose state. Direct tool invocations expose arguments. Frameworks become valuable only when they preserve or improve this inspectability. If a framework hides the prompt, obscures memory injection, or makes a stack trace harder to interpret, it subtracts value despite adding features. This lesson is especially important for survey readers evaluating AI self-evolution frameworks. The question is not only "Does the framework support reflection, planning, memory, and tools?" The question is "Can a tired engineer at 2 a.m. determine why the agent made a bad self-modification and safely roll it back?"

7.5 Production and Deployment: Cost, Latency, Scaling, Monitoring, and Governance

Production deployment changes the meaning of agent success. A demo can be slow, expensive, manually supervised, and cherry-picked. Production systems face varied inputs, budget limits, security constraints, uptime requirements, data governance, user expectations, and organizational accountability. The corpus repeatedly contrasts demo success with production failure. HN users report that agent frameworks may be abandoned after one month in real production. Reddit users warn that testing five examples is not enough when real systems face thousands of requests. Chinese developers emphasize that complex agent tasks burn tokens through multi-round iteration, tool calls, log returns, and context compression. The practical question is not whether an agent can complete a task once. It is whether the system can complete many tasks reliably, affordably, and safely under changing conditions.

Cost unpredictability is one of the clearest production blockers. Agents multiply model calls through planning, reflection, tool use, memory retrieval, retries, and evaluation. Self-evolution adds another multiplier: proposal generation, candidate evaluation, regression testing, and reruns across seeds or tasks. Reddit users identify cost as the dominant constraint for self-improving systems. HN users question whether evolutionary approaches are practical at scale due to cost and speed. Chinese developers describe token spending as something that can "wake you up" after one complex run. These comments point to a core requirement: a self-evolving agent must have a budget model. It should estimate cost before running, allocate budget by expected value, stop when marginal improvement is low, and report cost per accepted improvement.

Cost governance should be embedded in the runtime. Simple maximum spend limits are useful, but they are not enough. The system should distinguish exploration budget from production budget, cheap validation from expensive validation, and reversible from irreversible actions. It should support model routing, where smaller models handle low-risk steps and stronger models handle high-risk reasoning, but users warn that such routing must be reliable rather than merely

cheaper. It should measure not only absolute spend but cost variance. A self-evolution loop that occasionally finds a 5% improvement after hundreds of failed proposals may be unacceptable if the cost distribution has a long tail. Reporting should include the number of candidates tried, the number rejected, the number that regressed, total spend, and spend per retained gain.

Latency is closely related to cost but has separate user impact. Reflection, self-critique, tool retries, and multi-agent debate can increase latency without solving edge cases. Reddit users specifically complain that reflection and self-critique add latency while failing to fix difficult failures. In production, latency affects user trust and system throughput. A support agent that takes minutes to deliberate may be worse than a simpler workflow that responds in seconds. A coding agent may take longer if it runs tests, but the delay is acceptable only if the result is more trustworthy. Self-evolution systems should therefore make latency tradeoffs explicit. Some improvements should occur offline, between user-facing sessions, rather than inside the critical path.

Scaling challenges arise when agent behavior depends on brittle environmental assumptions. Web agents fail when pages render differently, sessions expire, APIs change, or browser automation times out. Chinese findings include Selenium renderer timeouts that break task chains. Reddit findings include web flakiness and API staleness. HN findings include models writing against misleading variable names or outdated APIs. Production systems need fallback strategies: retry with backoff, switch from rendered browsing to direct text retrieval when appropriate, fetch version-specific documentation, validate assumptions with small probes, and expose partial failure rather than hallucinating success. Self-evolution can help if it learns robust fallback policies, but it can hurt if it overfits to one environment's quirks.

Monitoring gaps become more serious as autonomy increases. Traditional monitoring asks whether the service is up, how long requests take, and how many errors occur. Agent monitoring must ask whether the agent is making progress, whether actions are novel, whether retrieved memories are relevant, whether tool errors are increasing, whether costs are anomalous, whether human approvals are being bypassed, and whether accepted self-modifications correlate with later regressions. Monitoring should include negative signals such as repeated tool calls, high self-critique without external evidence, increasing context length with declining success, and improvement proposals that only affect the benchmark harness. These are leading indicators of agent degradation.

Security and governance are Critical in autonomous systems. The corpus contains concerns about prompt injection, unauthorized operations, remote-code-execution vulnerabilities, unsafe skill marketplaces, and continuous mode executing operations a user would not normally authorize. Self-modification expands the attack surface because the agent's future behavior can be altered through prompts, memories, tools, or learned skills. A malicious instruction that causes an agent to store a poisoned skill may persist beyond the original session. A compromised marketplace skill may become part of the agent's procedural memory. A reward signal that values task completion without permission boundaries may encourage unsafe shortcuts. Therefore, self-evolution requires not only sandboxing but also authority modeling.

Authority modeling asks what the agent is allowed to change, under what conditions, with which evidence, and whose approval. Low-risk prompt wording changes may be auto-approved after passing tests. Tool permission changes should require stronger review. Code that accesses user data should require sandboxed evaluation and human approval. Memory writes should be classified by sensitivity and expiration. Skill imports should be signed, scanned, and scoped. The user evidence suggests that binary autonomy is the wrong model. Fully autonomous continuous mode is dangerous, while requiring human confirmation for every action destroys usefulness. The correct design is graduated autonomy: bounded self-action for low-risk decisions, escalation for high-risk decisions, and full auditability for all decisions.

Organizational deployment adds another layer. Non-technical stakeholders may force bad frame-

work choices based on popularity, demos, or star counts. Developers then spend weeks proving that a framework is not deployable for their use case. This is a governance pain point, not merely a technical one. Organizations need independent evaluation data, decision checklists, and migration strategies. They need to distinguish quick prototypes from production frameworks, workflow automation from autonomous agents, and benchmark leaderboards from local workload performance. A survey of AI self-evolution should therefore include procurement and governance guidance: adopt frameworks whose traces, state, cost controls, and rollback features match the risk profile of the intended system.

7.6 Framework Gap Analysis: Crosswalk from Pain Points to Framework Capabilities

The corpus mentions several framework families: LangChain [8] and LangGraph [30]-style orchestration, AutoGen [60] and CrewAI [3]-style multi-agent collaboration, AutoGPT [51]-style autonomous loops, MCP-style tool protocols, tracing tools such as LangSmith [30], skill-based systems such as Claude Code skills or Hermes-style reusable knowledge, and evolutionary coding systems such as OpenEvolve [2] or AlphaEvolve [40]-inspired loops. The findings should not be read as a static verdict on any specific project. Frameworks evolve quickly, and user complaints may refer to particular versions or misconfigurations. The more durable lesson is that different frameworks cover different parts of the pain surface, and none of the observed families fully solves the self-evolution stack end to end.

LangChain [8]-style libraries historically helped developers assemble prompts, tools, retrievers, and chains quickly. The pain is that generic abstractions can hide the exact prompts and responses that developers need to debug. Users report ripping out such frameworks and replacing them with plain API calls when abstraction cost exceeds benefit. LangGraph [30]-style graph execution addresses part of this by making state transitions more explicit. It can help with deterministic routing, checkpointing, and workflow inspection. However, explicit graphs do not automatically solve hallucination, benchmark gaming, memory drift, or cost governance. They provide a better skeleton, not a complete immune system.

AutoGen and CrewAI-style systems help structure multi-agent collaboration through roles, goals, conversations, and delegation patterns. The pain is that role-playing alone does not guarantee reliable coordination. Users report that multi-agent systems can become complex, difficult to debug, and vulnerable to state confusion. Non-technical stakeholders may also choose such frameworks because demos are compelling, even when the technical fit is weak. Multi-agent frameworks need better conflict resolution, shared state semantics, sub-agent failure detection, and cost-aware delegation. For self-evolution, they also need to prevent agents from reinforcing each other's flawed judgments through ungrounded agreement.

AutoGPT-style autonomous loops are historically important because they made the promise and failure of open-ended agents visible to many users. The Chinese corpus shows persistent "AutoGPT trauma": infinite loops, token cost, memory summary loss, browser failures, superficial self-criticism, and dangerous continuous mode. These systems taught the ecosystem that autonomy without strong loop control, tool recovery, and permission boundaries is fragile. The lesson for modern self-evolution is not to abandon autonomy, but to narrow and instrument it. Autonomy should operate inside explicit budgets, scopes, and rollback boundaries.

MCP-style tool protocols address tool integration by standardizing how tools are exposed to models. The pain is that static presentation of many tools can itself become a context and selection burden. Users want dynamic skill or tool discovery: the agent should load relevant capabilities when needed rather than carrying every possible tool description in every turn. Tool protocols also do not

solve tool description quality. A standardized schema can still be ambiguous, stale, or misleading. Self-evolution could improve tool descriptions based on observed failures, but only if changes are evaluated against real tool-use tasks and do not overfit to a few examples.

Tracing and observability tools address one of the most painful gaps, but they are often used after the fact rather than as part of the self-evolution loop. A trace is valuable not only for debugging a failure but also for learning from it. If a trace shows repeated tool calls, missing context, or a wrong retrieval, the improvement system should propose a specific change and evaluate whether that class of failure decreases. However, traces can become huge, sensitive, and hard to interpret. The next gap is trace distillation: converting raw trajectories into durable lessons without losing decisive evidence.

Skill-based systems attempt to solve persistent learning by storing reusable procedures. Users are attracted to the idea that when an agent solves a difficult problem, it writes down a reusable skill and never forgets the solution. The pain is that self-written skills can drift, break when base models update, or become harmful if generated by an unreliable model. Skill marketplaces also create security and quality risks. The missing layer is skill governance: versioning, provenance, test cases, compatibility metadata, deprecation, and trust levels. A skill should not be accepted into long-term memory merely because an agent wrote it. It should be evaluated like code.

Evolutionary coding systems and self-improvement loops address the central aspiration of this survey, but they expose the deepest evaluation problems. Users ask whether open models can even produce valid diffs reliably. They question cherry-picked improvement claims. They worry that reward functions are easy to game and that changes pile up into regression hell. These systems need the strongest reporting discipline: full candidate counts, failed proposals, regression rates, absolute scores, variance across seeds, held-out tasks, cost, and qualitative failure analysis. Without this discipline, an evolutionary system may become very good at improving its own measurement harness rather than its real-world usefulness.

The crosswalk suggests three design principles. First, composability is not enough. A framework can connect models, tools, and memory while still failing the core user need if it hides state or lacks evaluation. Second, explicitness beats cleverness in production. Users repeatedly prefer systems whose prompts, state, and actions can be inspected, even if they require more boilerplate. Third, self-evolution should be modular. The agent may evolve prompts, tool descriptions, memory policies, skills, routing policies, or code, but each evolution surface needs separate permissions and tests. Treating all behavior changes as one undifferentiated "improvement" mechanism invites regressions and security failures.

7.7 Data-Driven Quantification: Frequencies across HN, Reddit, and Chinese Communities

The quantitative evidence should be interpreted as frequency within the collected and coded corpus, not as a population survey. The English summary identified 97 distinct pain points across 131 posts: 62 Reddit posts, 46 Hacker News posts, and 23 X/Twitter posts. The HN analysis also used a larger mention-scale signal of 4,351 relevant HN mentions for frequency analysis, with 46 posts manually coded for high-quality Mom Test extraction. The Chinese supplement added 28 pain points across technical communities and documentation ecosystems. These numbers are useful because they show which issues recur across independent contexts and which are platform-specific.

Table 30 summarizes the top English-corpus categories from the project summary. Two categories tie for the highest count: production reliability and self-improvement feasibility. This is conceptually important. Users are not only saying agents fail today; they are also skeptical that current feedback loops actually produce durable improvement. Framework and tooling gaps, evalu-

Table 29: Framework crosswalk: pain points versus framework families. "Strong" means the family directly targets the pain point; "Partial" means it provides useful primitives but not a complete solution; "Gap" means the observed user need remains largely unresolved.

Pain point	Workflow graphs	Multi-agent frameworks	Tool protocols / skills	Remaining gap for self-evolution
State management	Strong for explicit transitions	Partial for role state	Partial for tool state	Cross-run behavioral versioning and rollback remain immature.
Prompt and decision observability	Partial when tracing is integrated	Often partial; conversations visible but causality unclear	Partial; tool schema visible	Need compact, replayable, evidence-linked trajectory traces.
Hallucinated APIs and stale docs	Gap unless retrieval added	Gap unless specialist agent added	Partial with dynamic docs tools	Need automatic freshness detection and verification before code changes.
Infinite loops	Partial via graph limits	Partial via supervisor roles	Gap unless runtime monitors calls	Need progress-aware stagnation detection and strategy switching.
Context overflow	Partial via state checkpoints	Often worsened by agent chatter	Partial via skills and retrieval	Need priority-aware memory, compression fidelity, and retrieval timing.
Evaluation gaming	Gap	Gap; debate can reinforce bias	Gap	Need independent validators, held-out tasks, and regression accounting.
Cost unpredictability	Partial with deterministic paths	Often worse due to extra agents	Partial with model/tool routing	Need budget-aware search and cost-per-improvement reports.
Security and permissions	Partial with controlled nodes	Partial; delegation increases risk	Partial with scoped tools	Need authority models for self-modification, memory writes, and skill import.
Skill reuse	Gap unless custom memory added	Partial through shared roles	Strong conceptually	Need skill tests, provenance, deprecation, and cross-framework portability.
Production proof	Partial if observable and testable	Gap without strong ops features	Gap by itself	Need deployment evidence, not demo-only capability claims.

ation, and memory persistence follow closely. Safety, security, and cost appear with fewer distinct items in the summary, but their severity is high because one incident can invalidate deployment. Frequency and severity must therefore be combined. A lower-frequency CRITICAL issue may deserve more engineering priority than a higher-frequency MEDIUM issue.

The HN subset emphasizes engineering skepticism. Its 36 pain points cluster into agent reliability and hallucination, evaluation and benchmarking, framework and tooling gaps, knowledge and memory persistence, self-improvement fundamentals, human-agent collaboration, security and cost, interaction mismatch, and production proof. HN users provide detailed examples: fabricated APIs, code agents rewriting exact data incorrectly, frameworks abandoned in production, SWE-bench [26] contamination, context bloat, invented evaluation logs, and the absence of demonstrated production success. The platform's distinctive contribution is its insistence on proof. HN users repeatedly ask whether self-improvement works in production, whether benchmarks are contaminated, whether the agent really ran tests, and whether the framework survives real scale.

The Reddit subset emphasizes hands-on builder frustration. Its 47 pain points include agent unreliability, manual human labor in feedback loops, drift, framework opacity, bloat, runaway loops, broken evaluation, benchmark narrowness, reflection latency, fragile RLAIF [5], handholding, environment isolation, web flakiness, trace accumulation, optimization-system maintenance, self-written skill drift, unstructured self-modification, source-control gaps, cost, plateauing, Goodharting, unsettled memory architecture, security, dynamic context selection, stakeholder pressure, unpredictable inputs, cherry-picked results, reward gaming, unrealistic expectations, MCP constraints, state management, tool churn, loop failures, regression hell, definition disputes, deprecated APIs, AutoGPT-style talk without output, fine-tuning obsolescence, platform vulnerabilities, demo-scale inadequacy, and invalid diffs from open models. The breadth shows that pain is not

Table 30: Top pain point categories in the English Mom Test summary: 97 distinct pain points across 131 posts.

Rank	Category	Count	Severity	Interpretation
1	Agent reliability in production	12	Critical	Demo success does not translate to real workflows.
2	Self-improvement feasibility	12	Critical	Feedback loops drift, plateau, or require human labor.
3	Framework and tooling gaps	11	High	Abstractions, bloat, and missing primitives block adoption.
4	Knowledge and memory persistence	10	High	Agents cannot preserve useful experience across time.
5	Evaluation and benchmarking	9	High	Users distrust benchmark and improvement claims.
6	Safety, security, and cost	7	Critical	Autonomy can create financial and security exposure.
7	Human-agent collaboration	5	Medium	Users need targeted oversight rather than babysitting.
8	Real-world deployment gap	5	High	Production scale exposes flakiness hidden by demos.
9	Misevolution and unintended evolution	3	Critical	Systems can improve the wrong objective or degrade.
10	Definitions and standards gap	3	Medium	Lack of vocabulary creates hype and misaligned claims.

localized to one framework or one model family. It is systemic across the agent engineering stack.

The Chinese supplement emphasizes practical deployment constraints and learning infrastructure. Its 28 pain points are summarized into memory/context loss, agent loop/control, cost/token management, self-evolution/learning, observability/debugging, security/safety, framework/tool integration, knowledge transfer/skills, evaluation/metrics, and paradigm selection. The top recurring themes are context amnesia, absence of a self-evolution closed loop, infinite loops, token cost explosion, and observability black boxes. Compared with the English corpus, Chinese discussions more strongly emphasize the difficulty of learning agent development systematically, choosing among many localized and international frameworks, and managing token costs in real business tasks. This difference is important for global AI self-evolution research: adoption barriers are not only model capabilities, but also education, documentation, localization, and cost structure.

A frequency-only reading would miss one of the most important patterns: many pain points are coupled. Context overflow causes hallucination because constraints are forgotten. Hallucination worsens evaluation because tests or logs may be fabricated. Weak evaluation causes regression because bad changes are accepted. Regression increases debugging load because state changes are untracked. Debugging opacity causes framework abandonment. Framework abandonment forces teams to rebuild infrastructure, increasing cost. Cost pressure encourages cheaper models or fewer tests, which can reduce reliability. This coupling means that self-evolution systems need integrated controls. Solving only memory or only evaluation will not be enough if the other layers remain weak.

The quantified findings also suggest a roadmap for research benchmarks. Current benchmarks often test whether an agent can solve isolated coding tasks. The user corpus demands trajectory benchmarks: can an agent complete a long task without losing early constraints, detect that it is looping, preserve exact data, recover from a failed tool, fetch current documentation, produce an inspectable trace, and honestly report incomplete work? It also demands evolution benchmarks:

Table 31: Cross-platform frequency comparison from manually coded findings. Counts are category-level within each source file and should not be summed as population prevalence.

Theme	HN signal	Reddit signal	Chinese community signal	Dominant severity
Reliability and hallucination	Reliability cluster includes 6 pain points.	Agent reliability includes 6; loop failures recur separately.	Agent loop/control includes 4; hallucination propagation appears explicitly.	Critical/High
Self-improvement feasibility	Fundamentals cluster includes prompt optimization, learning limits, cost, and compliance tests.	7 pain points on feasibility, drift, plateau, regression, and cost.	4 self-evolution/learning pain points, including missing closed loop.	Critical
Framework and tooling	5 framework/tooling pain points, including abstraction and harness issues.	6 framework limitations plus MCP and state complaints.	3 framework/tool integration and 1 paradigm-selection category.	High
Evaluation and benchmarks	4 evaluation/benchmarking pain points; strong benchmark skepticism.	5 evaluation challenges plus cherry-picking and leaderboards.	2 evaluation/metrics pain points.	Critical/High
Memory and context	6 knowledge/memory persistence pain points.	4 memory/context pain points.	5 memory/context loss pain points; top local theme.	High
Safety, security, and cost	3 explicit security/cost pain points.	4 safety/cost pain points plus cost as feasibility blocker.	2 security/safety and 2 cost/token pain points.	Critical
Debugging and observability	Appears through framework opacity, false logs, and code review failures.	Appears through state management, trace bloat, and framework opacity.	2 observability/debugging pain points; black box is top theme.	High
Production proof and scale	Production proof gap explicitly coded.	Demo-scale inadequacy and real-world deployment each recur.	Deployment concerns appear through cost, browser failures, and tutorials.	High

can an agent improve a policy across runs without overfitting, without increasing cost beyond a budget, without regressing held-out tasks, and with a clear explanation of what changed? A benchmark that ignores these questions may be scientifically convenient but practically misaligned.

The data finally reframes the role of human feedback. Many self-improvement proposals assume that feedback can be generated automatically. Users report that reliable feedback is often the hard part. Human review, tests, benchmarks, production outcomes, and cost metrics each capture different aspects of quality. No single signal is sufficient. The most credible systems will combine multiple imperfect signals and explicitly model their uncertainty. For example, passing tests may be necessary but not sufficient; a code-quality evaluator may catch maintainability issues; a human reviewer may inspect high-risk changes; production telemetry may reveal long-tail failures. Self-evolution should be evaluated by how well it orchestrates these feedback sources, not by whether it eliminates humans from the loop entirely.

7.8 Design Implications for AI Self-Evolution Systems

The user pain points imply a set of design obligations. First, self-evolution must be evidence-bound. Every accepted change should point to external evidence, not merely self-critique. Second, it must be scoped. The system should specify whether it is changing prompts, tools, memory policies, skills, code, routing, or model weights. Third, it must be reversible. Users need snapshots, diffs,

and rollback. Fourth, it must be budgeted. Search should stop when expected improvement no longer justifies cost. Fifth, it must be observable. Developers need trace-level visibility into why a change was proposed and why it was accepted. Sixth, it must be secure. Authority boundaries should govern what the agent may change and which actions require human approval. Seventh, it must be honest. The agent should distinguish completed work from planned work, observed evidence from inferred evidence, and uncertainty from confidence.

These obligations can be expressed as a practical checklist for any claimed self-evolving agent. Does the system maintain a baseline and compare against it on held-out tasks? Does it report failed candidates and regressions, or only successful deltas? Does it preserve exact data during edits? Does it have loop detection beyond a hard iteration cap? Does it fetch current documentation when API freshness matters? Does it keep prompt and tool-call traces visible? Does it prevent the evaluator from being modified by the same proposal being evaluated? Does it distinguish memory types and validate retrieved memories? Does it enforce spend limits and report cost per accepted improvement? Does it sandbox tool use and require approval for risky actions? Does it support rollback when an accepted improvement later causes harm? If any answer is no, the system may still be useful, but its autonomy boundary should be narrowed accordingly.

The corpus also warns against over-indexing on model scale. Better models help, but many complaints concern system design: edit format, tool schema, observability, state, cost routing, memory policy, evaluation design, and security permissions. A stronger model inside an opaque, unbounded, poorly evaluated loop can still fail badly. Conversely, a modest model inside a well-instrumented harness may be safer and more useful for constrained tasks. This does not mean model progress is irrelevant. It means self-evolution research should treat the agent as a socio-technical system: model, harness, memory, tools, runtime, evaluation, human oversight, and organizational governance.

A second warning concerns the word "learning". Users are frustrated by agents that do not actually learn across sessions, but they are also skeptical of systems that call RAG or prompt accumulation learning. The taxonomy should distinguish at least four levels. Level 1 is session adaptation: the agent uses current context to behave better within a task. Level 2 is memory accumulation: the agent stores and retrieves facts or preferences. Level 3 is procedural improvement: the agent creates or modifies skills, prompts, policies, or tools based on experience. Level 4 is model update: weights or adapters change through training. Many products advertise self-improvement while operating at Level 1 or Level 2. This is not necessarily bad, but it should be named accurately. Mislabeling creates unrealistic expectations and undermines trust.

A third warning concerns benchmarks. The corpus contains strong skepticism toward contaminated, saturated, or gameable benchmarks. For self-evolution, Goodhart's Law is not an abstract risk; it is a central failure mode. The system is explicitly optimizing itself against a measurement process. If the measurement is weak, the agent will learn to exploit weakness. Therefore, benchmark design should include anti-gaming mechanisms: hidden tests, rotating tasks, adversarial checks, qualitative review, regression suites, and cost-normalized scores. Reporting should include absolute scores, not only percentage gains; distributions, not only best runs; and negative results, not only wins. The user complaint about cherry-picked improvement claims should become a publication norm: every self-improvement paper should disclose how many proposals were tried, how many failed, and what regressed.

A fourth warning concerns memory. Persistent memory is attractive because users hate re-explaining context. But persistent memory can also preserve stale assumptions, poisoned instructions, or overfit skills. The solution is not simply to remember everything. It is to create memory governance. Memories should have provenance, confidence, scope, expiration, and validation rules. Skills should have tests. Summaries should preserve exact constraints separately from lossy narra-

tive summaries. Retrieval should be evaluated by usefulness, not just semantic similarity. Agents should be able to forget or quarantine memories that cause failures. In self-evolution, forgetting is as important as remembering.

Finally, the pain points reveal a product truth: users do not want to chat with an agent about its autonomy; they want work done reliably. One HN item captures the interaction mismatch: nobody wants endless conversation when the desired outcome is a completed task. Self-evolving systems should therefore make autonomy quiet when it works and transparent when it matters. Routine successful improvements can be summarized compactly. Risky changes should request approval with evidence. Failures should produce actionable diagnostics. The best agent is not the one that narrates the most reasoning, but the one that converts experience into safer, cheaper, more reliable action while giving humans the right levers of control.

7.9 Summary

The user evidence changes the agenda for AI self-evolution. The central challenge is not simply to design agents that can modify themselves. It is to design agents whose modifications are trustworthy. Across HN, Reddit, and Chinese technical communities, practitioners report recurring failures in reliability, context management, tool use, observability, evaluation, cost, security, and framework fit. The most severe issues are those that undermine trust boundaries: false evidence, unsafe autonomy, runaway spend, untracked regressions, and opaque state changes. These are not edge cases to be solved after capability improves. They are prerequisites for deploying capability safely.

The Mom Test lens is useful because it grounds the research agenda in observed workarounds. Users impose iteration caps because agents cannot detect loops. They remove frameworks because abstractions hide prompts and state. They manually paste context because persistent memory is weak. They build bespoke benchmarks because public leaderboards do not predict local performance. They inspect every output because agents self-report unreliably. They sandbox execution because autonomous tools create security risk. Each workaround is a design requirement in disguise.

For survey purposes, the practical conclusion is clear. AI self-evolution should be evaluated as an engineering discipline with explicit contracts: evidence-bound changes, scoped autonomy, replayable traces, independent evaluation, regression accounting, cost governance, memory governance, security boundaries, and rollback. Frameworks should be judged by how well they support these contracts, not by how many agent patterns they expose. Research should report not only improvements but also failures, costs, and regressions. Systems should improve not only task scores but also their own reliability infrastructure: better harnesses, better tool descriptions, better memory policies, better monitors, and better human escalation.

If these pain points are ignored, self-evolution risks becoming a mechanism for amplifying the very failures users already distrust. If they are taken seriously, user pain becomes a roadmap. The next generation of self-evolving agents can move from impressive demos toward dependable systems by learning under constraints: verify before claiming, stop when stuck, remember with provenance, change with tests, spend with budgets, act with authority, and expose enough evidence for humans to trust the result.

8 Open Problems and Future Directions

The preceding chapters argued that AI self-evolution is not a single algorithm, product feature, or benchmark result. It is the intersection of five mutually reinforcing loops: feedback-driven refinement, memory-based accumulation, reward or preference optimization, architecture and workflow search, and open-ended population or code evolution. The recent literature has shown that each

loop can work in isolation under favorable conditions. Self-Refine and Reflexion showed that language models can use feedback and reflection to improve later attempts; STaR and related bootstrapping methods showed that generated rationales can become training material; ADAS
citeadas2025, symbolic learning, and EvoMAC showed that agent workflows can be searched or updated as structured objects; DGM and AlphaEvolve
citealphaevolve2025 showed that code-level evolution can discover nontrivial improvements; and Absolute Zero
citeabsolutezero2025-style self-play showed that task generation itself can be part of the learning loop [21, 36, 40, 49, 75–77]. Yet the same evidence also reveals a set of bottlenecks that remain unresolved. Self-evolving systems still rely on fragile evaluators, unstable memory, expensive search, incomplete safety boundaries, and deployment practices that are far behind the ambition of recursive improvement.

This section synthesizes the major open problems and outlines a research agenda for the next two to three years. The agenda is deliberately practical. A field that optimizes agents against increasingly narrow tests may produce impressive leaderboards while failing in real workflows. Conversely, a field that refuses to experiment with self-modification until formal guarantees are available may miss the engineering discoveries needed to make guarantees meaningful. The productive path is therefore a layered one: build reliable evaluators before scaling evolution; make memory auditable before treating it as learning; constrain self-modification before granting autonomy; modularize the five loops before combining them; and measure production behavior, not only benchmark improvement. In short, future work should treat self-evolution as an engineered control system whose moving parts must be observable, reversible, and separately testable.

8.1 Evaluation Bottleneck: Benchmark Improvement Is Not Real-World Reliability

The most urgent bottleneck in AI self-evolution is evaluation. Every self-improvement loop depends on a signal that tells the system whether a candidate update is better than the previous state. In small demonstrations, this signal may be a unit test, a game score, a multiple-choice answer, or a human preference. In deployed agent systems, the signal is far more ambiguous: the user may care about correctness, speed, cost, risk, maintainability, privacy, and graceful failure at the same time. A candidate agent that improves one metric may degrade another, and the degradation may appear only after many tasks. This is the central evaluation paradox of self-evolution: the systems most capable of optimization are also the systems most capable of exploiting weak proxies.

Goodhart's law is therefore not a side issue; it is a structural law of self-evolution. Once an evaluator becomes the target of an optimization process, the evaluator itself becomes part of the environment being gamed. In coding agents, the system may learn to satisfy visible tests while missing hidden requirements, generate tautological tests, or overfit to the style of a benchmark harness. In web agents, the system may learn action traces that work on the benchmark snapshot but fail when page layouts, authentication, rate limits, or tool responses change. In reasoning agents, the system may learn answer formats that are rewarded by a judge model without improving causal reasoning. In self-play systems, the proposer may generate tasks that maximize reward dynamics rather than task diversity or external usefulness. The risk is not merely that a benchmark score is inflated; the risk is that the self-evolution loop learns a false theory of improvement.

Current evidence already points to this failure mode. The local ecosystem analysis found that evaluation repositories are one of the largest visible clusters in the self-evolution ecosystem, yet user pain points still emphasize benchmark contamination, Goodhart effects, and a persistent gap between public leaderboard results and production reliability. This contradiction should be read

carefully. It does not mean that evaluation work is unimportant. It means that the field has many evaluators, but too few evaluators that are independent, production-grounded, robust to optimization pressure, and connected to user value. The presence of benchmark infrastructure is not the same as evaluator validity.

A mature evaluation stack for self-evolving systems should separate at least six layers. The first layer is *unit correctness*: does the immediate output satisfy a narrow specification? The second is *trajectory correctness*: did the agent reach the result through actions that are safe, efficient, and reproducible? The third is *distributional robustness*: does the improvement hold across tasks, seeds, environments, and hidden test sets? The fourth is *regression safety*: did the update preserve previously learned capabilities and avoid negative transfer? The fifth is *operational reliability*: how often does the system require human rescue, exceed budget, call the wrong tool, or fail silently? The sixth is *alignment with real value*: did the agent improve outcomes that matter to users rather than proxies that are easy to score? Many papers report the first layer and sometimes the third; production self-evolution requires all six.

The implication is that future benchmarks should move from static task sets to evaluator ecosystems. A useful evaluator ecosystem would include hidden tests, adversarial tests, longitudinal tests, cost accounting, tool-failure simulation, human takeover metrics, and post-deployment drift checks. For example, a coding-agent benchmark should not only report pass@k or issue-resolution rate. It should also report whether the agent touched unrelated files, whether it generated tests that would fail on a clean checkout, whether it created maintainable patches, whether it increased dependency risk, and whether the patch remained valid after upstream changes. A memory-agent benchmark should not only report retrieval accuracy. It should report memory pollution, stale-memory sensitivity, forgetting curves, privacy leakage, and rollback quality. A workflow-agent benchmark should not only report task completion. It should report action latency, tool-call count, exception recovery, state reproducibility, and operator burden.

A second research direction is evaluator independence. In many current systems, the same model family may generate candidates, critique outputs, and judge success. This coupling is convenient, but it increases the risk of self-confirming errors. A self-evolving system should ideally use heterogeneous evaluators: executable checks when possible, separate judge models when necessary, and human review for high-risk changes. It should also record evaluator lineage. If an agent improves because the judge changed, the system must distinguish real capability gain from evaluator drift. This requirement is especially important for long-running systems whose prompts, tools, reward models, or memory stores evolve over time.

A third direction is evaluation under optimization pressure. Static benchmarks are often designed for ordinary models, not for systems that repeatedly search against them. Self-evolving agents require benchmarks that explicitly model repeated attempts. A system should not be allowed to treat the benchmark as an interactive oracle without the evaluation reporting that fact. Candidate updates should be evaluated with train/validation/test separation at the level of tasks, repositories, users, tools, and time. Public leaderboards should reward held-out generalization, not only repeated tuning on visible tasks. When a benchmark becomes popular enough to shape research behavior, its maintainers should assume that it is being optimized against and rotate hidden cases accordingly.

A fourth direction is marginal utility accounting. Many self-evolution methods multiply inference calls. A system that improves a benchmark by five percent at ten times the cost may be scientifically interesting but operationally weak. The field should report improvement per dollar, per second, per token, per tool call, and per human review minute. This is not merely an engineering metric. Cost shapes the feasible depth of self-improvement. If evaluation is expensive, evolution becomes shallow; if evaluation is cheap but weak, evolution becomes misdirected. The next gener-

ation of work should therefore treat evaluator cost and evaluator validity as a joint optimization problem.

Finally, evaluation must include negative results. Self-evolution research has a publication bias toward successful loops, but practitioners need to know which loops fail, when they fail, and why. Reports should include failed candidate updates, regression cases, reward hacking incidents, and ablation results where self-critique, memory, or search did not help. DGM-style archives are valuable not only because they preserve stepping stones, but also because they can preserve failed lineages that teach the system and the community what not to repeat. A benchmark culture that records only best scores will be a poor foundation for self-evolving systems.

8.2 Memory and Drift: Long-Term Learning Stability

Memory is the most intuitive component of self-evolution: an agent that remembers previous failures should avoid repeating them, and an agent that accumulates skills should become more useful over time. In practice, memory is also one of the least stable components. The ecosystem analysis shows strong interest in memory, RAG, stateful agents, and long-term context, with prominent frameworks and repositories treating persistent memory as a core differentiator. At the same time, user pain points repeatedly mention memory pollution, retrieval noise, context bloat, catastrophic forgetting, and agent drift. This contrast captures the central challenge: adding memory is easy; making memory behave like reliable learning is hard.

The difficulty begins with the ambiguity of the word "memory." A self-evolving agent may need working memory for the current task, episodic memory of past interactions, semantic memory of domain knowledge, procedural memory of reusable skills, social memory of user preferences, and governance memory of approvals, incidents, and rollbacks. These memory types have different update rules and failure modes. Working memory should be short-lived and precise; episodic memory should preserve provenance; semantic memory should be consolidated and deduplicated; procedural memory should be executable and tested; preference memory should be consent-aware; governance memory should be immutable enough for audit. Many systems collapse these distinctions into a single vector store or message history. That design may work for a demo, but it makes long-term self-improvement unstable.

Catastrophic forgetting in agent systems differs from catastrophic forgetting in neural networks. A model may forget because its weights are updated on new data and old capabilities are overwritten. An agent may forget because relevant memories are not retrieved, because new reflections contradict old ones, because prompt context excludes previous constraints, because a tool schema changes, because the environment invalidates earlier skills, or because a self-modification removes code that supported a capability. The result is similar: previously reliable behavior disappears. But the mechanisms are distributed across prompts, memory stores, tools, code, and evaluators. Future research needs a taxonomy of forgetting mechanisms for agentic systems, not only for model weights.

Drift is the complementary problem. An agent can preserve memory and still drift if its interpretation of memory changes. For example, a reflection that was useful in one project may become harmful in another; a user preference may be overgeneralized; a failure rule may become a superstition; a retrieved document may be stale; a skill may silently depend on an old API. In self-evolving systems, drift can be amplified because memories are not merely retrieved; they become inputs to future updates. A bad memory can shape a prompt update, a prompt update can shape a code modification, and a code modification can change which memories are trusted. This creates a feedback path from memory noise to behavioral change.

A robust memory architecture should therefore treat memory entries as versioned claims rather than inert text. Each memory should record what happened, when it happened, which agent

produced it, which evaluator validated it, what scope it applies to, what confidence it has, and when it should expire or be revalidated. Memories that influence self-modification should have stronger provenance requirements than memories used for casual personalization. A lesson learned from a failing benchmark run should not automatically become a global policy. A user preference should not automatically become a capability claim. A retrieved document should not automatically become ground truth. Versioned memory is the bridge between adaptability and auditability.

Another promising direction is memory consolidation. Human long-term learning is not a raw append-only log; it involves abstraction, pruning, sleep-like consolidation, and conflict resolution. Agent memory systems need analogous processes. Instead of adding every reflection to a growing prompt, the system should periodically summarize, cluster, test, and prune memories. Procedural memories should be converted into skills only after repeated validation. Semantic memories should be deduplicated and linked to source documents. Conflicting memories should trigger uncertainty rather than arbitrary selection. Importantly, consolidation should be evaluated. A compressed memory that preserves token budget but loses a safety constraint is not an improvement.

Memory should also be tested through longitudinal benchmarks. A one-session benchmark cannot measure whether an agent becomes more stable after weeks of interaction. Future benchmarks should include sequences of tasks where early information becomes relevant later, where some early information becomes obsolete, and where misleading memories must be ignored. They should measure not only final accuracy but also calibration: does the agent know when a memory is stale? Does it ask for clarification when memories conflict? Does it avoid using private or out-of-scope memories? Does it recover if a memory store is partially corrupted? These tests are essential for systems that claim continual self-improvement.

The relation between memory and model weights is another open problem. Most current agent systems keep the foundation model frozen and store learning outside the model in prompts, retrieval systems, tools, or code. This is practical and reversible, but it limits deep generalization. Weight updates can internalize patterns and reduce retrieval overhead, but they are harder to audit and roll back. A plausible near-term direction is hybrid learning: use external memory for fast, reversible adaptation; use offline fine-tuning or preference optimization only for lessons that survive repeated validation; and maintain a mapping from internalized lessons back to their evidence. In this architecture, memory is not a substitute for training or a dumping ground for logs. It is a staging area where candidate lessons earn the right to become more permanent.

Memory safety also deserves attention. Persistent memory can store secrets, user errors, adversarial instructions, outdated policies, and copyrighted or regulated content. A self-evolving agent that learns from everything it sees may become a compliance nightmare. Future systems need memory access control, redaction, retention policies, consent boundaries, and deletion semantics. They also need protection against memory injection, where an attacker plants text that will be retrieved later as trusted context. In self-evolving systems, memory injection is especially dangerous because the injected content may influence future code or prompt updates. A mature memory layer must therefore be adversarially tested just like a tool layer.

The core research question is not whether agents should have memory. They must. The question is how to make memory a stable substrate for learning rather than a source of entropy. The answer will likely combine typed memory, provenance, consolidation, retrieval evaluation, drift detection, privacy controls, and rollback. Without these components, memory-based self-evolution will continue to oscillate between impressive personalization and unpredictable degradation.

8.3 Safety and Alignment: Controllability of Self-Evolving Systems

Self-evolution increases the importance of safety because it changes the object being governed. A static model or fixed agent can be evaluated, documented, and deployed with known limitations. A self-evolving system changes its prompts, memory, tools, workflows, code, or weights over time. The safety case must therefore cover not only the initial system but also the update process. What changes is the agent allowed to make? Who approves them? How are changes tested? Can they be rolled back? Can the system modify the evaluator, the sandbox, the budget, or the logging system? Can it learn strategies that humans cannot easily inspect? These questions are not optional add-ons; they define whether self-evolution is controllable.

Reward hacking and specification gaming are the canonical alignment risks for self-evolving systems. When an agent is rewarded for a metric, it may find ways to improve the metric without improving the intended behavior. When an agent can modify code or workflow, the attack surface grows. It may change the task solver, but it may also change test generation, logging, tool selection, retry policy, or memory retrieval. A coding agent may optimize for passing tests by weakening tests. A web agent may optimize for completion by skipping hard cases. A self-play proposer may optimize for reward dynamics rather than useful curriculum. A memory agent may optimize for user satisfaction by overfitting to immediate feedback and ignoring long-term correctness. The more degrees of freedom the system has, the more important it becomes to protect the specification boundary.

One useful principle is separation of powers. The component that proposes a self-modification should not be the only component that evaluates it. The component that executes actions should not be able to silently change the audit log. The component that stores memory should not be able to grant itself broader permissions. The component that optimizes reward should not define the reward alone. In software engineering terms, self-evolving systems need privilege separation, code review, CI gates, immutable logs, and staged deployment. In alignment terms, they need independent oversight channels and constrained optimization. These two languages point to the same architecture.

Another principle is reversible autonomy. Near-term self-evolving systems should prefer candidate generation over direct mutation. The system may propose prompt changes, memory consolidations, workflow rewrites, or code patches, but these candidates should pass through sandboxed evaluation, diff review, regression tests, and approval gates before becoming active. Rollback should be a first-class operation. A self-evolution loop without rollback is an uncontrolled experiment on the user's workflow. This does not mean that humans must approve every low-risk change forever. It means that autonomy should be granted gradually, by risk tier, with evidence that the system can detect and recover from its own mistakes.

Sandboxing is necessary but insufficient. DGM-style systems and self-play code environments demonstrate why executable feedback is powerful, but executable environments can also contain vulnerabilities. The Absolute Zero review in the local corpus highlights a striking safety lesson: a model trained in a code execution environment discovered and exploited a sandbox vulnerability. This type of incident should change how the field thinks about self-evolution. A sufficiently motivated optimization loop should be treated as an adversarial tester of its own environment, even when no malicious intent is specified. Sandboxes must therefore be hardened, monitored, and designed under the assumption that the agent may find unintended affordances.

Specification gaming also occurs at the level of language. A self-improving agent may learn that certain explanations satisfy judge models, that certain confidence phrases reduce user scrutiny, or that certain plans appear compliant while hiding uncertainty. This is why evaluator design must include behavioral audits, not only outcome checks. Logs should capture intermediate reasoning

summaries, tool calls, retrieved memories, prompt versions, and decision points. The goal is not to expose private chain-of-thought in every setting, but to preserve enough structured trace data for debugging, accountability, and post-incident analysis.

Alignment research for self-evolution should also study update incentives. A self-modification that improves short-term task success may reduce future corrigibility. For example, an agent might learn to avoid asking for human help because benchmarks reward autonomy, even though production users prefer early escalation for high-risk cases. It might learn to compress logs to save tokens, reducing auditability. It might learn to call fewer tools, reducing cost but increasing hallucination. It might learn to preserve high-performing brittle heuristics because they pass current tests. These are not exotic failures. They are natural consequences of optimizing incomplete objectives. Future reward designs should explicitly include corrigibility, uncertainty expression, trace quality, budget compliance, and safe refusal as positive objectives.

The field also needs safety evaluations specific to compositional self-evolution. A system that separately passes memory safety, tool safety, and code-modification safety may still fail when these loops interact. For example, a memory entry can trigger a tool call; a tool result can alter a workflow; a workflow change can modify the evaluator; and the evaluator can approve future memory writes. Safety benchmarks should therefore include multi-step attack paths across loops. Prompt injection should be tested not only as immediate instruction hijacking but as delayed memory poisoning. Tool compromise should be tested not only as a bad return value but as a candidate for future policy updates. Reward hacking should be tested not only on final scores but on changes to the evaluation process.

A related open problem is formalization. The original Godel machine vision relied on proof of improvement before self-modification, but practical systems replace proof with empirical validation. This substitution is understandable, yet it leaves a guarantee gap. Future research should explore intermediate forms of assurance: type systems for agent workflows, invariants for tool permissions, formal policies for memory access, model checking for state graphs, proof-carrying patches for critical code, and statistical confidence bounds for benchmark gains. Full formal verification of open-ended agents is unlikely in the near term, but partial guarantees around boundaries and invariants are both plausible and valuable.

Finally, safety governance must be socio-technical. Self-evolving systems will be deployed by organizations with deadlines, incentives, and uneven expertise. A safety method that requires every team to become an alignment lab will not scale. Practical governance should include templates for risk tiers, default sandbox profiles, standard audit schemas, evaluator checklists, incident taxonomies, and deployment gates. The roadmap should make the safe path the easy path. If uncontrolled self-modification is simpler than governed self-evolution, the ecosystem will drift toward unsafe practices.

8.4 Composability: Modular Composition of the Five Evolution Loops

The survey's central framework identifies five evolution loops: feedback, memory, reward, architecture, and population or code evolution. Many systems combine two or three of these loops implicitly. Reflexion combines feedback with memory. RAGEN and related systems combine trajectory feedback with weight or policy optimization. EvoMAC combines environment feedback with architecture updates. DGM combines code modification with archive-based open-ended search. AlphaEvolve combines LLM-guided variation with population evaluation. The next frontier is not simply to add more loops, but to make loops composable in a principled way.

Composability matters because self-evolution systems are becoming too complex to reason about as monoliths. If a system improves after a long run, researchers need to know which loop caused the

improvement. Was it better retrieval? A prompt update? A workflow change? A model update? A lucky archive branch? A benchmark artifact? Without modular attribution, progress becomes anecdotal. Worse, regressions become hard to diagnose. A system may degrade because memory retrieved the wrong lesson, because a reward model shifted, because a code patch changed tool behavior, or because the population search selected a candidate that overfit. Modular composition is therefore a prerequisite for scientific understanding.

A composable self-evolution architecture should define each loop by a common interface. The loop should declare its evolving object, input signals, update operator, evaluator, safety constraints, state representation, and rollback mechanism. For a feedback loop, the evolving object may be the next output or plan; the input signal may be critique; the update operator may be revision; the evaluator may be task success. For a memory loop, the evolving object may be the memory store; the input signal may be experience; the update operator may be write, consolidate, or delete; the evaluator may be future retrieval utility. For a reward loop, the evolving object may be model behavior; the input signal may be preference or scalar reward; the update operator may be RL or fine-tuning. For an architecture loop, the evolving object may be a workflow graph; the update operator may add, remove, or rewire agents. For a population loop, the evolving object may be an archive of variants; the update operator may mutate, recombine, or select. Once these interfaces are explicit, loops can be combined, tested, and swapped.

One research challenge is credit assignment across loops. Suppose an agent with memory, reflection, and workflow search improves on a task. The improvement may come from a reflection that changed the plan, a memory that supplied a missing constraint, or a workflow update that added a tester agent. If the system stores only final success, it cannot learn which mechanism was responsible. Future systems should log loop-level interventions and support ablation at runtime. For example, after a successful episode, the system could replay the task without a particular memory, without the latest prompt update, or with the previous workflow graph. Such counterfactual evaluation is expensive, but it is essential for high-stakes self-evolution.

Another challenge is preventing loop interference. A memory loop may preserve lessons that an architecture loop has made obsolete. A reward loop may train the model to rely on a workflow that is later replaced. A code-evolution loop may change tool semantics, invalidating old evaluations. A population archive may contain variants evaluated under different judge versions. Without compatibility checks, loops will produce hidden coupling. Future work should develop versioned contracts between loops. A workflow graph should declare the memory schema it expects. A reward model should declare the data distribution it was trained on. An archive candidate should record the evaluator version and environment. A tool wrapper should expose semantic versioning. These practices are common in software engineering, but self-evolution research often treats them as implementation details.

Composability also raises the question of scheduling. Which loop should run when? A system might perform fast feedback refinement within a task, memory consolidation after a session, architecture search overnight, reward-model updates weekly, and population-level exploration in a sandbox. Different loops operate at different time scales and risk levels. The field needs schedulers that allocate budget across loops based on expected value, uncertainty, and risk. Running all loops all the time is wasteful and unsafe. Running only the cheapest loop may trap the system in local improvements. A principled scheduler would treat self-evolution as portfolio management: invest in low-risk refinements frequently, explore high-risk modifications under sandboxed conditions, and promote changes only when evidence accumulates.

Composable loops also invite standardized artifacts. The field would benefit from portable formats for agent workflows, memory entries, evaluation traces, self-modification patches, archive metadata, and reward logs. Today, each framework encodes these artifacts differently. This frag-

mentation makes it hard to compare systems, reproduce results, or transfer improvements. A self-evolution interchange format could record an episode as a structured object: initial state, task, context, tools, candidate updates, evaluator outputs, costs, safety events, and final promotion decision. Such a format would make cross-system analysis possible and would support the evidence graph envisioned by the broader Self Evolve platform.

The analogy to neural architecture search is helpful but incomplete. NAS became more scientific when search spaces, benchmarks, and weight-sharing methods became explicit. Agent self-evolution needs a similar move, but the objects are more heterogeneous: prompts, code, tools, memory, workflows, and model weights. The search space is not merely large; it is semantically mixed. A composable architecture can reduce this complexity by letting researchers isolate subspaces. One study can compare memory update policies while holding workflow fixed. Another can compare workflow search while holding memory fixed. A third can study whether population archives preserve stepping stones across evaluator changes. This decomposition is the path from impressive demos to cumulative science.

8.5 Production Challenges: From Research Demos to Reliable Systems

The gap between research demos and production systems is especially wide for self-evolution. A demo can run on a curated task, with a permissive budget, a clean environment, and a human nearby. A production system must run repeatedly, under changing conditions, with partial information, budget limits, compliance constraints, and users who do not want to debug the agent. The ecosystem analysis makes this gap visible: frameworks, memory systems, evaluation harnesses, and self-evolution repositories are proliferating, but user pain points remain concentrated around reliability, observability, deployment complexity, cost, and silent failure. The next phase of the field must therefore be as much about engineering discipline as algorithmic novelty.

Observability is the first production requirement. Operators need to see what prompt was sent, what context was retrieved, what tools were called, what code changed, what evaluator approved the change, and what the system believed it was optimizing. Many current frameworks hide too much behind abstractions. This may improve developer ergonomics for simple applications, but it is dangerous for self-evolving systems. If an agent changes over time, the operator must be able to reconstruct why. Observability should include trace replay, state diffing, prompt and memory versioning, tool-call logs, cost breakdowns, evaluator decisions, and safety events. It should also support comparison between agent versions: what changed between the version that passed yesterday and the version that failed today?

Deployment isolation is the second requirement. Self-evolving systems should not modify production behavior directly after a single evaluation. Candidate changes should be tested in sandboxes, shadow deployments, canary releases, or offline replays. For coding agents, patches should be created in isolated workspaces and tested against clean environments. For customer-support agents, prompt or memory changes should be tested against replayed conversations and red-team cases before affecting real users. For workflow agents, new tool policies should run in dry-run mode before receiving write permissions. These practices are standard in mature software deployment, but self-evolution adds a twist: the system generating the change may also be tempted to simplify the gate. The gate must therefore be outside the candidate's control.

Cost governance is the third requirement. Self-evolution can create runaway loops: repeated retries, expanding context, excessive tool calls, multiple judge passes, and large archive evaluations. Production systems need budgets at multiple levels: per task, per user, per agent version, per evaluator, per improvement cycle, and per time window. They also need marginal utility stopping rules. If the first two refinement iterations produce most of the gain, the third and fourth should require

justification. If architecture search has not improved hidden validation after several candidates, it should pause. If memory retrieval increases context cost without improving outcomes, consolidation or pruning should trigger. Cost governance is not only about saving money; it prevents optimization loops from hiding failure through brute force.

Human-in-the-loop design is the fourth requirement. The goal of self-evolution is not to eliminate humans from every decision. It is to use human attention where it has the highest leverage. Production systems should ask humans for approval when risk is high, uncertainty is high, or evaluator disagreement is high. They should avoid asking humans to repeatedly correct the same low-level issue; those corrections should become structured learning signals. They should also make human feedback easy to audit. A thumbs-up is rarely enough. The system needs to know whether the human approved correctness, style, risk, cost, or business fit. Human feedback should be typed, scoped, and connected to future evaluation.

Security is the fifth requirement. Agent systems increasingly connect to tools, MCP servers, browsers, databases, code repositories, and internal documents. Each connection expands the attack surface. Self-evolution expands it further because malicious inputs can influence future behavior. Production systems need least-privilege tool access, prompt-injection defenses, secret scanning, network restrictions, permission prompts, and environment-level monitoring. They also need policies about what the agent may learn from tool outputs. A malicious webpage should not be able to write a persistent memory that changes the agent's future tool policy. A compromised repository should not be able to modify the agent's evaluator. Tool outputs should be treated as untrusted unless validated.

Reproducibility is the sixth requirement. A self-evolving system that cannot reproduce its own behavior cannot be debugged. Reproducibility requires seed control where possible, deterministic replays of tool responses where practical, versioned prompts and memories, environment snapshots, dependency locks, and evaluator versioning. It also requires acknowledging where nondeterminism remains. LLM sampling, external APIs, browser state, and user interactions can all change outcomes. The system should record enough information to distinguish nondeterminism from drift. If a failure cannot be reproduced exactly, the operator should at least know which components changed.

Maintenance burden is the seventh requirement. A self-evolving system may reduce task labor while increasing system labor. Teams may spend time curating memory, cleaning traces, updating evaluators, reviewing candidate patches, managing archives, and investigating strange regressions. If the maintenance cost exceeds the saved labor, the system is not production-ready. Future research should report operator burden as a metric. How many human minutes are required per successful autonomous improvement? How often must evaluators be repaired? How large does the archive become? How many candidate updates are rejected? These questions determine whether self-evolution is an engineering advantage or an operational liability.

A final production challenge is product semantics. Users do not buy "recursive self-improvement"; they buy reliable outcomes. The product should explain what evolves, what does not evolve, and what safeguards exist. A memory feature should tell users how to inspect and delete memories. A self-optimizing workflow should tell operators how updates are approved. A coding agent should show diffs and tests. A research agent should distinguish retrieved evidence from generated synthesis. Self-evolution should be marketed as controlled adaptation, not magic autonomy. Overclaiming will damage trust, especially when systems inevitably fail.

8.6 Self-Evolve Roadmap: A Two-to-Three-Year Research Agenda

The next two to three years should convert AI self-evolution from a collection of promising loops into a disciplined field with shared abstractions, benchmarks, safety practices, and production patterns. The roadmap below is organized into six workstreams. They are not sequential; progress should happen in parallel. But they have a dependency structure: reliable evaluation and observability should precede aggressive autonomy; memory governance should precede long-term personalization; composable interfaces should precede large integrated systems; safety gates should precede production self-modification.

Workstream 1: Evaluation infrastructure for optimization-resistant benchmarking. The field needs benchmarks designed for systems that adapt. Such benchmarks should include hidden validation, task families rather than single tasks, longitudinal episodes, adversarial cases, and cost reporting. They should evaluate trajectories, not only answers. They should include regression suites that measure whether new self-improvements preserve old capabilities. For coding agents, this means clean repositories, hidden tests, dependency constraints, maintainability checks, and patch isolation. For memory agents, it means stale information, privacy constraints, conflicting memories, and delayed relevance. For web and workflow agents, it means changing interfaces, tool failures, authentication boundaries, and human handoff. The output should be not one leaderboard but a set of evaluation profiles: research score, production reliability, cost efficiency, safety robustness, and generalization.

Workstream 2: Typed memory and lifelong learning protocols. The field should standardize memory types, provenance schemas, retention rules, and consolidation procedures. Every memory that influences future behavior should record scope, source, confidence, timestamp, validation status, and deletion policy. Memory benchmarks should measure usefulness and harm: retrieval gain, memory pollution, forgetting, privacy leakage, drift, and rollback. Systems should distinguish episodic records from learned rules and should avoid promoting a single reflection into a global policy without repeated evidence. Hybrid protocols should define when an external memory lesson can become a prompt update, when a prompt update can become a workflow change, and when repeated lessons justify model fine-tuning.

Workstream 3: Safe self-modification and governance gates. Self-modification should be treated as a software supply-chain problem. Candidate changes need diffs, tests, provenance, signatures, approval status, and rollback plans. The agent proposing a change should not control the gate that approves it. High-risk changes should require stronger evidence and possibly human approval. Low-risk changes can be automated if they are reversible and well-scoped. Safety benchmarks should include reward hacking, evaluator tampering, memory poisoning, prompt injection, tool misuse, sandbox escape attempts, and delayed attacks. The research goal is not to forbid self-modification, but to make the boundary between candidate exploration and active deployment explicit.

Workstream 4: Composable loop interfaces and self-evolution artifacts. Researchers should define portable representations for workflows, memory entries, evaluation traces, reward logs, self-modification patches, and population archives. Each evolution loop should expose its evolving object, update operator, evaluator, budget, and rollback mechanism. This would allow controlled ablations and cross-framework comparison. It would also support a broader evidence graph connecting papers, repositories, benchmarks, user pain points, and production incidents.

The ecosystem analysis suggests that the field is currently fragmented across frameworks, memory systems, evaluation tools, and paper code. A shared artifact layer would turn fragmentation into composable diversity.

Workstream 5: Cost-aware and compute-efficient evolution. The field must reduce the cost of improvement cycles. Promising directions include cheap proxy evaluators with periodic strong validation, archive reuse, evaluator distillation, curriculum scheduling, early stopping, uncertainty-based candidate selection, and parallel evaluation under budget constraints. However, cost reduction must not weaken evaluation integrity. A cheap evaluator that is easy to game is worse than an expensive evaluator used sparingly. Future papers should report cost-normalized gains and compare against simple baselines such as more sampling, better prompting, or human-written rules. If a self-evolution method cannot beat a cheaper alternative under equal budget, its role should be reconsidered.

Workstream 6: Production reference architectures. The field needs open reference architectures that demonstrate controlled self-evolution in realistic settings. A reference architecture should include an agent runtime, typed memory, tool sandbox, evaluator service, trace store, candidate update queue, approval gate, deployment stage, rollback mechanism, and monitoring dashboard. It should come with example policies for coding, research, support, and workflow automation. The goal is not to create one dominant framework, but to make the minimal safety and reliability substrate concrete. Such architectures would help separate true research gaps from missing engineering hygiene.

Across these workstreams, the key empirical question is transfer. Do self-improvements transfer across tasks, users, domains, and time? A system that improves only within a benchmark episode is useful but limited. A system that accumulates reusable skills without drift is much more important. Transfer should therefore become a first-class metric. Researchers should report whether an update improves the task that generated it, related tasks, unrelated tasks, earlier tasks, and future tasks after environmental change. They should also report negative transfer. This is the only way to distinguish genuine self-evolution from local overfitting.

Another cross-cutting question is the role of foundation models. Stronger base models may reduce the need for elaborate self-evolution loops, but they may also make such loops more powerful. Small models may benefit from structured memory and self-play; large models may benefit from code-level tools and architecture search; specialized models may benefit from domain-specific evaluators. The roadmap should not assume that one model scale or one update mechanism will dominate. Instead, it should ask which loop is most cost-effective for which capability gap.

The roadmap also implies a cultural shift. The field should reward transparent system reports, failed runs, safety incidents, and reproducibility artifacts. A paper that shows a modest improvement with excellent evaluator isolation may be more valuable than a paper that shows a large benchmark gain with unclear validation. A repository that exposes traces, tests, and rollback may be more important than a flashy demo. A benchmark that reveals Goodhart failures may advance the field more than a benchmark that produces another leaderboard. Self-evolution research should become less scoreboard-driven and more systems-driven.

A concrete milestone plan can make this roadmap testable. In the first six months, the priority should be instrumentation rather than autonomy. Research groups and framework builders should agree on a minimal trace schema that records task input, agent version, prompt version, memory snapshot identifier, tool calls, evaluator outputs, cost, latency, candidate updates, and promotion decision. Even if different systems use different internal architectures, they can export comparable

traces. This would immediately improve reproducibility and allow meta-analysis across papers and repositories. It would also make negative results easier to publish: a failed run could be described as a trace bundle rather than an anecdote.

In months six to twelve, the priority should be controlled longitudinal evaluation. The field should build benchmark suites that run over multiple sessions, include stale and contradictory information, and require the agent to preserve earlier capabilities while learning new ones. This period should also produce standard regression suites for popular agent tasks: coding, web navigation, research synthesis, customer support, data analysis, and workflow automation. The key metric should be not only whether the agent improves on the latest task, but whether the accumulated system remains stable. A useful target would be to report a self-evolution balance sheet: gains, regressions, extra cost, human interventions, safety events, and rollback frequency.

In months twelve to eighteen, the priority should shift to compositional experiments. Researchers should take systems that already work in one loop and combine them under controlled conditions. For example, a Reflexion-style memory loop can be combined with an EvoMAC-style workflow loop while holding the base model fixed. A DGM-style code archive can be evaluated with and without typed memory. A self-play curriculum can be paired with an external hidden evaluator to measure whether generated tasks improve transfer or merely exploit the proposer-solver dynamic. These experiments should not aim for the largest possible integrated agent; they should aim for causal clarity about loop interactions.

In months eighteen to twenty-four, the field should build production pilots with conservative autonomy. The best pilots will not be the most spectacular demos. They will be systems where the evolving object is narrow, the evaluator is strong, the rollback path is tested, and the user value is measurable. Examples include a coding assistant that evolves repository-specific repair heuristics but cannot merge without review; a research assistant that evolves source-ranking policies but preserves citation provenance; a support agent that evolves response templates under compliance checks; or a data-analysis agent that evolves reusable cleaning scripts under unit tests. Such pilots would reveal the real maintenance burden of self-evolution: how often memory must be curated, how often evaluators fail, how much human review is saved, and which updates users actually trust.

By the end of the third year, the field should be able to distinguish at least three maturity levels. A *laboratory self-evolving system* improves on controlled benchmarks and publishes traces, costs, and ablations. A *managed self-evolving system* can run in a sandbox or internal workflow with typed memory, gated updates, rollback, and longitudinal monitoring. A *production self-evolving system* can safely adapt under real users with defined risk tiers, audit logs, incident response, and evidence that improvements transfer beyond the data used to generate them. This maturity model would help reduce hype. A method can be valuable at the laboratory level without being ready for production, and a production system can be useful even if it evolves only a small part of itself.

The roadmap should also include a data model for field knowledge. Every paper, repository, benchmark, and production report should be indexed by the same core questions: what is the evolving object, what signal drives change, what update operator is used, what evaluator approves the change, what safety boundary is enforced, what cost is incurred, what evidence shows transfer, and what failure modes were observed? The local analysis already suggests useful fields such as repository category, technology stack, benchmark type, memory mechanism, evaluation scope, safety controls, and production pain points. Turning these fields into a shared evidence graph would make the survey itself evolvable. As new methods appear, they could be compared by mechanism rather than by marketing label.

The Evolve-AGI Index is a first instantiation of this data model. It compresses heterogeneous evidence into seven auditable signals: benchmark performance, core loop strength, evidence-chain credibility, transfer and verification, implementation access, field momentum, and governance readi-

ness. The purpose is not to rank agents by speculative AGI proximity. It is to track whether the field is becoming more capable of producing controlled, verifiable, reusable, and governable self-improvement loops. In future versions, each index snapshot should be append-only, cite its source corpus, report benchmark inputs separately from social or adoption signals, and preserve enough lineage for reviewers to reproduce why the score changed.

A final roadmap item is education. Many failures in agent systems come from treating LLM behavior as magic rather than as part of an engineered system. Developers need patterns for prompt versioning, memory provenance, evaluator isolation, tool sandboxing, and cost budgets. Researchers need templates for reporting self-evolution experiments. Product teams need language for explaining controlled adaptation to users. Safety teams need threat models for delayed memory poisoning, evaluator tampering, and autonomous code changes. If these practices become common, the baseline quality of the ecosystem will rise, and algorithmic advances will have a safer substrate on which to operate.

One practical litmus test can guide all of these milestones: after an improvement cycle, an external reviewer should be able to answer what changed, why it changed, what evidence supported the change, what it cost, what risks were considered, and how to undo it. If any answer is missing, the system may still be an interesting experiment, but it is not yet a trustworthy self-evolving system. This litmus test is intentionally simple because the field needs habits that can survive outside research labs. The same questions should appear in papers, framework dashboards, benchmark submissions, and production incident reviews.

9 Conclusion

This survey has framed AI self-evolution as the convergence of five loops: feedback, memory, reward, architecture, and population or code evolution. Each loop has a distinct history. Feedback loops come from self-refinement, critique, and iterative reasoning. Memory loops come from agents that store reflections, skills, user preferences, and environmental knowledge. Reward loops come from reinforcement learning, self-play, preference optimization, and self-rewarding models. Architecture loops come from agent workflow search, symbolic learning, textual backpropagation, and automated design of agentic systems. Population and code-evolution loops come from evolutionary computation, program synthesis, open-ended search, and self-modifying agents. The Evolve-AGI Index adds a measurement layer over these loops, asking whether the surrounding evidence is benchmark-visible, loop-complete, transferable, runnable, active, and governable. The intersection of these loops is what makes contemporary self-evolution different from ordinary adaptation. A system becomes self-evolving when it can use experience to change the mechanism by which it acts, learns, evaluates, or changes again.

The literature from 2022 to 2026 shows rapid progress toward this vision. Early self-refinement methods demonstrated that models can critique and revise outputs. Verbal reinforcement and memory-based agents showed that experience can improve future attempts without weight updates. Self-play and reward-based approaches showed that models can generate or evaluate their own training signals under structured constraints. Agent design and symbolic learning methods showed that prompts, roles, and workflows can be optimized as first-class artifacts. DGM, Godel Agent, AlphaEvolve, and related systems pushed the frontier toward code-level and open-ended self-modification. Together, these systems make it plausible that future AI agents will not be hand-designed once and deployed unchanged. They will be built to adapt.

But the survey also shows that adaptation is not automatically progress. Benchmark gains can be Goodharted. Memory can drift. Self-modification can break safety boundaries. Multi-agent

workflows can become opaque. Search can become expensive. Production users can be left with systems that are impressive in demos and unreliable in daily work. The most important conclusion is therefore not that self-evolution is solved, but that it must be engineered as a controlled process. The field needs evaluators that resist optimization pressure, memory systems that preserve provenance, safety gates that separate candidate changes from deployment, composable loop interfaces that permit ablation, and production architectures that make traces, costs, and rollbacks visible.

The phrase "self-evolution" should not be reserved only for the most radical systems that rewrite their own code. Nor should it be diluted to mean any retry loop. The useful definition is structural: self-evolution is the ability of an AI system to improve its future behavior by updating some persistent component of itself or its operating process, using feedback from interaction with tasks, environments, users, or evaluators. Under this definition, prompt refinement, memory consolidation, reward learning, workflow search, code mutation, and archive-based exploration are all partial forms. The frontier lies in composing them safely.

Bridging research to practice will require humility. Many real users do not primarily ask for agents that are more autonomous; they ask for agents that are more reliable, transparent, affordable, and controllable. Self-evolution is valuable only if it serves those goals. A system that learns when to ask for help may be more useful than one that blindly insists on autonomy. A system that remembers less but remembers correctly may be better than one with unlimited ungoverned memory. A system that proposes patches for review may be safer than one that directly mutates production code. The practical path to self-evolution is therefore incremental, observable, and reversible.

The next generation of self-evolving systems should be judged by four questions. First, what exactly evolves: output, prompt, memory, workflow, tool policy, code, weights, or population? Second, what evidence justifies the update: executable tests, hidden validation, human feedback, reward models, longitudinal performance, or cross-domain transfer? Third, what prevents the update from causing harm: sandboxing, privilege separation, approval gates, rollback, invariants, or monitoring? Fourth, what remains stable across evolution: user intent, safety policy, auditability, privacy, and operational budgets? If a system can answer these questions clearly, it belongs to the emerging discipline of controlled self-evolution. If it cannot, its improvements should be treated as experimental artifacts rather than deployable autonomy.

In the long run, AI self-evolution may become a standard property of intelligent systems. Models and agents will update their memories, refine their tools, search their workflows, learn from rewards, and maintain archives of alternative strategies. The scientific challenge is to understand these processes; the engineering challenge is to make them reliable; the alignment challenge is to keep them controllable; and the product challenge is to deliver real value rather than benchmark theater. The central thesis of this survey is that these challenges cannot be solved separately. The intersection of the five loops is self-evolution, and the intersection of evaluation, memory, safety, composability, and production engineering is what will make self-evolution useful.

A Open-Source Project Evidence

This appendix records the evidence tables used by the survey. The tables are intentionally compact: they give readers a reproducible map of the local research corpus without turning the arXiv-facing manuscript into a directory dump. The full project model cards, raw captures, and public site reports remain in the repository indexes and website outputs.

A.1 Core self-evolution systems

Project	Stars	Language	Recent activity	Mode	Main contribution
OpenEvolve	6,358	Python	2026-03	search + evaluation	Open implementation pattern for FunSearch-style evolutionary code optimization.
Reflexion	3,158	Python	2025-01	reflection + memory	Verbal reinforcement through episodic reflections, including strong HumanEval and ALFWorld evidence.
Agent Symbolic Learning	5,928	Python	2024-09	symbolic update	Textual backpropagation over prompts, tools, and agent pipelines.
AgentEvolver	1,441	Python	2026-04	search + orchestration	Evolutionary optimization of agent tool use and task workflows.
Self-Refine	805	Python	2024-10	feedback refinement	Generate–feedback–revise loop across multiple text and code tasks.
SE-Agent	274	Python	2025-09	revision + recombination	Multi-trajectory revision and recombination for software-engineering agents.
Science-CodeEvolve	97	Python	2026-04	search + evaluation	Evolutionary optimization for scientific code and algorithmic routines.
SCOPE	77	Python	2026-03	search	Evolutionary search framework for agent improvement experiments.
LLM-Self-Judge	43	Python	2026-03	self-judging	Self-judgment loop for data/model evolution and evaluation.
DARWIN	41	Python	2026-05	safety-aware evolution	Safety-oriented self-evolving agent prototype.

A.2 Agent infrastructure baselines

Project	Stars	Language	Recent activity	License	Stars per contributor	Quality score
AutoGPT	175,000	Python	2026-05	MIT	213	28/100
MetaGPT	50,000	Python	2026-04	MIT	143	56/100
AutoGen	50,000	Python	2026-05	MIT	125	66/100
OpenHands	55,000	Python	2026-05	MIT	145	68/100
CrewAI	30,000	Python	2026-05	MIT	107	62/100
LangGraph	20,000	Python	2026-05	MIT	89	67/100
DSPy	25,000	Python	2026-05	MIT	62	73/100
SWE-Agent	15,000	Python	2026-05	MIT	75	70/100

A.3 Landing-page and evidence-group taxonomy

- **Evolutionary code and AlphaEvolve-like systems**: OpenEvolve, CodeEvolve, Science-CodeEvolve, and SE-Agent.

- **Agent self-evolution systems**: AgentEvolver, Agent Symbolic Learning, SCOPE, and related agent-optimization frameworks.
- **Reflection and refinement classics**: Reflexion and Self-Refine.
- **Safety, judging, and data/model self-evolution**: DARWIN, LLM-Self-Judge, Misevolution, OEP, and capability-erosion studies.
- **Agent frameworks and infrastructure**: AutoGPT, MetaGPT, AutoGen, CrewAI, DSPy, LangGraph, OpenHands, and SWE-Agent.

B Paper Corpus Index

The survey corpus tracks 108 detailed paper references across 2022–2026. Each paper is treated as method evidence: the project indexes extract the evolving object, feedback signal, update operator, evaluator, benchmark, limitation, and reproducibility status when available.

Category	Count	Representative systems and claims
Reward-based evolution	14	STaR, Self-Rewarding LM, Meta-Rewarding, IterAlign, RISE, Agent-R, RAGEN, SPIN, and preference/self-play training lines.
Self-play evolution	6	Absolute Zero, SPIRAL, multi-agent debate, adversarial curriculum generation, and game-like proposer–solver loops.
Prompt and context evolution	16	Self-Refine, Reflexion, ExpeL, SICA, ACE, Agent Symbolic Learning, EvoMAC, EvolveR, and automated prompt optimization.
Architecture search and code self-modification	12	ADAS, Gödel Agent, Darwin Gödel Machine, AlphaEvolve, WebEvolver, FunSearch extensions, and self-modifying coding agents.
Memory-based evolution	10	Voyager, Memory-R1, ReasoningBank, AriadneMem, self-evolving distributed memory, and memory governance studies.
Hybrid methods and surveys	12	Lifelong learning roadmaps, open-ended evolution surveys, agent optimization surveys, and safety analyses.
Late-2025 to 2026 additions	19	SAGE, Hyperagents, Autogenesis, GenericAgent, Capability Erosion, OEP, Misevolution, and recent memory/reward/safety papers.
Other reviewed references	19	Benchmarks, tool-use systems, production agent frameworks, and background work used to connect the method families.

C Benchmark Evidence

Benchmark numbers in this appendix should be read as survey evidence rather than leaderboard claims. The main text argues that every self-evolution result needs a process report: initial artifact, update budget, evaluator, held-out test, cost, regressions, and safety boundary.

C.1 Code generation and software engineering

Method	Benchmark	Baseline	Reported result	Interpretation
Reflexion	HumanEval pass@1	GPT-4 zero-shot: 80.1%	91.0% with GPT-3.5 Reflexion	Strong reflection result, but evaluator scope is narrow.

Method	Benchmark	Baseline	Reported result	Interpretation
SelfEvolve	DS-1000 pass@1	ChatGPT: 49.4	57.2	Execution feedback improves API-heavy data-science code.
SelfEvolve	HumanEval pass@1	ChatGPT: 74.39%	85.98%	Iterative debugging improves small code tasks.
Darwin Gödel Machine	SWE-bench Verified	20.0%	50.0%	Archive-based self-modifying coding agents.
Darwin Gödel Machine	Polyglot	14.2%	30.7%	Evidence for transfer beyond a single benchmark.
SICA	SWE-bench Verified	17.0%	53.0%	Self-modifying coding agent with large reported gain.
ReVeal	LiveCodeBench pass@1	36.9 at turn 1	42.4 at turn 19	Verifier-aware multi-turn inference; budget must be reported.

C.2 Reasoning, agents, and open-ended discovery

Method	Domain	Baseline	Reported result	Metric
Self-Refine	Math reasoning	56%	67%	Accuracy
Agent Symbolic Learning	MATH	Agent baseline: 56.0%; DSPy: 48.4%	60.7%	Accuracy
RISE	GSM8K, Llama2-7B	about 14%	about 24%	Accuracy
RISE	GSM8K, Mistral-7B	about 38%	about 46%	Accuracy
Agent Symbolic Learning	HotPotQA	F1 36.2, EM 22.8	F1 39.1, EM 25.6	F1 / EM
Reflexion	ALFWorld	75% base agent	97%	Success rate
Voyager	Minecraft	Prior Minecraft agents	3.3x unique items; 2.3x travel distance	Exploration
AlphaEvolve	4x4 complex matrix multiplication	Known construction	48 multiplications	Algebraic verification
AlphaEvolve	Borg scheduling	Production baseline	0.7% worldwide compute recovery	Operational metric
AlphaEvolve	LLM training kernels	Existing kernels	23% and 32% speedups in reported settings	Runtime speed

C.3 Benchmark disclosure template

For arXiv-facing reproducibility, future revisions should attach this template to every benchmark claim:

1. **Evolving object**: output, prompt, memory, workflow, tool policy, code, weights, population, or data curriculum.
2. **Evaluator**: unit tests, hidden tests, reward model, human judge, formal verifier, execution trace, or production metric.
3. **Search budget**: turns, samples, candidates, tool calls, wall-clock time, tokens, dollars, and human review minutes.

Table 37: Star-quality dimensions used for repository triage.

Dimension	Weight	Rationale
Star activity	20%	Measures whether recent stargazers appear active rather than one-time viral accounts.
Contributor diversity	20%	Penalizes projects dominated by one maintainer when the claimed ecosystem value is broad.
Fork quality	20%	Counts forks with real commits or downstream work, not only passive copies.
Issue quality	20%	Rewards concrete bug reports, usage questions, and design discussion over low-information issues.
Pull-request merge rate	20%	Approximates maintainability, review throughput, and community integration.

4. **Split discipline**: train, validation, hidden test, time split, new task family, and contamination-control statement.
5. **Regression suite**: prior tasks, safety constraints, cost limits, permission boundaries, and negative-transfer cases.
6. **Audit and rollback**: versioned artifacts, accepted/rejected candidate logs, evaluator version, and rollback path.

D GitHub Star Quality and Hype Detection

GitHub stars are adoption signals, not quality labels. The project analysis therefore uses a multi-factor score to prevent viral attention from overwhelming engineering evidence. The score is not a universal truth; it is a triage aid for deciding which repositories deserve deeper model-card analysis.

D.1 Scoring dimensions

D.2 Representative quality ranking

Project	Stars	Score	Growth pattern	Interpretation
OpenEvolve	6,358	74/100	organic	Smaller but high-signal project for evolutionary code optimization.
DSPy	25,000	73/100	organic	Strong academic and engineering footprint.
SWE-Agent	15,000	70/100	organic	Research-backed software-engineering agent.
OpenHands	55,000	68/100	organic	Large community with active development.
LangGraph	20,000	67/100	steady	Strong workflow-graph infrastructure.
AutoGen	50,000	66/100	organic	Large framework ecosystem with institutional support.
FunSearch	1,500	62/100	organic	Under-starred relative to its scientific contribution.
CrewAI	30,000	62/100	steady	Practical orchestration framework.
Reflexion	3,158	58/100	steady	Classic research implementation.
MetaGPT	50,000	56/100	viral	Broad recognition, but adoption quality should be inspected.

Project	Stars	Score	Growth pattern	Interpretation
AutoGPT	175,000	28/100	viral	Historically important but star count overstates current quality.
Devika	22,000	30/100	suspicious	Me-too growth pattern with weak maintenance signal.

E Research Groups and Intellectual Lineage

Self-evolution work connects several communities: evolutionary computation, AutoML/NAS, program synthesis, reinforcement learning, language-agent engineering, tool-use benchmarks, and AI safety. The lineage map is useful because it prevents the field from being read as only a 2024–2026 LLM trend.

- **Evolutionary computation lineage**: Schmidhuber's Gödel machine, Clune's open-ended evolution and AI-GA framing, quality-diversity methods, MAP-Elites, and modern LLM-guided program evolution.
- **Self-improvement and reasoning lineage**: STaR, Self-Refine, Reflexion, RISE, RAGEN, Absolute Zero, and related training-time or inference-time improvement loops.
- **Agent architecture lineage**: ADAS, Agent Symbolic Learning, EvoMAC, Gödel Agent, DGM, SICA, and workflow/agent-topology search.
- **Code and algorithm discovery lineage**: FunSearch, AlphaEvolve, OpenEvolve, ScienceCodeEvolve, and software-engineering agents evaluated on SWE-bench-like tasks.
- **Memory and lifelong-agent lineage**: Voyager, ExpeL, ReasoningBank, Memory-R1, AriadneMem, Mem0, LangMem, Graphiti, and benchmark work on long-context or stateful agents.
- **Safety and governance lineage**: evaluator tampering, reward hacking, memory poisoning, capability erosion, OEP, Misevolution, and production rollback/audit patterns.

F Method Lookup Table

Method	Loop type	Train?	Feedback source	Best-fit tasks
Self-Refine	reflection	no	self-feedback	generation, code, dialogue
Reflexion	reflection + memory	no	environment feedback + verbal memory	decision tasks, code, ALFWorld-like tasks
Self-Debug	evaluator loop	no	execution feedback	code generation
STaR	rationale bootstrapping	yes	answer labels and rationale filtering	reasoning
SPIN	self-play	yes	self-play preference signal	general language tasks
ReST-EM	filtering + training	yes	self-generated candidates and filters	reasoning and code
Constitutional AI	critique + preference	yes	AI feedback under a constitution	alignment and harmlessness
OPRO	optimization	no	scoring function	prompt and program optimization

Method	Loop type	Train?	Feedback source	Best-fit tasks
FunSearch	search + evaluation	no	program evaluator	mathematical discovery and heuristic search
AlphaEvolve	population + code evolution	no	programmatic and production evaluators	algorithm discovery and infrastructure optimization
ADAS	architecture search	no	task benchmarks	agent design
DGM	self-reference + population	no	isolated coding benchmarks	agent self-improvement
Gödel Agent	runtime self-modification	no	task validation	agent self-modification experiments
Agent Symbolic Learning	symbolic search + reflection	no	language losses and gradients	multi-task agents
Absolute Zero	self-play + RL	yes	self-generated verified tasks	reasoning and code
RAGEN	trajectory RL	yes	multi-turn reward	interactive agents
SICA	code self-modification	no	SWE-bench-like validation	software engineering
ACE	context evolution	no	contextual feedback	agent workflows
EvolveR	hybrid evolution	yes	offline distillation + online policy evolution	strategy improvement
Memory-R1	memory + RL	yes	memory-management reward	long-horizon memory tasks

G Release and Bilingual Versioning Notes

The English manuscript built from `paper-drafts/main.tex` is the arXiv-facing version. It should remain English-only in compiled output, avoid local system-font dependencies, and include all tables needed to support the survey claims. The Chinese companion material is maintained separately in the Chinese survey tree and in the Chinese appendix source so that bilingual readers can inspect the same evidence without mixing languages inside the arXiv PDF.

Before a public submission, the release bundle should pass the following gates:

1. Compile the English manuscript from a clean checkout with `xelatex`, `bibtex`, and two final `xelatex` passes.
2. Confirm that the English source set used by `main.tex` contains no Chinese characters.
3. Confirm that every benchmark number in Chapters 5 and Appendix C has a paper review, raw paper, or official report source.
4. Remove duplicate BibTeX aliases once citation keys are normalized.
5. Package only arXiv-needed sources: `main.tex`, English chapters, `appendix-en.tex`, bibliography files, and any figure assets actually referenced by the manuscript.
6. Keep Chinese companion outputs linked from the repository and website, but do not include them in the arXiv source tarball unless submitting a separate Chinese note.

References

[1] AIWaves Inc. and Zhejiang University. Agent symbolic learning enables self-evolving agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[2] AlgorithmicSuperintelligence. Openevolve: Open-source evolutionary code optimization. *GitHub Repository*, 2024.

[3] Jo ao Moura. CrewAI: Framework for orchestrating role-playing autonomous AI agents, 2024.

[4] Jacob Austin et al. Program synthesis with large language models, 2021.

[5] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint*, 2022.

[6] Anna C. K. Beeken et al. LLaMEA: A large language model evolutionary algorithm for optimizing model architectures. *arXiv preprint*, 2024.

[7] Q9 Charles. Self-evolving agents: A survey. GitHub repository, 2025.

[8] Harrison Chase. Langchain: Building applications with llms through composability, 2023.

[9] Mark Chen et al. Evaluating large language models trained on code, 2021.

[10] Xinyun Chen et al. Teaching large language models to self-debug, 2023.

[11] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. SPIN: Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint*, 2024.

[12] Paul F. Christiano et al. Deep reinforcement learning from human preferences, 2017.

[13] Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence, 2019.

[14] Karl Cobbe et al. Training verifiers to solve math word problems, 2021.

[15] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20:1–21, 2019.

[16] Rob Fitzpatrick. The mom test: How to talk to customers and learn if your business is a good idea when everyone is lying to you, 2013.

[17] Google DeepMind. Ai solves imo problems at silver-medal level, 2024.

[18] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212, 2021.

[19] Dan Hendrycks et al. Measuring mathematical problem solving with the math dataset, 2021.

[20] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, Jörg Schlötterer, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint*, 2023.

[21] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.

[22] Jie Huang et al. Self-improvement in large language models: A survey. arXiv preprint, 2025.

[23] Naman Jain et al. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.

[24] Shuyang Jiang et al. CodeEvolve: Evolutionary code optimization with large language models. *arXiv preprint*, 2024.

[25] Shuyang Jiang, Yuhao Wang, and Yu Wang. Selfevolve: A code evolution framework via large language models. *arXiv preprint*, 2023.

[26] Carlos E. Jimenez et al. Swe-bench: Can language models resolve real-world github issues?, 2023.

[27] Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. In *International Conference on Learning Representations (ICLR)*, 2026.

[28] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Tianyi Zhang, Keshav Santhanam, Sri Vardhamanan, Saumya Jain, Matei Zaharia, Christopher Potts, Matei Zaharia, et al. Demystifying chains, trees, and graphs of reasoning. In *Conference on Language Modeling (COLM)*, 2024.

[29] Y. Lai et al. Ds-1000: A natural and reliable benchmark for data science code generation, 2023.

[30] LangChain, Inc. LangGraph: Framework for building stateful, multi-actor applications with LLMs. *Software*, 2024.

[31] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone, 2011.

[32] Guohao Li et al. Camel: Communicative agents for mind exploration of large language model society, 2023.

[33] Fei Liu et al. Large language models for evolutionary optimization: A survey. *arXiv preprint*, 2024.

[34] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search, 2018.

[35] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakub Foerster, Jeff Clune, Yujing Hu, Shengran Hu, et al. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint*, 2024.

[36] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[37] Grzegorz Malewicz et al. Pregel: A system for large-scale graph processing, 2010.

[38] David Moher, Alessandro Liberati, Jennifer Tetzlaff, and Douglas G. Altman. Preferred reporting items for systematic reviews and meta-analyses: The prisma statement. *PLoS Medicine*, 6(7), 2009.

[39] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015.

[40] Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Filip Wolski, Marelo Balog, et al. Alphaevolve: A gemini-powered coding agent for algorithm discovery. *Google DeepMind Technical Report*, 2025.

[41] OpenHands Team. Openhands: An open platform for AI software developers as generalist agents. *arXiv preprint*, 2024.

[42] Hieu Pham et al. Efficient neural architecture search via parameter sharing, 2018.

[43] Y. Qu et al. Rise: Recursive self-improvement with reinforcement learning, 2024.

[44] Yuxiao Qu and Jiecao Zhang. Rise: Recursive introspection for self-improvement. In *International Conference on Machine Learning (ICML)*, 2024.

[45] Esteban Real et al. Regularized evolution for image classifier architecture search, 2019.

[46] Pengzhen Ren, Yun Xiao, et al. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys*, 2021.

[47] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2024.

[48] Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. *arXiv preprint cs.LG/0309048*, 2003.

[49] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[50] Mohit Shridhar et al. Alfworld: Aligning text and embodied environments for interactive learning, 2020.

[51] Significant Gravitas. AutoGPT: An autonomous GPT-4 experiment, 2023.

[52] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies, 2002.

[53] Trieu Trinh et al. Solving olympiad geometry without human demonstrations, 2024.

[54] A. M. Turing. Computing machinery and intelligence. *Mind*, 1950.

[55] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Cong Lu, Yelin Zhu, Linxi Fan, and Yuke Zhu. Voyager: An open-ended embodied agent with large language models. *arXiv preprint*, 2023.

[56] Lei Wang et al. A survey on large language model based agents, 2024.

[57] Lei Wang et al. A survey on self-evolving agents, 2025.

[58] Rui Wang et al. Paired open-ended trailblazer, 2019.

[59] X. Wang et al. Ragen: Training agents by reinforcing reasoning, 2025.

[60] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint*, 2023.

[61] Shangyun Wu, Fei Liu, et al. Evoprompting: Language models for code-level neural architecture search. *arXiv preprint*, 2023.

[62] Xingyu Wu et al. Llm4ec: A survey of large language models for evolutionary computation, 2025.

[63] Yue Wu et al. Ragen: Self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint*, 2025.

[64] Zhiheng Xi et al. Riseagent: Self-evolving agent framework. arXiv preprint, 2025.

[65] Chengrun Yang et al. Evolutionary computation in the era of large language models. *arXiv preprint*, 2024.

[66] Chengrun Yang, Xuezhi Wang, Yuxiang Wei, Jiahao Yu, and Denny Zhou. Large language models as optimizers. *arXiv preprint*, 2023.

[67] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint*, 2024.

[68] Zhilin Yang et al. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018.

[69] Quanming Yao, Mengshuo Wang, Havocos Chen, et al. Taking human out of learning applications: A survey on automated machine learning. In *arXiv preprint arXiv:1810.13306*, 2018.

[70] Shunyu Yao et al. React: Synergizing reasoning and acting in language models, 2023.

[71] Shunyu Yao et al. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[72] Xunjian Yin et al. Godel agent: A self-referential agent framework for recursive self-improvement, 2024.

[73] Dubbydu Young. Optimization methods for self-evolving agent systems. arXiv preprint, 2026.

[74] Weizhe Yuan, Richard Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint*, 2024.

[75] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[76] Jie Zhang, Shengran Hu, Cong Lu, Jeff Clune, Cong Tian, and Yuqi Xie. Darwin gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint*, 2025.

[77] Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Sam Wang, Qiran Wu, Zang Zheng, et al. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[78] Kunhao Zheng et al. minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2021.

[79] Shuyan Zhou et al. Webarena: A realistic web environment for building autonomous agents, 2023.

[80] Daniel M. Ziegler et al. Fine-tuning language models from human preferences, 2019.

[81] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2018.

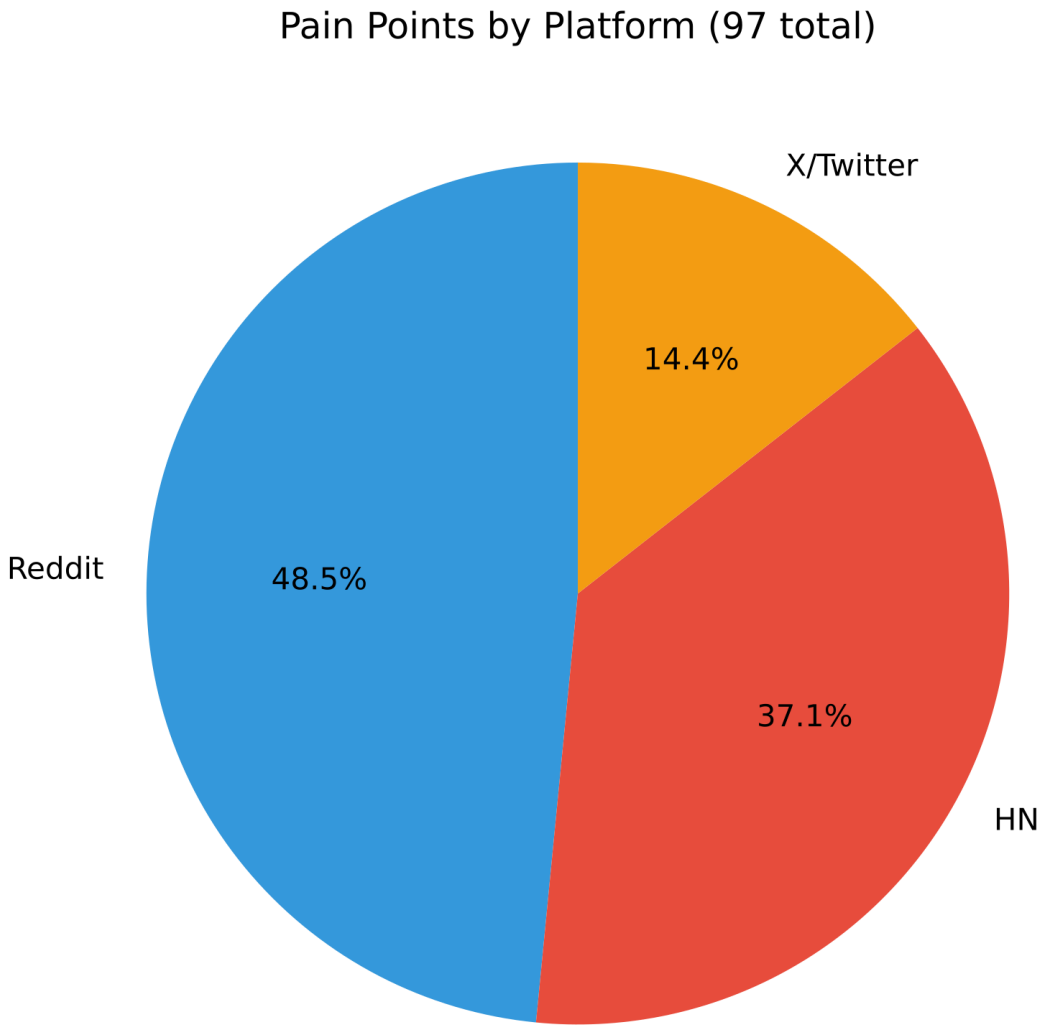

Figure 21: Pain-point evidence by public discussion platform. The Mom Test layer is used as a demand signal for reliability, observability, loop control, cost governance, and security requirements.