Review

The Path to Recursive Self-Improving Agents: Foundation, Framework, and Future Directions

Shuaiqi Liu †, Zhengkai Lin †, Yuxiang Zhang †, Yuanyi Ren †, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu *,†, Jieping Ye

Posted Date: 3 August 2026

doi: 10.20944/preprints202608.0051.v1

Keywords: AI agent; recursive self-improvement; self-improving agent; LLM; artificial intelligence

 doi:10.20944/preprints202608.0051.v1

Review

The Path to Recursive Self-Improving Agents: Foundation, Framework, and Future Directions

Shuaiqi Liu †, Zhengkai Lin †, Yuxiang Zhang †, Yuanyi Ren †, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu *,† and Jieping Ye

Alibaba Group, China
* Correspondence: zhihang.fzh@alibaba-inc.com
† Core contributors.

Abstract

Agent systems based on foundation models have advanced rapidly and can now accomplish increasingly complex tasks with greater autonomy. However, manually developing and refining agent systems requires substantial human effort and time, motivating a shift toward agent systems that can improve themselves. In this survey, we study self-improving agent systems that autonomously transform experience and evaluation feedback into persistent updates to their components. To compare the self-improvement capabilities of diverse agent systems, we introduce a five-level grading standard, ranging from manual improvement to general recursive self-improvement. Furthermore, we propose a unified research framework that jointly models the foundation model, agent harness, agent data system, agent trainer, and the mechanism governing their improvement as a coupled system. Guided by this framework, we systematically review the literature on self-improvement and provide a taxonomy covering self-improvement in individual components and co-improvement across components. In addition, this survey identifies open research problems on the path toward recursive self-improving agents. Overall, this survey provides a foundation for studying self-improving agent systems and a roadmap for future research.

Keywords: AI agent; recursive self-improvement; self-improving agent; LLM; artificial intelligence

1. Introduction

Agent systems built on foundation models have made substantial progress in recent years. They can now complete increasingly complex tasks with greater autonomy, and the frontier of their capabilities continues to expand. In particular, an empirical study quantifies this trend by measuring agent capability in terms of *task length*, defined as the time required for human experts to complete a given task. Within the field of software engineering, it shows that the length of tasks that frontier agents can solve at a 50% success rate has approximately doubled every seven months since 2019 [1,2]. This progress reflects coordinated advances in four aspects. First, frontier *foundation models* have been trained at scale to strengthen their intrinsic agentic capabilities such as long-horizon planning, code generation, tool use, and iterative reflection [3–5], leading to rapid progress on agent benchmarks [6–8]. Second, more capable *agent harnesses* now support planning, long-context management, tool interaction, code execution, file-system operations, and failure handling, helping models handle complex and long-horizon tasks [9–12]. Third, the *production of agent data*, including environments, tasks, and trajectories, has scaled up dramatically [5,13,14]. Agent data systems can provide a continuing supply of training and evaluation signals for improving models' agentic capabilities. Finally, advances in *training algorithms and training infrastructure* have supported large-scale training of agent models [15–19].

However, progress across these aspects typically relies heavily on manual design and implementation. Each cycle of diagnosis, redesign, and implementation demands substantial human effort and

 doi:10.20944/preprints202608.0051.v1

time. Consequently, the scope and speed of improvement are constrained by the available expertise and budget. These constraints motivate a shift from manually improved agent systems toward those capable of autonomous self-improvement.

In this survey, we define an agent system as *self-improving* if it can autonomously transform experience and evaluation feedback into persistent updates to its internal components or parameters, thereby enhancing its capabilities and behaviors [20–23]. To formalize this notion, we formulate an agent system as a tuple of five interdependent components: the foundation model $\mathcal{M}$, agent harness $\mathcal{H}$, agent data system $\mathcal{D}$, agent trainer $\mathcal{T}$, and improvement mechanism *Imp*, as shown in Figure 1.

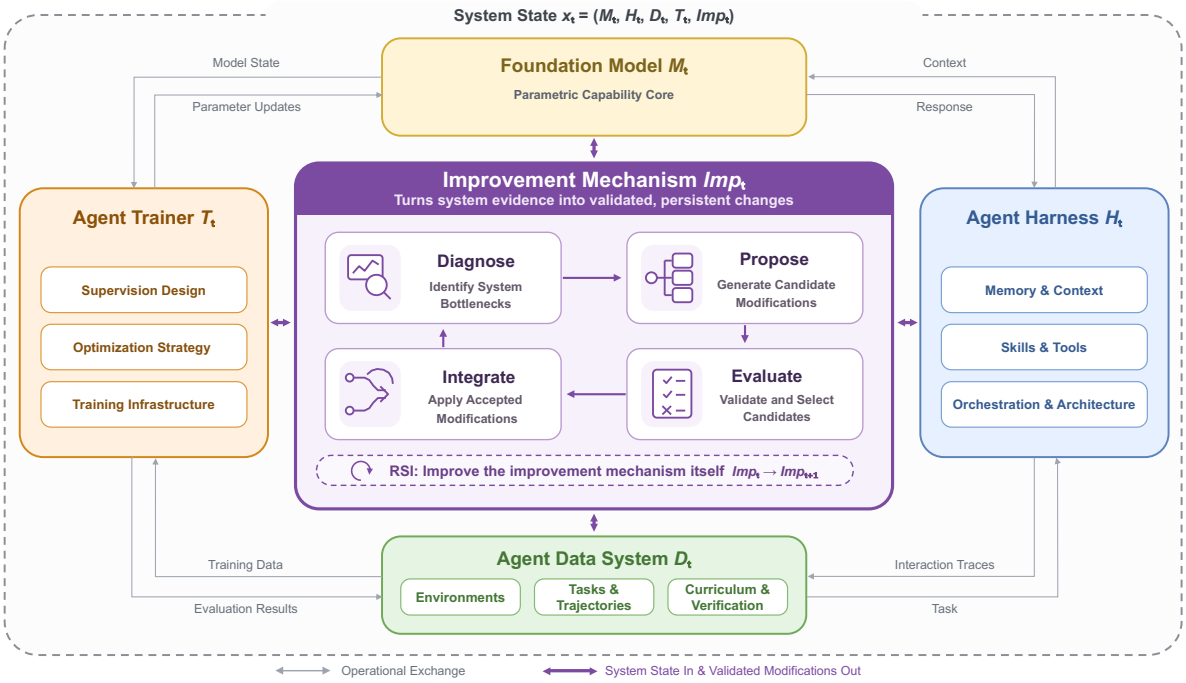

Figure 1. A unified framework for agent system self-improvement.

This formulation naturally suggests a further extension: since *Imp* is itself a component of the system, it can likewise be subject to recursive modification. In this scenario, each improvement cycle not only enhances operational capabilities but also optimizes the mechanism that drives future improvements [24,25], enabling the agent system to effectively **improve how to improve**. We refer to this capability as *recursive self-improvement* (RSI), raising the possibility of open-ended, self-reinforcing improvement.

To evaluate where existing agent systems stand on the path to RSI, we further introduce a grading standard that classifies them into five levels of self-improvement capability, ranging from manual improvement, where every change requires external human intervention, to RSI, where the improvement mechanism itself is subject to the system's own modification. This standard offers a shared basis for comparing these systems and indicating how far they have progressed toward more autonomous and general self-improvement.

Furthermore, we introduce a unified research framework that models the self-improving agent system, enabling a holistic analysis across components. The framework specifies two aspects: the scope of autonomous improvement and the procedure of improvement. As shown in Figure 1, the scope of autonomous improvement includes the foundation model, agent harness, agent data system, agent trainer, and improvement mechanism. The procedure of improvement includes the diagnosis of system bottlenecks, the proposal of candidate modifications to these components, the validation and selection of these candidates, and the integration of the selected modification. Based on this research framework, this survey provides a comprehensive review of self-improving agent systems.

In summary, this survey makes the following contributions:

 doi:10.20944/preprints202608.0051.v1

- We establish formal definitions of agent system self-improvement and RSI, together with a five-level capability grading standard ranging from manual improvement to general RSI.
- We propose a unified research framework (Figure 1) that models the foundation model, agent harness, agent data system, agent trainer, and improvement mechanism as a coupled evolving system, capturing the scope, dependencies, and dynamics of self-improvement.
- We systematically reviews existing work on self-improving agent systems and provides a taxonomy (Figure 2), covering self-improvement in individual components and co-improvement across components.
- We identify key open problems on the path to RSI, which provide research directions for developing more autonomous, reliable, and general self-improving agent systems.

Together, these contributions provide a unified foundation for studying self-improving agent systems and a roadmap toward RSI agent systems.

The remainder of this survey is organized as follows. Section 2 presents formal definitions of agent systems and self-improvement, establishes a grading standard for self-improvement capability, and introduces our unified research framework. Sections 3, 4, and 5 review the existing work on self-improvement in the agent harness, the agent data system, and the agent trainer, respectively. Section 6 summarizes existing work on collaborative improvement of multiple components within a self-improving agent system. Section 7 identifies open problems on the path toward general RSI agent systems. Finally, Section 8 concludes this survey.

 doi:10.20944/preprints202608.0051.v1

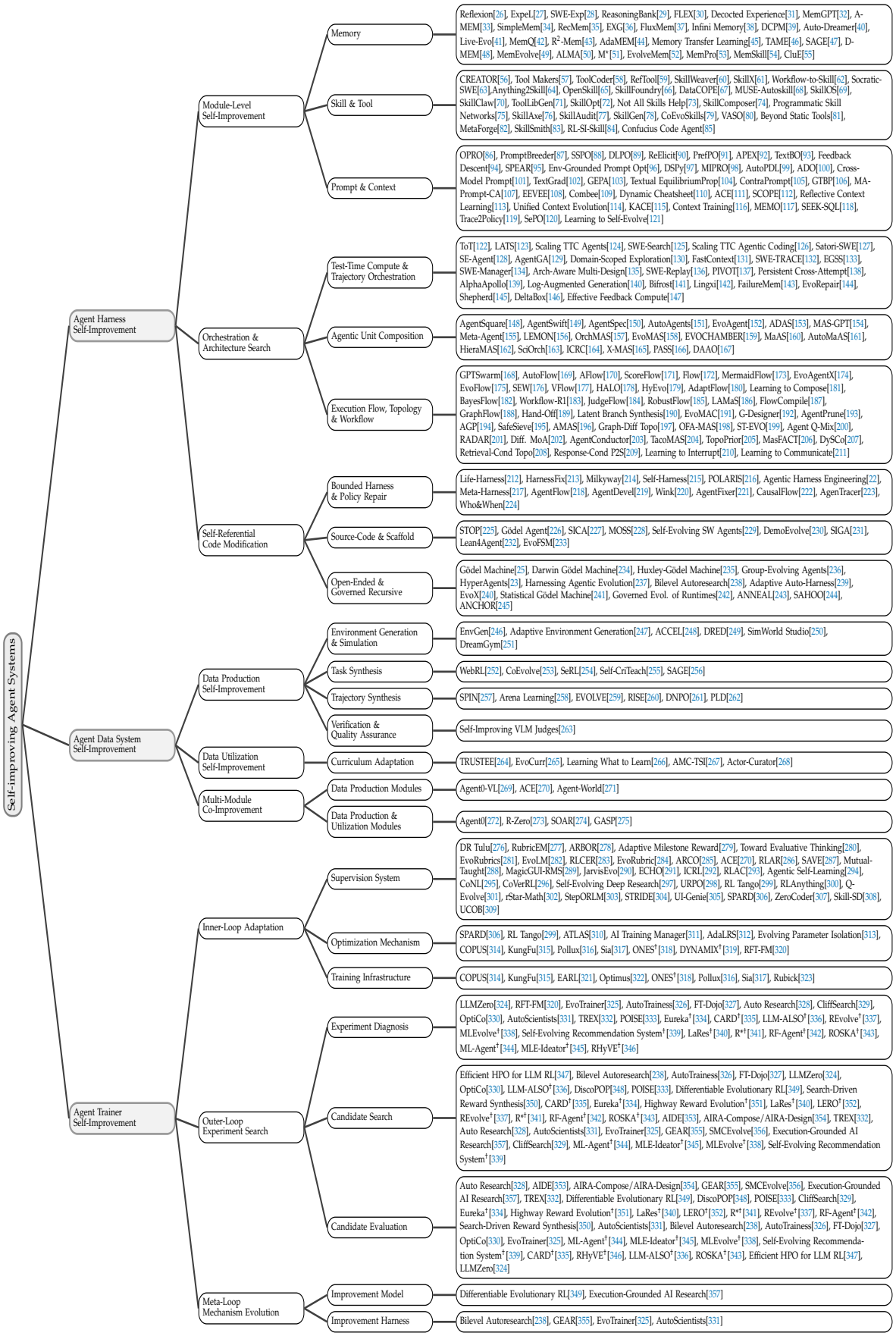

Figure 2. Taxonomy of self-improving agent systems organized into *Agent Data System Self-Improvement*, *Agent Harness Self-Improvement*, and *Agent Trainer Self-Improvement*, with representative works annotated at each leaf node.

 doi:10.20944/preprints202608.0051.v1

2. Foundation of Self-Improving Agent Systems

This section establishes the foundation for the rest of the survey by clarifying the key concepts of self-improving agent systems, defining a grading standard for self-improvement capability, and introducing a unified research framework. Together, these elements provide the conceptual basis, evaluation criteria, and analytical framework for examining existing work in later sections. Section 2.1 defines agent systems by introducing their main components and then formalizes self-improvement and recursive self-improvement (RSI). Section 2.2 introduces a five-level grading standard for comparing systems by autonomy in the improvement loop, modification of the improvement mechanism, and generalization across domains. Section 2.3 proposes a unified research framework for self-improving agent systems.

2.1. Preliminaries

To provide a clear basis for the subsequent discussion, this section establishes the conceptual and formal foundations of self-improving agent systems. Section 2.1.1 formalizes an agent system as a five-component tuple and explains the role of each component within the system. Section 2.1.2 formalizes the self-improvement process.

2.1.1. Agent Systems

An agent system is a computational system in which one or more autonomous agents observe an environment, reason over available information, and act to achieve specified objectives. Such a system typically contains several components that support planning, interaction, learning, and adaptation. In this survey, we represent an agent system as a tuple of five components: $(\mathcal{M}, \mathcal{H}, \mathcal{D}, \mathcal{T}, Imp)$, where $\mathcal{M}$ denotes the foundation model, $\mathcal{H}$ denotes the agent harness, $\mathcal{D}$ denotes the agent data system, $\mathcal{T}$ denotes the agent trainer, and Imp denotes the improvement mechanism. These five components serve distinct but complementary functions. We elaborate on each component in turn below.

Foundation model.

The foundation model $\mathcal{M}$ is the parametric core of the agent system. It provides general capabilities, including language understanding, reasoning, planning, code generation, and tool use [3,4]. These capabilities support the operation of the other components, while feedback from those components can in turn guide updates to $\mathcal{M}$.

Agent harness.

The agent harness $\mathcal{H}$ is the execution layer that places the foundation model in an operational loop. It governs what the model observes, what actions it can take, and how observations and actions are organized into task-completion workflows [9–12]. The harness therefore enables the system to adapt its behavior without necessarily changing the parameters of the foundation model.

Agent data system.

The agent data system $\mathcal{D}$ manages the lifecycle of training and evaluation data. It converts agent trajectories, environmental observations, performance feedback, and existing datasets into experience and evaluative signals [14,246,253]. It typically consists of two closely coupled stages: data production and data utilization. Together, these two stages connect the inference-time interaction in the harness with persistent learning in the trainer, and directly shape learning efficiency and system capability.

Agent trainer.

The agent trainer $\mathcal{T}$ converts agent experience into persistent model updates [282,325,358]. At each training step, $\mathcal{T}$ consumes data from the agent data system $\mathcal{D}$, and returns updated model parameters. It comprises three coupled modules. First, an evaluative feedback module transforms raw outcomes into learning signals, which may involve evaluation criteria, verifiers, or credit-assignment procedures. Second, a policy-optimization module specifies the objective, algorithm, update sched-

 doi:10.20944/preprints202608.0051.v1

ule, and constraints used to update model parameters. Third, a training-infrastructure module provides the computational substrate on which these updates run, such as computational backends and reinforcement-learning environments. Together, these modules determine how agent experience becomes a persistent parameter change in the agent system.

Improvement mechanism.

The improvement mechanism *Imp* governs how evidence and feedback are used to modify the agent system over time [234,324,325]. Its operational structure is specified in Section 2.3.1.

2.1.2. Self-Improvement

Self-improvement is an iterative process in which a system autonomously modifies its own state to improve its capabilities, behaviors, or underlying components [22,23]. Let $x_t \in \mathcal{X}$ denote the complete state of the agent system at discrete step t. This state includes the source code, parameters, configurations, and outputs of the agent system's components defined in Section 2.1.1. At each step, the improvement mechanism Imp_t in x_t reads the current state and produces the next state:

$$x_{t+1} = Imp_t(x_t). \tag{1}$$

The complete agent system participates in this iterative process, but not every component must change at every step. A given modification also need not immediately improve the component being modified. The improvement mechanism may greedily select the candidate with the largest predicted short-term gain. It may also maintain multiple evolutionary lineages, evaluate them in subsequent rounds, and retain candidates that support stronger long-term improvement.

Recursive Self-Improvement.

Recursive self-improvement (RSI) is a stronger form of self-improvement in which the system can also modify the mechanism that governs its future improvements [24,25]. In other words, an RSI agent system improves not only its current capabilities but also its ability to improve. In Equation 1, this means that Imp_t is not an external or fixed procedure. As part of x_t, Imp_t can itself be updated to a successor Imp_{t+1}, which then governs subsequent improvement cycles.

2.2. Self-Improvement Capability Grading Standard

Existing agent systems differ in how autonomous they are, whether the improvement mechanism can improve itself, and whether self-improvement can generalize to open domains. Consequently, the term *self-improving* has been applied to systems with substantially different properties. Inspired by the level-based structure of the SAE J3016 driving-automation taxonomy [**?**], we propose a five-level grading standard for more systematic comparison.

As shown in Table 1, the five levels of self-improvement capability can be distinguished by several aspects. The **proposer**, **modifier**, and **validator** describe whether the agent system can autonomously propose changes, apply them, and validate their effects without human intervention. The ***Imp* modification** dimension specifies whether the improvement mechanism *Imp* itself lies within the system's autonomous modification scope. This dimension is decisive for RSI, as it determines whether the system's improvement capacity itself can grow over time. Finally, **generality** indicates whether self-improvement transfers beyond a bounded task family or restricted operational setting, reflecting genuine capability gains rather than in-distribution overfitting.

 doi:10.20944/preprints202608.0051.v1

Table 1. Capability-grading standard for agent system self-improvement. A checkmark indicates that the agent system possesses the corresponding capability without per-update human intervention. *Imp* denotes the improvement mechanism. *Imp Modification* requires a functionally meaningful change that affects subsequent improvement cycles. *Generality* refers to the domain generality of recursive self-improvement, rather than task-performance generalization alone.

Level	Category	Programmatic Improvement Loop			Recursion	Domain
		Proposer	Modifier	Validator	*Imp* Modification	Generality
L1	Manual Improvement					
L2	Assisted Improvement	✓				
L3	Programmatic Self-Improvement	✓	✓	✓		
L4	Bounded Recursive Self-Improvement	✓	✓	✓	✓	
L5	General Recursive Self-Improvement	✓	✓	✓	✓	✓

The five levels are defined as follows:

(L1) **Manual improvement.** The agent system has no autonomous improvement capability. It is deployed and executed in a fixed form, and every change requires an external development and deployment process.

(L2) **Assisted improvement.** The agent system can propose candidate modifications or provide diagnostic evidence, but humans remain responsible for validating and applying substantive changes. The improvement mechanism *Imp* is maintained by humans rather than by the agent system. Therefore, L2 is regarded as a precursor to self-improvement rather than an instance of it.

(L3) **Programmatic self-improvement.** The agent system can autonomously propose, apply, and validate modifications to its operational components, such as the foundation model, agent harness, data system, or trainer. These modifications may be substantial and may span multiple components. However, the improvement mechanism *Imp* that governs these modifications remains fixed or externally maintained. L3 is therefore programmatic: the system controls the execution of improvement, but does not yet modify the mechanism *Imp* that governs subsequent improvement cycles.

(L4) **Bounded recursive self-improvement.** The agent system can not only propose candidate modifications, validate their effects, and apply the validated modifications, but also rewrite its own improvement mechanism, which makes the improvement mechanism self-referential. Individual steps need not modify every component, but the overall process supports sustained long-term progress within a bounded domain.

(L5) **General recursive self-improvement.** L5 retains the autonomy, self-reference, and long-term progress of L4. Beyond this, its improvement capability can transfer effectively across broad and evolving task domains rather than remaining limited to a fixed benchmark, narrow task family, or restricted operational setting. Such generality marks a qualitative shift: the system's capacity to improve is no longer tied to any particular problem space, opening the possibility of improvement across an ever-expanding range of domains.

In summary, this grading standard separates manual improvement, assisted improvement, programmatic self-improvement, bounded RSI, and general RSI. It provides a shared basis for comparing existing agent systems and for identifying how far current systems have progressed toward general RSI.

 doi:10.20944/preprints202608.0051.v1

2.3. Unified Research Framework for Agent System Self-Improvement

The preceding subsections define the agent-system components, self-improvement and RSI, and the grading standard for self-improvement capability. Based on these definitions, this subsection presents a unified research framework for analyzing self-improving agent systems as a coherent whole. The subsection first defines the scope of the research framework, then discusses the self-improvement process, and finally summarizes the main advantages of this framework.

2.3.1. Definition and Formulation

We define the unified research framework as a state-based model of the objects, dependencies, and update process involved in agent system self-improvement. It focuses on two key aspects: the scope of autonomous improvement and the mechanism of improvement. In other words, the framework specifies what can be improved and how improvement proceeds.

The scope of autonomous improvement specifies which parts of the agent system can be inspected, modified, and validated by the system itself. In this framework, the scope covers the states of all system components, while improvements to these states are constrained by their dependencies.

System state.

Building on the agent-system definition in Section 2.1.1 and Figure 1, we represent an agent system at iteration t as the following revisable state:

$$x_t = (\mathcal{M}_t, \mathcal{H}_t, \mathcal{D}_t, \mathcal{T}_t, Imp_t) \in \mathcal{X}, \tag{2}$$

where $\mathcal{M}_t$, $\mathcal{H}_t$, $\mathcal{D}_t$, and $\mathcal{T}_t$ denote the current foundation model, agent harness, agent data system, and agent trainer, respectively. Imp_t denotes the current improvement mechanism, and $\mathcal{X}$ contains the possible agent system states.

Operational dependencies.

The five components of x_t in Equation 2 do not operate independently. For this reason, improving a state in x_t requires considering how that change affects, and is limited by, the other components of the system. Figure 1 illustrates that the five components of x_t are mutually dependent at the system level. During task execution, the harness and the foundation model form the acting loop. The *harness* $\mathcal{H}_t$ assembles a context from sources such as the user instructions, memory, skills, tools, and observations. Given the context, the foundation model $\mathcal{M}_t$ then outputs an action decision. The harness executes the decision in the environment, collects observations, and stores the resulting interaction trace.

The *data system* $\mathcal{D}_t$ converts accumulated traces and other artifacts into training or evaluation data. Given the training data, the *trainer* $\mathcal{T}_t$ then converts data into persistent updates of the *model* $\mathcal{M}_t$. The evaluation data, in turn, provide feedback signals that guide these updates and inform improvement decisions across these components. The *improvement mechanism* Imp_t modifies the agent system as a whole by determining which of the five components should be changed and how, turning feedback into concrete updates across the system.

If these dependencies are ignored, a local improvement may leave the overall performance unchanged. This is likely when improving one component creates incompatibility with other components. It can also occur when the current bottleneck is distributed across several components rather than located in a single component.

Self-Improvement Process.

The improvement mechanism Imp_t realizes the transition from the current system state x_t to a revised state x_{t+1} through four functional stages, as summarized in Figure 1. Within the unified framework, these stages operate over the coupled system state and its component dependencies.

1. The Diagnose stage analyzes end-to-end performance, component states, interface behavior, and feedback signals to identify the bottlenecks that currently constrain system-level improvement.

 doi:10.20944/preprints202608.0051.v1

2. The Propose stage generates candidate modifications, each specifying the target subset of x_t and the proposed change. A candidate may target one component or coordinate changes across multiple interdependent components.
3. The Evaluate stage assesses the effects of candidate modifications on the coupled system, including performance, cost, compatibility, and safety, and selects candidates for integration.
4. The Integrate stage applies accepted modifications to produce x_{t+1} while preserving cross-component consistency and ensuring that the changes persist across subsequent improvement cycles.

Together, these four stages characterize the generic improvement dynamics of the unified framework. Different self-improving agent systems may implement each stage through different mechanisms.

2.3.2. Advantages of the Unified Framework

The unified framework has three main advantages.

First, it provides a common representation for research threads that are often studied separately. This makes it easier to compare methods that otherwise appear in separate discussions of harness design, data construction, training, or self-modification. Each existing paradigm can be characterized as a special case of the general update rule $x_{t+1} = Imp_t(x_t)$. A static agent system corresponds to the degenerate case where the system state remains unchanged, i.e., $x_{t+1} = x_t$. A fixed-mechanism agent system applies a rule-based $Imp_t(\cdot)$ that remains fixed and falls outside the autonomous modification scope. Finally, an RSI agent system represents the full realization of the framework, in which $Imp_t(\cdot)$ acts on the entire system state, including itself.

Second, covering all system components in this framework helps locate and eliminate system-level bottlenecks. End-to-end performance alone can show that a system has failed, but it often cannot explain why the failure occurs. Component-level evidence can identify local weaknesses, but it may miss failures caused by interface mismatch or by constraints that span multiple components [270,291]. By placing all components in one framework, the analysis can determine whether a proposed improvement changes the actual bottleneck or only improves an isolated part of the system.

Third, modeling the dependencies among system components helps clarify cross-component causality. This is essential for multi-component co-evolution, where a modification in one component often causes indirect, widespread effects across the system. Identifying the scope of impact along these dependency paths is necessary both for evaluating the effectiveness of the improvement and for achieving long-term, sustainable co-evolution. The framework helps distinguish a multi-component co-evolution system from a system that merely optimizes individual components in isolation.

3. Agent Harness Self-Improvement

As defined in Section 2, the agent harness $\mathcal{H}$ is the non-parametric execution layer that determines what the model can observe, what actions it can invoke, and how observations and actions are organized into goal-directed workflows. Harness self-improvement studies persistent changes to this layer, where interaction traces, evaluation feedback, and execution failures are converted into durable changes in external state, action interfaces, prompts, workflows, or executable scaffolds. The central idea is that an agent can improve by changing the conditions under which the foundation model is invoked, rather than by changing the model parameters themselves. This makes the harness a natural substrate for early agent self-improvement: it can accumulate experience, reshape tool use and coordination, and repair parts of the agent's operating procedure while the underlying model remains fixed. Its relevance to RSI depends on what becomes mutable. When only harness modules are updated by a mostly fixed procedure, the result is usually module-level self-improvement; when the scaffold or improvement procedure itself becomes a target of modification, harness evolution begins to approach recursive self-improvement. Figure 3 situates these cases within the scope of this section. We first examine persistent changes confined to individual harness modules, then changes to

 doi:10.20944/preprints202608.0051.v1

orchestration and architecture, and finally self-referential modifications to executable harness code and the mechanism governing future updates.

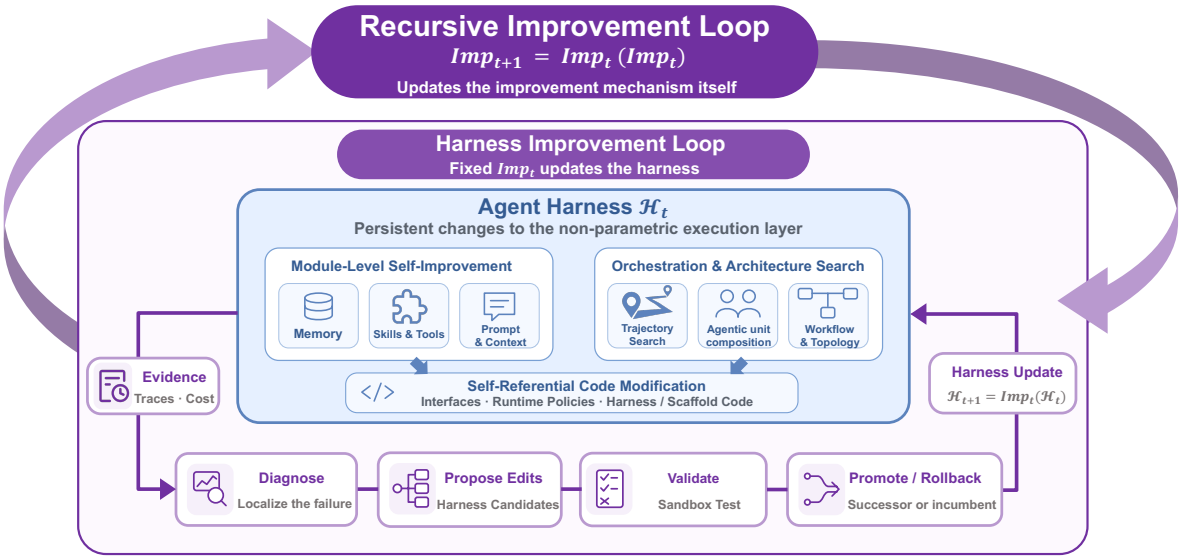

Figure 3. Conceptual framework of an operational agent harness, harness self-improvement, and improvement-mechanism evolution. The innermost layer depicts how an agent harness $\mathcal{H}_t$ organizes agentic units and their supporting modules through an orchestration structure. The harness-improvement loop uses execution evidence to diagnose problems, propose edits, validate candidates in a sandbox, and either promote a successor harness $\mathcal{H}_{t+1}$ or roll back. At the outer level, improvement histories and validation outcomes provide evidence for revising and meta-validating the improvement mechanism Imp_t itself, yielding Imp_{t+1} when a candidate is promoted.

Table 2. Representative self-improvement methods in agent harness evolution.

Work	Self-Improvement Process			Level
	Object	Evidence	Mechanism	
Module-Level Self-Improvement				
Reflexion [26]	Experience memory	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
SWE-Exp [28]	Experience memory	Trace + Outcome label	Experience reuse	L3
ReasoningBank [29]	Experience memory	Trace + Evaluator judgment	Experience reuse	L3
FLEX [30]	Experience memory	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
Decocted Experience [31]	Experience memory	Trace + Outcome label	Experience reuse	L3
Live-Evo [41]	Experience memory	Trace + Outcome label	Experience reuse	L3
Memory Transfer Learning [45]	Experience memory	Cross-domain feedback	Experience reuse	L3
Tool Makers [57]	Tool library	Execution result + Outcome label	Artifact synthesis	L3
ToolCoder [58]	Tool library	Execution result	Artifact synthesis	L3
SkillWeaver [60]	Skill library	Trace + Execution result + Evaluator judgment	Artifact synthesis	L3
SkillX [61]	Skill library	Trace + Outcome label + Evaluator judgment	Artifact synthesis	L3
Workflow-to-Skill [62]	Skill library	Trace	Artifact synthesis	L3
SkillFoundry [66]	Skill library	Execution result + Outcome label + Evaluator judgment	Artifact synthesis	L3
OpenSkill [65]	Skill library	Execution result + Outcome label + Evaluator judgment	Artifact synthesis	L3
SkillOS [69]	Skill library	Trace + Evaluator judgment	Policy learning	L3
SkillComposer [74]	Skill library	Trace + Outcome label	Policy learning	L3
Programmatic Skill Networks [75]	Skill library	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
SkillAxe [76]	Skill library	Evaluator judgment	Diagnostic repair	L3
SkillAudit [77]	Skill library	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
SkillGen [78]	Skill library	Trace + Outcome label + Evaluator judgment	Artifact synthesis	L3
CoEvoSkills [79]	Skill library	Execution result + Outcome label + Evaluator judgment	Artifact synthesis	L3
SkillSmith [83]	Skill-tool library	Trace + Execution result + Outcome label + Evaluator judgment	Search-based optimization	L3
Confucius Code Agent [85]	Skill-tool library	Trace + Execution result + Evaluator judgment	Artifact synthesis	L3
Dynamic Cheatsheet [110]	Context playbook	Evaluator judgment	Experience reuse	L3
ACE [111]	Context playbook	Trace + Execution result + Evaluator judgment	Experience reuse	L3
SCOPE [112]	Context playbook	Trace + Execution result + Evaluator judgment	Experience reuse	L3

 doi:10.20944/preprints202608.0051.v1

Work	Object	Evidence	Mechanism	Level
Reflective Context Learning [113]	Context playbook	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
Unified Context Evolution [114]	Context playbook	Trace + Outcome label + Improvement statistics	Experience reuse	L3
KACE [115]	Context playbook	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
MEMO [117]	Context playbook	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
SEEK-SQL [118]	Context playbook	Trace + Execution result + Outcome label + Evaluator judgment	Experience reuse	L3
GEPA [103]	Multi-stage prompts	Trace + Execution result + Outcome label	Search-based optimization	L3
Trace2Policy [119]	Decision-rule library	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
Learning to Self-Evolve [121]	Context-editor weights	Trace + Outcome label	Policy learning	L3
SePO [120]	Task prompt + Improver prompt	Outcome label	Search-based optimization + Meta-optimization	L4
Orchestration-Level Self-Improvement				
AgentGA [129]	Trajectory orchestration	Outcome label + Improvement statistics	Search-based optimization	L3
SWE-Replay [136]	Trajectory memory	Trace + Execution result + Outcome label	Experience reuse	L3
Log-Augmented Generation [140]	Trajectory memory	Trace	Experience reuse	L3
FailureMem [143]	Trajectory memory	Execution result + Outcome label + Evaluator judgment	Experience reuse	L3
EvoRepair [144]	Trajectory memory	Trace + Outcome label + Evaluator judgment	Experience reuse	L3
EvoAgent [152]	Agent composition	Evaluator judgment	Search-based optimization	L3
ADAS [153]	Agent system code	Execution result + Outcome label + Evaluator judgment	Search-based optimization	L3
EvoMAS [158]	Agent composition + Experience memory	Trace + Execution result + Evaluator judgment	Search-based optimization	L3
EVOCHAMBER [159]	Agent composition + Experience memory	Cross-domain feedback	Search-based optimization	L3
MermaidFlow [173]	Workflow structure	Execution result + Outcome label + Evaluator judgment	Search-based optimization	L3
EvoAgentX [174]	Workflow structure	Outcome label	Search-based optimization	L3
EvoFlow [175]	Workflow structure	Outcome label + Evaluator judgment	Search-based optimization	L3
SEW [176]	Workflow structure	Outcome label	Search-based optimization	L3
HyEvo [179]	Workflow structure	Execution result + Outcome label + Evaluator judgment	Search-based optimization	L3
AdaptFlow [180]	Workflow structure	Outcome label + Evaluator judgment	Search-based optimization	L3
JudgeFlow [184]	Workflow structure	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
Lean4Agent / LeanEvolve [232]	Workflow structure	Trace + Execution result + Outcome label + Evaluator judgment	Diagnostic repair	L3
EvoFSM [233]	Workflow structure	Trace + Evaluator judgment	Diagnostic repair	L3
ScoreFlow [171]	Workflow-manager weights	Outcome label	Policy learning	L3
Learning to Compose [181]	Workflow-manager weights	Outcome label	Policy learning	L3
Workflow-R1 [183]	Workflow-manager weights	Execution result + Outcome label	Policy learning	L3
Learning to Hand Off [189]	Workflow-manager weights	Outcome label	Policy learning	L3
Self-Referential Code Modification				
Life-Harness [212]	Harness policy	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
HarnessFix [213]	Harness policy	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
Milkyway [214]	Harness policy	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
Self-Harness [215]	Harness policy	Trace + Execution result + Outcome label	Diagnostic repair	L3
POLARIS [216]	Harness policy	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
DemoEvolve [230]	Runtime scaffold	Trace + Execution result + Outcome label	Search-based optimization	L3
SIGA [231]	Runtime scaffold	Trace + Outcome label	Artifact synthesis	L3
HarnessForge [359]	Runtime scaffold + Model weights	Trace + Outcome label + Evaluator judgment	Diagnostic repair + Policy learning	L3
Agentic Harness Engineering [22]	Runtime scaffold + Evolution history	Trace + Outcome label + Evaluator judgment	Diagnostic repair	L3
Meta-Harness [217]	Runtime scaffold + Evolution history	Trace + Outcome label	Search-based optimization	L3
AgentFlow [218]	Runtime scaffold + Evolution history	Trace + Execution result + Outcome label + Evaluator judgment	Search-based optimization	L3
Adaptive Auto-Harness [239]	Runtime scaffold + Evolution history	Trace + Outcome label + Evaluator judgment	Search-based optimization	L3
Group-Evolving Agents [236]	Agent source code + Evolution history	Trace + Outcome label + Evaluator judgment	Search-based optimization	L3
Gödel Agent [226]	Agent source code + Improvement mechanism	Execution result + Outcome label	Meta-optimization	L4
SICA [227]	Agent source code + Improvement mechanism	Execution result + Outcome label + Evaluator judgment	Search-based optimization + Meta-optimization	L4
Darwin Gödel Machine [234]	Agent source code + Improvement mechanism	Trace + Outcome label + Evaluator judgment	Search-based optimization + Meta-optimization	L4
Huxley-Gödel Machine [235]	Agent source code + Improvement mechanism	Outcome label + Improvement statistics	Search-based optimization	L4
Red Queen Gödel Machine [360]	Agent source code + Evaluation mechanism	Outcome label + Evaluator judgment + Improvement statistics	Search-based optimization + Meta-optimization	L4
HyperAgents [23]	Agent source code + Improvement mechanism	Outcome label + Improvement statistics	Search-based optimization + Meta-optimization	L4

 doi:10.20944/preprints202608.0051.v1

Work	Object	Evidence	Mechanism	Level
Harnessing Agentic Evolution [237]	Improvement mechanism	Trace + Execution result + Outcome label + Improvement statistics	Meta-optimization	L4
EvoX [240]	Improvement mechanism	Outcome label + Improvement statistics	Meta-optimization	L4
ANCHOR [245]	Governance mechanism	Safety oversight	Governed commit	L3
Statistical Gödel Machine [241]	Governance mechanism	Safety oversight	Governed commit	L3
ANNEAL [243]	Governance mechanism	Safety oversight	Diagnostic repair + Governed commit	L3

3.1. Module-Level Self-Improvement

For a fixed agent to improve across tasks, knowledge accumulated from earlier work can be reused to guide its subsequent behavior. At the module level within the harness, that carry-over is localized within memory, skill and tool, or prompt and context modules: experience is written into memory, stabilized as reusable skills or tools, or folded back into the prompts and contexts that condition later decisions. The central move is to externalize learning into artifacts that a mostly fixed harness can inspect, retrieve, revise, and reuse, rather than to derive them from scratch on each new task. Although this is the simplest form of self-improvement, it allows improvements to persist across runs and thereby supports more extensive forms of harness evolution.

3.1.1. Memory Self-Modification

Memory self-modification refers to improving an agent by persistently changing the external memory module that conditions later executions, without updating the model parameters. This is different from static RAG or simple long-context retrieval: the memory is produced by the agent's own interaction history and is reused across later tasks. Early work establishes this non-parametric form of self-improvement by turning one-off execution traces into reusable textual experience. Reflexion writes task feedback into verbal self-reflections stored in an episodic memory buffer, allowing later trials to avoid previously observed mistakes [26]. ExpeL expands the same idea from trial-level reflection to experiential learning, extracting natural-language insights from collected trajectories and recalling both insights and examples at inference time [27]. Later systems make the experience artifact more specialized. SWE-Exp builds an experience bank from successful and failed software-repair trajectories so that coding agents can reuse repair patterns [28]; ReasoningBank distills successful and failed interaction histories into reasoning memory for later strategy selection [29]; FLEX organizes continual reflections into an inheritable experience library whose accumulated contents improve later agents [30]. Decocted Experience further emphasizes that useful memory is not a full transcript but a distilled, coherent, and retrievable essence of prior experience [31]. In this first stage, the memory module improves mainly by changing its contents: lessons, strategies, and procedural hints accumulate outside the fixed model and condition future behavior.

As agents operate over longer horizons, the central problem shifts from storing more experience to organizing, compressing, and maintaining it. A flat list of transcripts or reflections quickly creates redundancy, conflict, retrieval noise, and context pressure. MemGPT provides an early architectural anchor by treating LLM context as a memory hierarchy, where information can be moved between working context and archival storage [32]. A-MEM makes the store more self-organizing through Zettelkasten-style notes, dynamic indexing, and links that allow new memories to update the attributes and connections of older ones [33]. SimpleMem compresses interaction histories into structured semantic memory units, synthesizes related memories online, and plans retrieval according to the current intent [34]. RecMem adds a timing principle for consolidation: repeated semantic patterns, rather than every incoming interaction, trigger the extraction of episodic or semantic memory [35]. Graph and document structures offer another way to keep long-term memory usable. EXG organizes successes and failures as an experience graph that can grow during execution and be reused later [36]; FluxMem models memory as continuously evolving connectivity, repairing useful links, pruning interference, and distilling recurrent trajectories into procedural circuits [37]; Infini Memory maintains topic-structured documents that aggregate evidence and revise facts over time [38]. DCPM

 doi:10.20944/preprints202608.0051.v1

separates fast belief-revision writes from slower schema, intention, and cross-domain pattern induction [39], while Auto-Dreamer decouples online acquisition from offline consolidation through a learned consolidator that replaces redundant memory regions with compact abstractions [40]. Together, these systems turn memory from a collection of records into an evolving long-term state.

A further step is to make memory writing, retrieval, and reuse adaptive. Long-running agents must decide which memories remain useful, which memories deserve trust, and which abstractions transfer to new tasks. Live-Evo treats deployment as online memory evolution: an Experience Bank records what happened, a Meta-Guideline Bank summarizes reusable guidance, and feedback reinforces helpful memories while reducing the influence of stale or misleading ones [41]. MemQ introduces credit assignment by propagating Q-values backward through a memory provenance DAG, so memories that helped generate later useful memories receive delayed credit [42]. R^2-Mem applies a similar idea to memory search, evaluating successful and unsuccessful search trajectories and distilling them into reusable search experience [43]. AdaMEM addresses the timing of retrieval in long-horizon interaction by retrieving and synthesizing memory at each decision state instead of relying on a static episode-initial memory injection [44]. Memory Transfer Learning shows that cross-domain memory is most effective when it captures high-level debugging or validation principles rather than overly specific traces [45]. Other systems regulate the write and reuse process directly: TAME calibrates memory use through trust-aware executor-evaluator feedback [46], SAGE uses a novelty gate to decide whether newly extracted facts should be added, merged, ignored, or escalated to an LLM [47], and D-MEM routes memory updates through a fast/slow mechanism in which only high-surprise and high-utility inputs trigger deeper memory evolution [48]. These methods make clear that memory module self-improvement is not mere accumulation; it is a controlled process of selection, credit assignment, abstraction, calibration, and forgetting.

Recent work pushes the improvement target from memory contents and retrieval scores to the memory mechanism itself. MemEvolve decomposes agent memory systems into encoding, storage, retrieval, and management operations, and lets experiential knowledge and memory architecture co-evolve [49]. ALMA represents memory designs as executable code and uses a meta-agent to search over representations, storage rules, retrieval behavior, and update logic across domains [50]. M* similarly treats the memory module as an executable program containing schema, logic, and instructions, producing task-specific memory programs with different structures [51]. EvolveMem focuses on retrieval architecture, using failure logs to diagnose weak retrieval configurations and adjust them with rollback safeguards [52]. MemPro broadens this view to the full memory construction-retrieval pipeline, maintaining a version tree of runnable memory-system implementations and creating improved versions from recurring failure modes [53]. On the write side, MemSkill treats extraction, consolidation, and pruning as selectable and evolvable memory skills [54], while CluE adapts memory-extraction prompts to heterogeneous task groups [55]. This frontier moves memory self-modification from the question of what the agent remembers to the question of how the memory module improves its own representation, storage, retrieval, and update procedures. Most systems still rely on an externally specified improvement loop, but the mutable object now includes key parts of the memory-improvement mechanism itself, making memory module self-improvement an important bridge from experience accumulation toward higher-level agent self-improvement.

3.1.2. Skill and Tool Self-Improvement

In skill and tool self-improvement, reusable capabilities may take the form of executable code skills, function repositories, API wrappers, workflow routines, skill documents, skill contracts, or tool-use procedures. Early systems explore how agents can construct and accumulate these different forms of capability artifacts. CREATOR and Large Language Models as Tool Makers show that LLMs can synthesize tools or Python utility functions that separate abstract reasoning from concrete computation and are reusable in later calls [56,57]. ToolCoder organizes tool learning as code generation, execution feedback, and function-repository maintenance [58], while RefTool shows that tools can also be

 doi:10.20944/preprints202608.0051.v1

constructed from external references and domain knowledge [59]. In these systems, the main object of improvement is the external capability library; the outer acquisition procedure is usually still fixed.

Recent work shifts from generating isolated skills or tools to building capability libraries from multiple sources. One line abstracts agent trajectories, demonstrations, or task logs into reusable skills: SkillWeaver, SkillX, and Workflow-to-Skill distill web interactions, raw trajectories, or workflow traces into API-like skills, hierarchical skill knowledge bases, or workflow-preserving skill representations [60–62]. Socratic-SWE follows the same trace-to-skill pattern by distilling historical solving or repair trajectories into an Agent Skill Registry of procedural skills [63]. A second line compiles external resources into skills: Anything2Skill, OpenSkill, SkillFoundry, and DataCOPE turn documents, open-world resources, scientific repositories, notebooks, databases, or unsupervised exploration into skill packages and SkillBanks with lifecycle or verification signals [64–67]. As these libraries grow, self-improvement also requires curation. MUSE-Autoskill and SkillOS organize creation, memory, management, selection, and refinement into a skill lifecycle; SkillClaw aggregates multi-user trajectories into a shared SkillHub; and ToolLibGen performs the analogous cleanup on the tool side by deduplicating, refactoring, consolidating, and organizing generated tools [68–71]. This moves the field from one-off capability creation toward maintaining a reusable substrate of agent actions.

A capability library only improves future behavior when its contents are selected, composed, and revised appropriately. SkillOpt and Not All Skills Help approach this from the use side: the former treats skill invocation as an executive strategy, while the latter estimates per-skill causal contribution to suppress or repair harmful skills [72,73]. SkillComposer and Evolving Programmatic Skill Networks focus on composition and generalization: one decomposes skill evolution into create, improve, and merge operations, while the other organizes executable symbolic programs into a skill network that supports planning, fault localization, maturity gating, and rollback [74,75]. Other systems refine the skills themselves. SkillAxe and SkillAudit revise LLM-authored skills through structured diagnostics or paired trajectories; SkillGen, CoEvoSkills, and VASO improve self-generated skills through verified synthesis, surrogate-verifier co-evolution, or formal skill contracts [76–80]. Together, these works show that skill/tool self-improvement is not linear library growth, but an ongoing process of selection, composition, revision, and retention.

The boundary between skills and tools is also becoming less rigid as capability libraries enter larger agent-improvement loops. Beyond Static Tools, MetaForge, and SkillSmith exemplify the move from fixed toolsets to dynamic tool ecosystems: agents can synthesize tools at test time, retrieve and adapt tools on demand, or apply update bundles that jointly modify skills and tools through operations such as wrapping, editing, composing, splitting, and retiring [81–83]. Coding-agent systems show the same integration in a different environment: Socratic-SWE feeds skill libraries into code repair, task generation, and solving loops, while Confucius Code Agent connects persistent notes and modular extensions to the build, test, and debugging process of real codebases [63,85]. Overall, the trajectory of this area is from individual reusable skills or tools toward a managed external capability layer. Most systems remain harness-level L3 self-improvement, but learned curators, co-evolving verifiers, and open-world resource-to-skill pipelines begin to improve not only the library contents but also parts of the mechanism by which that library is improved.

3.1.3. Prompt and Context Self-Optimization

Prompt and context self-optimization is the most lightweight form of harness self-improvement: model weights, external tools, and executable code may remain fixed, while the natural-language control surface that conditions later execution is revised. The optimization target can be a system prompt, task instruction, few-shot demonstration set, prompt program, context playbook, guideline, or rule document. The earliest and simplest line treats prompts as black-box textual parameters. OPRO establishes the basic loop by asking an optimizer LLM to propose new instructions from prior prompt-score histories [86], while PromptBreeder extends the idea by co-evolving task prompts and mutation prompts, making prompt improvement partly self-referential [87]. Later prompt-search methods vary the supervision and feedback budget: Self-Supervised Prompt Optimization reduces

 doi:10.20944/preprints202608.0051.v1

dependence on labeled references, DLPO imports robustness and update-control ideas into textual optimization, ReElicit builds interpretable elicited feature spaces for Bayesian optimization of system prompts, PrefPO uses pairwise preferences and natural-language criteria, APEX dynamically selects informative development examples, TextBO casts language-space search in a Bayesian-optimization style, and Feedback Descent uses pairwise comparisons with textual rationales for open-ended text artifacts [88–94]. In more agentic settings, the same search loop becomes grounded in executable analysis and environment interaction: SPEAR uses code execution and rollback validation to propose and retain prompt edits, while environment-grounded prompt optimization uses rollout returns and behavior analysis to revise game-agent prompts [95,96].

As agents become compound LM programs, the optimized object expands from a single instruction to a bundle of LM-call interfaces, demonstrations, prompting patterns, role prompts, and local context. DSPy makes this shift explicit by representing LM calls as declarative modules in its programming abstraction and compiling their prompts and demonstrations [97]. MIPRO jointly optimizes instructions and few-shot examples across multi-stage LM programs [98], and AutoPDL searches over prompting patterns and demonstrations to produce readable, executable prompt programs for agents [99]. ADO adds that the input data and example organization inside a prompt are themselves part of the control interface [100], while cross-model prompt translation separates transferable semantic directives from model-dependent interface constraints so prompt programs can adapt when the underlying foundation model changes [101]. Once the prompt surface is distributed across LM calls or agents, the central bottleneck becomes credit assignment: the system must identify which prompt, role instruction, demonstration, or context fragment caused a downstream error. TextGrad abstracts LLM critique as a gradient-like signal over textual variables [102], and GEPA uses execution and evaluation traces for reflective prompt evolution [103]. Textual Equilibrium Propagation, ContraPrompt, GTBP, and multi-agent prompt credit assignment then refine this feedback path through local equilibrium updates, success/failure trace contrast, graph-structured target propagation, and temporal/structural blame decomposition in multi-agent systems [104–107]. EEVEE and Combee further scale prompt learning to heterogeneous task streams and large volumes of agentic traces, pushing prompt-program adaptation from offline tuning toward deployment-time learning [108,109].

A parallel line of work focuses on constructing the context interface: what information enters the context window, how that information is structured, which rules or playbooks become executable constraints, and how the system updates and prunes the context and resolves conflicts within a finite context budget. Dynamic Cheatsheet and ACE provide two early anchors. Dynamic Cheatsheet compresses prior attempts into short strategy hints that can be injected into later prompts [110], while ACE uses generator, reflector, and curator roles to produce and organize context playbooks for subsequent executions [111]. SCOPE, Reflective Context Learning, and Unified Context Evolution make this context-compilation process more systematic: they derive context-management guidelines from execution traces, abstract context-space updates into reusable optimization primitives, or wrap information from memory, strategy, workflow, and skill sources into typed context units that can be selected, pruned, and budgeted [112–114]. KACE and Context Training with Active Information Seeking emphasize grounding: the former organizes knowledge as task-retrievable epistemic guidance, and the latter lets the optimizer acquire missing external information before updating context [115, 116]. In domain-specific settings, MEMO, SEEK-SQL, and Trace2Policy convert self-play outcomes, positive/negative execution trajectories, or expert behavior traces into hints, guidelines, or rule documents that can be consumed by the next inference-time decision [117–119]. Across these systems, context becomes an optimizable interface for the next run, not merely a larger container for previous experience.

From an RSI perspective, most prompt or context self-optimization still improves artifacts under externally specified update rules: the system repeatedly proposes, evaluates, and preserves better prompts, demonstrations, guidelines, or context units, while the outer improvement procedure remains fixed. A smaller set of systems begins to make the prompt optimizer itself mutable. PromptBreeder

 doi:10.20944/preprints202608.0051.v1

already co-evolves mutation prompts with task prompts, even though the top-level evolutionary framework remains hand-designed [87]. SePO goes further by optimizing the prompt agent's own system prompt alongside task-agent prompts, turning prompt optimization from a hand-written procedure into a reusable optimizer skill [120]. Learning to Self-Evolve trains a model to edit its own test-time context and rewards edits that improve downstream performance [121]. Overall, prompt and context self-optimization provides a fast, inexpensive, and inspectable adaptation layer. Within a broader self-evolving agent harness, prompt and context artifacts serve as a control interface connecting memory, skills, workflows, and code-level adaptation.

3.2. Orchestration and Architecture Search

Once agents acquire reusable memory, skills, tools, and context, a second bottleneck becomes visible: difficult tasks are rarely solved by one uninterrupted model call or one fixed execution path. Orchestration work treats the agent harness as a computation process that can be branched, searched, routed, and reorganized. Improvement comes from changing how trajectories are branched, specialized agents are assigned, feedback is routed, and communication is scheduled, so that the system can spend effort where uncertainty is high and reuse useful intermediate results rather than repeatedly starting from scratch. Within RSI systems, orchestration is the layer that turns improved harness modules into adaptive problem-solving systems, making self-improvement not only about owning stronger parts but also about organizing them effectively under real compute and reliability constraints.

3.2.1. Test-Time Compute and Trajectory Orchestration

Scaling at test time starts from a simple weakness of ordinary agent execution: a single run is a fragile commitment to one sequence of thoughts, tool calls, and environment states. Test-time compute and trajectory orchestration therefore improve a fixed or mostly fixed agent harness by giving it more paths to explore, compare, and combine. The basic unit of improvement is a reasoning path, tool-action sequence, repository state, or full execution trajectory. Early search-oriented methods provide the conceptual basis: Tree of Thoughts treats intermediate thoughts as expandable and evaluable search nodes, while Language Agent Tree Search extends this idea to agents that interleave reasoning, acting, planning, and environmental feedback [122,123]. A more systematic view appears in Scaling Test-time Compute for LLM Agents, which compares parallel sampling, step-wise Best-of-N, beam or tree search, sequential revision, and verifier-based merging across agent tasks [124]. In software engineering, this trajectory view becomes especially concrete. SWE-Search applies Monte Carlo Tree Search to software repair trajectories, using action generation, value estimation, and discriminator-style debate to decide which states to expand [125]. Scaling Test-Time Compute for Agentic Coding frames the long-horizon coding setting around three coupled problems: representing long rollouts compactly, selecting among them reliably, and reusing their useful information through recursive tournament voting and parallel-distill-refine strategies [126]. Other systems enlarge the candidate space through evolutionary or population-style search. Satori-SWE evolves candidate patches through selection and mutation, SE-Agent operates directly on multi-step trajectories through revision, recombination, and refinement, and AgentGA moves the search object to the agent-seed level, evolving task prompts and parent archives that initialize fresh autonomous runs [127–129]. Even repository exploration can be treated as a test-time orchestration problem: domain-scoped parallel exploration and FastContext-style repository explorer agents show that context acquisition itself can be parallelized, specialized, and compressed before the main coding trajectory proceeds [130,131].

More paths, however, only help when the harness can tell which ones deserve attention. As trajectory pools grow, the problem shifts from exploration to feedback: which branches should receive more compute, which candidates should be pruned, and which failed attempts should be repaired rather than restarted. SWE-TRACE uses rubric-based process reward models at inference time to guide action-level sampling and prune weak action candidates before a full trajectory is completed [132]. EGSS makes the allocation problem explicit by using tool and action entropy to detect high-ambiguity steps, invoking auxiliary judgment only where uncertainty is high, and selecting final

 doi:10.20944/preprints202608.0051.v1

patches with augmented regression and edge-case tests [133]. Some methods move selection earlier in the software-engineering workflow. SWE-Manager generates multiple natural-language repair proposals before coding, then selects and synthesizes a golden proposal, while architecture-aware multi-design generation creates several repository-level designs and filters them with static and dynamic impact analysis [134,135]. A related shift happens after execution has already exposed a failure: the system can refine or replay the useful parts of a trajectory instead of discarding the whole attempt. SWE-Replay archives previous trajectories and branches from informative intermediate steps without relying on an LLM-as-judge quality estimate [136]. PIVOT treats the trajectory as an optimizable plan-execution object: it preserves validated prefixes, inspects execution losses, rewrites unsupported suffixes, and verifies global constraints [137]. Persistent cross-attempt state optimization for repository-level code generation keeps task-local success knowledge, failure knowledge, and a historical-best repository across attempts, so later runs can inherit useful state while avoiding regressions [138]. AlphaApollo's propose-judge-update loop is a broader agentic reasoning example of the same pattern: test-time improvement arises not from a single retry, but from repeated proposal, verification, and update cycles over the evolving solution state [139]. These systems show that effective test-time scaling is not proportional to raw rollout count; it depends on whether feedback is local enough to intervene early, reliable enough to guide selection, and persistent enough to influence subsequent decisions.

Reliable feedback can guide search and revision within a task, but it does not eliminate duplicated computation across attempts: agents may still reconstruct context, repairs, and execution states that previous attempts have already produced. This motivates methods that make prior computation and prior experience available to the next trajectory rather than treating each run as isolated. Log-Augmented Generation stores and retrieves prior reasoning logs through key-value cache representations, directly reusing earlier computation rather than distilling it into textual reflection [140]. Decocted Experience instead extracts, organizes, and retrieves distilled experience from trajectories and feedback, making context construction itself a test-time scaling axis [31]. Bifrost adapts successful prior trajectories to new contexts through representation-level steering, while Lingxi retrieves procedural knowledge from historical issue-resolution cases to guide online analysis and scaling [141,142]. FailureMem and EvoRepair provide repair-specific variants by converting failed attempts, diagnostic lessons, and vulnerability-repair experience into reusable guidance for later fixes [143,144]. These experience-oriented methods sit near memory self-improvement, but their role here is narrower: they matter when stored trajectories or distilled lessons directly change current inference-time search, selection, or repair. At the same time, reusable trajectories require runtime support: the system must be able to capture, fork, restore, and compare executions whose actions have side effects. Shepherd records model calls, tool calls, and environment changes as formalized execution traces that can be monitored, forked, replayed, modified, and resumed, while DeltaBox provides sandbox checkpoint and rollback mechanisms for stateful agents whose actions affect files, processes, or tests [145,146]. Effective Feedback Compute introduces a scaling measure that credits feedback only when it is informative, valid, non-redundant, and retained for subsequent decisions, thereby distinguishing useful feedback from raw expenditure in tokens, tool calls, or runtime [147]. In summary, the key to effective test-time scaling is ensuring that additional inference compute produces reliable feedback and reusable intermediate results that guide final decisions.

3.2.2. Agentic Unit Composition

Agentic unit composition concerns the participant layer of an agentic computation: beyond optimizing the search process and message flow, the system must decide what kinds of agents are available to do the work. At the finest level of composition, the internal structure of an agent becomes part of the design space. Instead of treating an agent as a single hand-written template, AgentSquare constructs candidate systems from specialized agents responsible for planning, reasoning, tool use, and memory, then searches over their evolution and recombination through standardized interfaces [148]. AgentSwift keeps this compositional view but enlarges the space: workflows are searched together with

 doi:10.20944/preprints202608.0051.v1

specialized agents for memory, tool use, and planning, while a value model and uncertainty-guided MCTS make exploration practical [149]. AgentSpec adds a controlled-composition perspective by representing embodied systems as typed combinations of agents responsible for perception, memory, reasoning, reflection, action, and optional learning, making agent compatibility and interaction effects visible [150].

At a coarser granularity, the optimization target shifts from the internal modules of a single agent to the set of agents that jointly solve a task. AutoAgents establishes the basic task-conditioned pattern by generating specialized agents and an execution plan for the input task instead of relying on a fixed hand-written team [151]. EvoAgent turns role construction into an evolutionary process, varying roles, skills, prompts, and settings through mutation, crossover, and selection to expand a seed agent into a diverse multi-agent population [152]. ADAS broadens the design space further by letting a meta-agent iteratively propose, test, archive, and reuse code-defined child agent systems [153], while MAS-GPT learns to generate query-specific executable multi-agent systems in a single inference pass [154]. Later systems make these generated teams more structured and executable. Meta-Agent maps task descriptions into verified multi-agent systems with agent specifications, input/output contracts, tool configurations, and verification criteria [155]. LEMON generates orchestration specifications with task-specific roles, duties, capacity levels, and dependencies [156], and OrchMAS dynamically instantiates domain-aware scientific expert agents with tailored prompts [157]. EvoMAS represents multi-agent systems as structured configurations over roles, prompts, tools, model assignments, and communication structure, then evolves these configurations through selection, mutation, crossover, and cross-query reuse [158]. EVOCHAMBER adds a population-level substrate in which a coevolving agent pool forms teams through anchor, complement, and scout roles, while lifecycle operators fork, merge, prune, and seed agents under performance pressure [159]. The emphasis has therefore shifted from manually designing a team to making team construction itself an adaptive architecture layer.

Generated teams, however, are still a coarse answer if every query is forced through the same collection of roles or the same model choices. The next movement is to keep a heterogeneous pool of candidate units and select from it at query time. MaAS formulates this through an agentic supernet whose feasible operators include reasoning, debate, refinement, testing, tool-use, and early-exit patterns; each query samples a cost-aware multi-agent architecture from the learned operator distribution [160]. AutoMaAS extends the same idea with operator lifecycle management, so operators can be generated, fused, eliminated, and reweighted from online feedback [161]. HieraMAS makes heterogeneity internal to each functional role: a role becomes a supernode backed by mixtures of LLMs, with learned model selection, role pruning, and topology selection [162]. SciOrch applies this selection view to frontier scientific reasoning, where a lightweight orchestrator decomposes multimodal questions, delegates subproblems to selected expert LLM APIs, and synthesizes the final answer [163]. Iterative Critique-and-Routing Controller similarly treats coordination as a multi-round choice over a heterogeneous agent pool: the controller critiques the current draft, decides whether to stop, and routes the next step to the most suitable agent [164]. X-MAS supplies empirical support for this direction by showing that different LLMs specialize in different MAS roles and domains [165], while PASS and DAAO show how probabilistic tool/operator paths and model routing can be conditioned on domain structure, task difficulty, and cost [166,167]. Across these systems, orchestration-level self-improvement focuses on adaptive agent composition: the harness learns which specialized agents should be assembled for the computation at hand.

3.2.3. Execution Flow, Topology, and Workflow Optimization

Once the participating units have been chosen, a second architecture-search problem remains: how those units should be connected into an executable computation. One family of work treats the execution flow itself as the optimization target. GPTSwarm provides an early formulation by representing a language-agent system as an optimizable graph, where nodes are agents or operations and edges carry information among them [168]. AutoFlow, AFlow, and ScoreFlow make this idea more explicit for workflow generation: workflows are no longer hand-written prompt chains, but natural-language pro-

 doi:10.20944/preprints202608.0051.v1

grams, code-represented workflows, or continuous workflow representations that can be searched and optimized from execution feedback [169–171]. Flow expresses agentic workflows as activity-on-vertex graphs whose dependencies and subtask allocation can be refined over time, while MermaidFlow and EvoAgentX emphasize executable workflow representations with explicit roles, data-flow constraints, evaluation, and evolution loops [172–174]. Later work expands the search procedure itself. EvoFlow and SEW use evolutionary mechanisms to explore diverse workflow structures, VFlow and HALO adapt structured search to domain-specific or hierarchical orchestration, and HyEvo evolves hybrid workflows that mix LLM calls with deterministic code nodes [175–179]. Other systems improve how workflow search generalizes and localizes edits: AdaptFlow learns reusable workflow initializations, Learning to Compose builds cross-domain workflow capabilities, BayesFlow casts workflow generation as probabilistic inference, Workflow-R1 optimizes multi-turn workflow construction, JudgeFlow uses block-level judgment to target local workflow edits, and RobustFlow adds preference-based pressure for stable workflow structures [180–185]. Verification-aware and state-machine formulations make these local revisions more explicit. Lean4Agent formalizes workflow graphs, step contracts, and execution trajectories in Lean, while LeanEvolve translates localized verification failures into targeted revisions of the workflow specification [232]. EvoFSM instead exposes a finite-state workflow whose states, transitions, and state-specific instructions can be changed through atomic edit operations, with a persistent memory reusing successful configurations and operation sequences across tasks [233]. This line also includes execution-aware workflow optimization: LAMaS learns latency-aware orchestration graphs for parallel multi-agent execution, FlowCompile treats structured LLM workflows as a compiler optimization space, GraphFlow manages graph-structured workflows for efficient serving, Learning to Hand Off studies interface-constrained handoff policies, and latent-space branch synthesis changes how parallel workflow branches are merged for later reasoning [186–190]. Across these systems, self-improvement happens at the level of the workflow program: the system changes the dependency graph, ordering, handoff, parallelism, or executable structure through which fixed or selected units produce an answer.

Whereas workflow-search methods optimize the end-to-end execution structure, another line isolates communication among participating agents as the main object of self-improvement. These methods determine which participating agents exchange information, when communication occurs, how densely they are connected, and whether learned topological patterns can transfer across tasks. GPTSwarm already makes graph connectivity optimizable, while EvoMAC evolves multi-agent collaboration networks for software development and G-Designer uses graph neural methods to design task-aware communication topologies [168,191,192]. Cost and redundancy then motivate sparse communication. AgentPrune removes unnecessary or harmful message-passing edges, Adaptive Graph Pruning jointly prunes agents and communication links, and SafeSieve progressively prunes communication based on accumulated experience [193–195]. More recent methods generate or select topology directly: AMAS determines task-specific communication graphs, graph-diffusion topology generation and OFA-MAS synthesize task-adaptive graphs through generative models, ST-EVO evolves communication graphs across spatial and temporal dimensions, Agent Q-Mix frames agent action and connection choices as cooperative reinforcement learning, RADAR adds redundancy-aware diffusion, and differentiable mixture-of-agents models make topology selection trainable through sparse activation [196–202]. Domain- and state-conditioned topology methods push this further: AgentConductor evolves DAG communication structures for competition-level code generation, TacoMAS co-evolves topology and capability at test time, TopoPrior and MasFACT learn transferable topology priors for later collaboration, and DySCo selects sparse trust-aware edges per reasoning round [203–207]. Finally, topology optimization can be conditioned on richer runtime signals. Retrieval-conditioned topology selection chooses collaboration modes from codebase complexity under budget conservation, response-conditioned parallel-to-sequential orchestration decides whether parallel agent outputs should trigger sparse DAG refinement, Learning to Interrupt optimizes when agents should enter each other's communication stream, and Learning to Communicate moves beyond explicit edges

 doi:10.20944/preprints202608.0051.v1

toward learned communication channels and latent protocols [208–211]. Taken together, these works shift multi-agent orchestration from static debate or fixed pipelines toward adaptive communication structures whose edges, density, timing, and reusable priors become part of the evolving harness.

3.3. Self-Referential Code Modification

Memory, skill, prompt, and workflow optimization each targets a specific class of harness artifact. Self-referential code modification expands the editable scope to the harness or scaffold itself, allowing broader changes across its components and their interactions. At the more recursive end of this spectrum, the modified agent participates in subsequent improvement cycles, becoming both the object of improvement and part of the mechanism that produces further changes. This self-reference makes code-level modification the closest harness-side analogue of recursive self-improvement in this section, while also placing greater demands on validation, rollback, auditability, and controlled promotion.

3.3.1. Bounded Self-Harness and Policy Repair

The most local form of self-referential harness modification is bounded repair: the agent does not rewrite its full source code or redesign its entire scaffold, but it can turn execution evidence into persistent changes to the operating rules and interfaces that help future runs. At the lower level, these changes target action interfaces, tool contracts, runtime constraints, and domain-specific procedural rules. Life-Harness adapts the interface rather than the model: recurring interaction failures are converted into reusable interventions over environment contracts, action realization, procedural guidance, and trajectory regulation [212]. HarnessFix makes the trace-to-patch structure explicit. It compiles failed trajectories and harness code into a diagnostic representation, attributes failures to specific harness layers, generates scoped repair operators, and validates patches against regressions [213]. Milkyway applies this form of persistent harness adaptation to forecasting agents: it updates a persistent forecasting harness from pre-resolution signals across repeated forecasts, so later predictions are guided by revised procedures for evidence tracking, factor revision, and uncertainty management [214]. Across these systems, the important shift is that experience no longer only becomes memory or skill content; it changes the execution rules under which the next run is conducted.

A second group applies bounded repair directly to policy and harness code. Self-Harness lets the same fixed model mine its own failures, propose minimal edits to its harness, and promote only those edits that improve held-in tasks without regressing on held-out validation [215]. POLARIS applies a similar idea to small language models through experience-abstracted policy repair: failed validation tasks are distilled into reusable strategies and then implemented as auditable policy-code patches that can be executed in later tasks [216]. Around these core repair mechanisms, recent work also develops the surrounding infrastructure needed for reliable harness evolution. Agentic Harness Engineering uses trajectory observability to edit the coding-agent harness implementation with predicted impact and verification feedback [22]; Meta-Harness optimizes harness code end-to-end from source, scores, and execution traces [217]; and AgentFlow synthesizes multi-agent harnesses by revising roles, prompts, tools, communication topology, and coordination protocols from runtime signals [218]. Other systems strengthen the diagnosis and promotion stages: AgentDevel frames agent improvement as release engineering with executable diagnosis and non-regression gates [219], Wink provides runtime recovery from coding-agent misbehaviors through targeted interventions [220], AgentFixer connects failure detection to root-cause analysis and fix recommendations [221], CausalFlow performs step-level causal attribution and counterfactual repair [222], and AgenTracer and Who&When study automated failure attribution in multi-agent traces [223,224]. Together, these works mark a transition of self-improvement from modules used by the harness to modifications of the harness itself that persist across future executions.

3.3.2. Source-Code and Scaffold Self-Modification

Source-code and scaffold self-modification makes the agent's executable implementation itself editable, allowing revisions to span multiple interacting components. STOP gives an early formulation

 doi:10.20944/preprints202608.0051.v1

in recursive self-improving code generation: a seed improver calls a fixed language model to improve downstream programs, then applies the same improvement process to its own Python source, producing successor improvers with stronger scaffolding strategies such as beam search, genetic algorithms, and simulated annealing [225]. Gödel Agent moves the same self-referential idea into the agent policy itself: the agent inspects its runtime memory and uses monkey patching to revise its decision procedure and self-update logic, so that the modified policy participates in later recursive calls [226]. In software engineering settings, A Self-Improving Coding Agent provides a more complete repository-level loop. The coding agent edits its own Python codebase, tools, subagents, context management, and benchmark machinery, while the best archived agent version becomes the next meta-agent for subsequent improvement rounds [227]. MOSS pushes the object of modification into deployed agent substrates, rewriting source-level harness logic such as routing, state management, dispatch, hooks, mediator behavior, and session lifecycle; candidate changes are packaged as trial images, replayed against production-failure batches, and promoted through consent and rollback gates before entering the live container [228]. Self-Evolving Software Agents describes the same trend from a BDI-LLM architecture perspective, where experience can induce new goals, revised reasoning structures, intention-selection patterns, and executable actions that are retained for later behavior [229]. Around this central line, recent work increasingly treats other executable elements of the scaffold as evolvable parts of the agent harness. DemoEvolve treats the frozen-model agent's prompts, state tracking, action filters, tool exposure, and control flow as an executable harness, using prior harness code, scores, rollout traces, and human demonstrations to guide active model-facing edits that are selected and reused on held-out seeds [230]. SIGA shows a similar pattern at the adapter layer: simulator-specific contracts are injected through context, tool, and termination interfaces, and prior trajectories are used to rewrite the adapter's primer, procedural-memory cheatsheet, and auxiliary skills into a reusable plugin package [231]. Taken together, these systems show a shift from agents that merely write code as task output to agents whose improver programs, policies, repositories, runtime tools, production harnesses, or adapter plugins can themselves be rewritten, verified, and reused as part of the next agent execution.

3.3.3. Open-Ended and Governed Recursive Self-Modification

Open-ended recursive self-modification asks whether self-editing can become a sustained evolutionary process rather than a single scaffold repair or a one-step source-code patch. The theoretical reference point is the Gödel Machine: a fully self-referential system whose own problem-solving code and improvement procedure are both subject to rewriting when sufficient evidence supports the change [25].

Recent LLM-agent systems operationalize this ideal by progressively exposing different parts of an archive-based improvement loop to adaptation. The Darwin Gödel Machine (DGM) replaces the original Gödel Machine's impractical proof obligation with empirical validation: a coding agent edits its own repository, evaluates the resulting descendant on downstream tasks, and retains viable variants in a branching archive from which later agents can be selected for further modification [234]. Rather than committing only to the strongest current variant, this archive preserves diverse branches and temporarily suboptimal stepping stones that may support stronger descendants in later generations. DGM thus establishes a practical RSI loop in which the agent implementation being improved can also produce its next modification, although archive search, parent selection, and part of the surrounding improvement procedure remain externally specified. The Huxley–Gödel Machine (HGM) changes how this archive is searched. It addresses the fact that an agent's current task performance may be a poor indicator of its ability to generate better descendants by introducing clade metaproductivity, which evaluates a candidate through the outcomes of its descendant subtree. Metaproductivity-aware selection can therefore allocate expansion effort toward lineages with greater long-term potential rather than merely toward the highest-scoring current node [235]. Whereas HGM changes which lineage is expanded, Hyperagents allows the self-modification procedure to evolve as well. A hyperagent places the task agent and the meta-agent responsible for proposing modifications in the same editable program. For each selected parent, the meta-agent examines the agent's repository, previous evaluation

 doi:10.20944/preprints202608.0051.v1

results, and remaining iteration budget to diagnose failures and decide what to modify. Because the meta-agent's own implementation is part of the repository, it can rewrite both the task-solving code and the logic used to analyze performance and generate future modifications. Over successive generations, DGM-Hyperagents (DGM-H) develops reusable meta-level mechanisms such as performance tracking and persistent memory. Since these mechanisms learn from evaluation feedback rather than depend on a fixed, handcrafted procedure, they can support self-improvement even when task performance and code-editing ability are not naturally aligned. DGM-H consequently improves agents for paper review and robotics reward design, while the meta-level capabilities acquired in these domains further transfer to Olympiad-level mathematics grading [23]. Building on HGM's lineage search and Hyperagents-style editable workspaces, Red Queen Gödel Machine (RQGM) places learned evaluators in the same evolvable workspace as task agents, allowing the capability being optimized and the mechanism that evaluates that capability to improve together [360]. To keep scores meaningful as evaluators change, RQGM freezes the active evaluator within each epoch, promotes only challengers validated against a held-out anchor, and removes records whose scores depend on a displaced evaluator. This design turns evaluation from a fixed external constraint into an evolvable part of recursive self-improvement without destabilizing lineage selection. Figure 4 illustrates an integrated framework that captures the core self-improvement loop of DGM and RQGM.

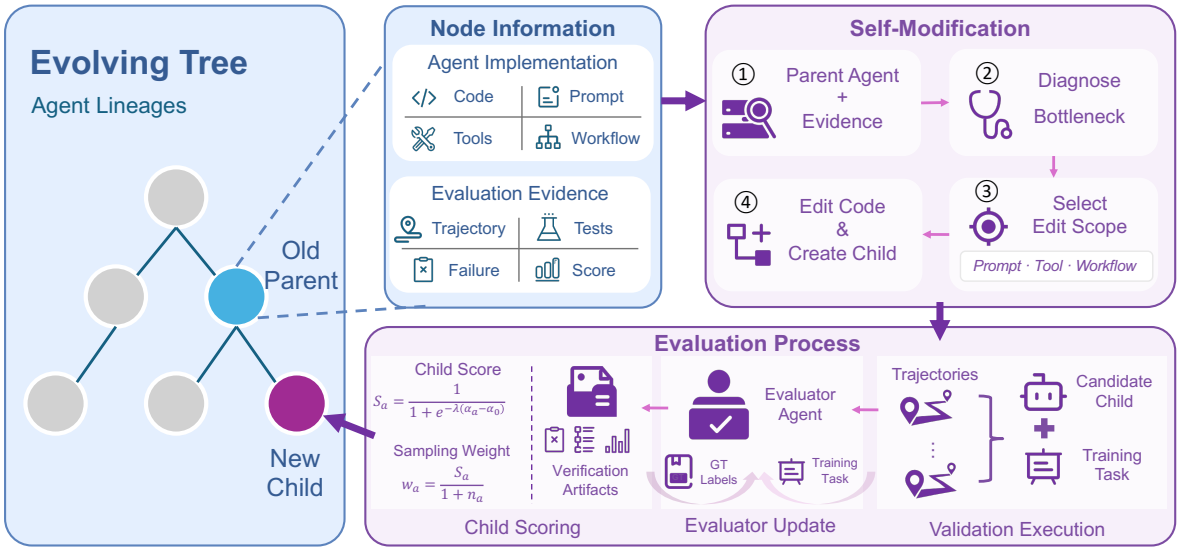

Figure 4. Conceptual synthesis of archive-based recursive self-improvement. The evolving tree retains agent lineages, with each node coupling an editable agent implementation, including code, prompts, tools, and workflows, with execution and evaluation evidence. A selected parent uses this evidence to diagnose bottlenecks, determine an edit scope, and rewrite its implementation to create a child, which is executed on sampled tasks to produce trajectories and verification artifacts. In DGM, valid children are added to the evolving tree, and each node's benchmark score and number of functioning children determine how likely it is to be sampled as a future parent. RQGM extends this loop by making the evaluator evolvable as well: the evaluator judges the child's trajectories, while task feedback and ground-truth labels also guide evaluator updates, enabling the task agent and its evaluation mechanism to improve together [234,360].

Group-Evolving Agents approaches self-improvement from a collective perspective, changing the evolutionary unit from an isolated parent to a group or population, selecting parent groups by performance and novelty, and sharing patches, trajectories, and failure experience so that discoveries from separate branches can be recombined by later offspring [236]. Other systems make the broader evolutionary process itself editable. Harnessing Agentic Evolution formulates evolution as an interactive environment: accumulated candidates, traces, failures, costs, and scores become process-level state, while a meta-editor changes the procedure or operating context that controls future evolution segments [237]. Bilevel Autoresearch makes a related move in automated research loops, where an outer loop reads the inner loop's code and experimental traces and injects new search mechanisms that

 doi:10.20944/preprints202608.0051.v1

alter subsequent research dynamics [238]. Broader work on harness and search-mechanism evolution extends the same trend: Meta-Harness searches over harness code using a filesystem history of prior sources, traces, and scores [217], Adaptive Auto-Harness maintains a harness tree for open-ended task streams and routes new tasks to specialized branches [239], and EvoX evolves parent-selection and variation strategies during LLM-driven program optimization [240]. Across these systems, the common movement is from producing a better next agent to maintaining self-modification history, selecting lineages with future potential, co-evolving the criteria that define improvement, sharing experience across branches, and changing the mechanism by which later improvements will be generated.

The same openness creates a governance problem: once self-modified agents, harnesses, skills, evaluators, or search procedures persist into future runtime, a bad commit can reshape the distribution of later proposals, contaminate the parent pool, or amplify objective hacking and safety drift. Statistical Gödel Machine addresses this at the commit-decision level by replacing formal proof obligations with statistical gates: a candidate edit is accepted only when paired evaluation and confidence certificates support superiority over the incumbent, while a global error budget controls cumulative risk across irreversible recursive edits [241]. Governed Evolution of Agent Runtimes generalizes this discipline into a runtime architecture in which prompts, tools, evaluators, routing policies, skills, and agent-generated code artifacts become persistent operational capabilities only through lifecycle promotion, validation, traceability, audit trails, and rollback [242]. ANNEAL gives a concrete instance of governed structural repair: recurring failures are converted into typed symbolic patches, but each patch must pass scoring, guardrails, canary tests, provenance recording, and deterministic rollback preparation before it enters future execution [243]. Safety monitoring adds another layer. SAHOO tracks goal drift, constraint preservation, and regression risk across recursive improvement cycles, making capability gains legible together with their alignment cost [244]. ANCHOR studies human-like supervision inside self-evolving systems, showing how review at phases such as task proposal, planning, output verification, and execution-result feedback can mitigate safety degradation while preserving core capabilities [245]. Governed recursive self-modification therefore combines the search for better successor agents with a controlled process for deciding which changes are retained. Sustainable RSI requires validation signals, statistical confidence, rollback paths, auditability, and automated oversight to become part of the self-improvement loop itself.

4. Agent Data System Self-Improvement

The lifecycle of training and evaluation data is a critical dimension of self-improving agent systems. Existing research on self-improvement mainly focuses on model training or agent harness design. However, the data system that supplies training and evaluation signals is equally important to the continued improvement of an agent system.

Figure 5 illustrates that the agent data system typically consists of two stages: **data production** and **data utilization**. The data production stage comprises four modules: environment generation or simulation, task synthesis, trajectory synthesis, and verification and quality assurance. Together, these modules form a pipeline that generates and validates training and evaluation data. The data utilization stage consists of curriculum design and adaptation, which determine how the generated data are selected and allocated for agent training. These two stages are closely coupled: training requirements from the utilization stage guide subsequent data production, while data production determines what the utilization stage can operate on.

 doi:10.20944/preprints202608.0051.v1

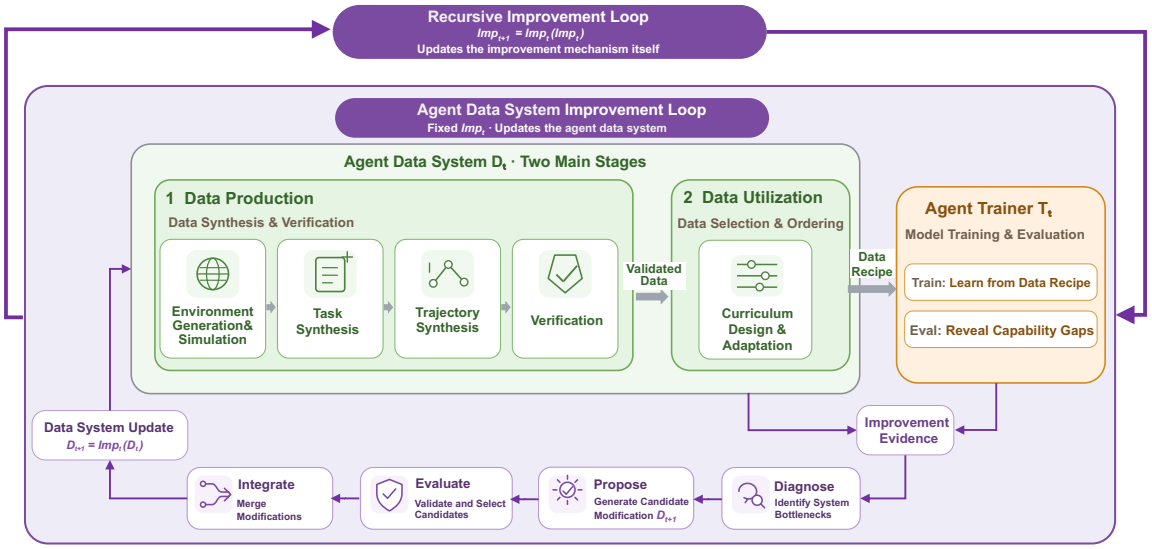

Figure 5. Conceptual framework of the agent data system, its improvement loop, and the recursive improvement loop. The data system consists of two main stages. *Data production* encompasses four modules that create new training and evaluation data. *Data utilization* consists of curriculum design and adaptation, which allocate the produced data for agent model training. The improvement loop uses training and evaluation feedback to diagnose bottlenecks, propose modifications, validate candidates, and promote a successor data system $\mathcal{D}_{t+1}$. At the outer level, improvement histories and validation outcomes provide evidence for revising the improvement mechanism Imp_t itself, yielding Imp_{t+1} when a candidate is promoted.

In this survey, **agent data system self-improvement** refers to an autonomous process in which a data system iteratively improves itself to generate higher-quality data for agent training and evaluation, without human intervention. To meet this definition, the modules within the data system need to actively modify their own implementations or adapt their behaviors based on verification results and model training outcomes, as shown in Figure 6.

Compared with conventional data pipelines that rely on manual design and improvement, a self-improving data system has advantages on both the training and evaluation sides. On the training side, it enables continuous generation of difficulty-adaptive curricula that target the current weaknesses of the agent model [246]. This enables adaptation of the effective training distribution beyond the limitations of a fixed corpus. On the evaluation side, the data system can continuously synthesize and refine benchmarks that evolve with the agent's capabilities. This helps prevent diminishing discriminability in evaluation and ensures that the feedback signals remain informative for subsequent improvement.

This section is organized by the *scope* of self-improvement. We first discuss *self-improvement in agent data production* (Section 4.1), where each data production module improves itself to adaptively produce agent data tailored to the current model's weaknesses. We then examine *self-improvement in agent data utilization* (Section 4.2), where curriculum design and adaptation improve the way produced data are selected, ordered, and allocated for agent model training. Finally, we consider *co-improvement across multiple modules in data production and utilization* (Section 4.3), where two or more modules are connected through feedback loops, enabling them to improve collaboratively.

4.1. Self-Improvement in Agent Data Production

As shown in Figure 5, agent data production includes the environment generation or simulation, task synthesis, trajectory synthesis, and verification of training and evaluation data. This subsection examines improvement within each individual module of the data production. A system may contain several interacting modules and still belong to this category under either of two conditions. The first is *single-module improvement*, where only one module improves substantially, while the remaining modules are fixed and serve as feedback sources or evaluation mechanisms. The second is *indepen-*

 doi:10.20944/preprints202608.0051.v1

dent improvement, where several modules improve, but their improvement processes are mutually independent with no causal influence between them.

4.1.1. Environment Generation or Simulation

An agent receives its feedback signals primarily from the environment it operates in. The fidelity, diversity, and difficulty distribution of training environments directly shape the agent's capabilities. Methods for preparing these environments can be divided into two classes. *Environment-generation methods* synthesize environments, including environment configurations, state spaces, and tool resources. *Environment-simulation methods* train a model to predict environmental dynamics and produce observation-action feedback. These two classes differ in mechanism but share a common goal: providing agents with feedback approximating actual operations in real-world environments.

Self-improvement in environment generation targets the validity and diversity of the produced environments. Compared with static, manually designed environments, a self-improving generator can adapt its outputs as the agent's capabilities grow [246]. It can also incorporate feedback to correct errors in previously generated configurations. With self-improvement, the generated environments remain aligned with the agent's current learning needs throughout training.

Environment-generation methods can be further organized by how they update the training distribution. The first class uses agent performance feedback to guide environment configuration generation. *EnvGen* [246] uses a fixed LLM to design environment configurations that target the weaknesses of an agent model trained by reinforcement learning. This process keeps the generated environments aligned with the agent's evolving capability boundary. *Adaptive Environment Generation* [247] extends the feedback-guided paradigm to spatial layout manipulation. It employs a fixed LLM as both a trajectory analyzer and an environment modifier. The analyzer extracts structured diagnostic feedback from the agent's navigation behavior, including failure modes and intermediate concerns, while the modifier adjusts individual object positions and orientations accordingly. This object-level adaptation uses collision-aware placement to produce physically valid layouts and targets specific spatial challenges without redesigning the entire environment.

The second class follows the unsupervised environment design paradigm, in which the curriculum is expanded through either search or distribution-based synthesis. *ACCEL* [248] represents the search-based approach. It applies small random mutations to existing environments and retains variants whose estimated regret exceeds a fixed threshold. The selected environments therefore remain challenging for the agent and can serve as starting points for further mutations. *DRED* [249] instead represents distribution-grounded synthesis. It uses a pre-trained variational autoencoder to generate environments that remain consistent with the target distribution, thereby reducing distributional shift and improving zero-shot generalization.

The third class improves the environment generator's capability through continual expansion of reusable tools and skills. *SimWorld Studio* [250] uses a tool-augmented coding agent to construct physically grounded 3D environments in Unreal Engine. When rule-based physics checks and vision-language model critiques identify recurring failures, the agent converts successful fixes into reusable tools and skills stored in its library. Agent performance feedback further guides difficulty adjustment, so that newly generated environments remain near the agent's capability boundary.

These generation approaches improve the validity and informativeness of the training environments through different operations. Feedback-guided generation targets observed weaknesses, mutation-based search expands the curriculum near the capability boundary, and distribution-grounded synthesis preserves task relevance during expansion. Additionally, self-improvement of the environment generator's harness allows the environment generator to internalize quality-control lessons and avoid previously encountered failure modes. The main challenge lies in maintaining environment validity at scale, because greater autonomy can increase the frequency of degenerate, unsolvable, or trivially easy configurations. Robust quality-control mechanisms are therefore necessary to keep the synthesized environments effective for agent training.

 doi:10.20944/preprints202608.0051.v1

For environment simulation, self-improvement aims to enhance the coverage and accuracy of the simulator's dynamics. As the agent explores new regions, a self-improving simulator can extend its coverage of the state-action space and progressively reduce the simulation-to-reality gap without manual recalibration. The main risk is that extrapolation beyond the simulator's training distribution may introduce drift into unrealistic dynamics. Such drift can lead to policies that exploit simulator artifacts rather than learn behavior that transfers to the target environment. Effective simulation improvement must therefore balance broader coverage with control of model drift. *DreamGym* [251] trains a reasoning-based experience model to predict environment dynamics in an abstract textual state space. Rather than replicating raw observations such as HTML or pixels, the model operates on structured state representations and uses chain-of-thought reasoning to predict subsequent states and rewards. An experience replay buffer seeded with trajectories reduces hallucination in state predictions, while a curriculum task generator selects tasks with high reward entropy to keep task difficulty near the agent's capability boundary.

4.1.2. Task and Trajectory Synthesis

Training and evaluation tasks translate capability objectives into specific units for learning and assessment. Therefore, the quality of these tasks is central to the performance of the overall system. Training tasks determine which capabilities the agent can acquire, whereas evaluation tasks reveal capability gains and weaknesses. An agent task typically comprises an instruction, an initial environment state produced by methods in Section 4.1.1, explicit constraints, available tools and resources, and verifiable evaluation criteria. Additionally, for supervised fine-tuning task data instances, corresponding trajectories synthesized from the task are also required. Task synthesis must coordinate these elements so that each instance is valid, diverse, and aligned with its intended training or evaluation purpose.

We first classify existing methods by their primary synthesis object. **Task synthesis** constructs task instructions, constraints, resource settings, and evaluation criteria, which define what the agent must solve. **Trajectory synthesis** builds solutions and interaction trajectories for a given task, which record the experiences of the agent. We categorize each system according to the object that plays a central role in its improvement loop.

Within **Task synthesis**, we further distinguish methods by the improvement signal used to adapt, select, or assess the synthesized tasks. The first class of task-synthesis methods uses *diagnostic performance feedback*. Its purpose is to identify capability gaps and generate tasks that target them. *WebRL* [252], for example, extracts failure patterns from agent experience and converts them into new training tasks. The task distribution changes with the agent's weaknesses, although the procedure that maps failures to tasks remains fixed. *CoEvolve* [253] extends this paradigm by extracting weakness signals from rollout trajectories. These signals guide a frozen LLM to re-explore the real environment along the identified failure dimensions, after which the resulting interactions are abstracted into validated task specifications and appended to the training set.

The second class uses *fixed quality-control signals* derived from rules, model agreement, or task structure. *SeRL* [254] generates instructions through few-shot prompting and selects them with fixed filters for similarity, content, length, and difficulty; majority voting provides an additional reward estimate. *Self-CriTeach* [255] uses a fixed prompting mechanism to synthesize symbolic planning domains. These domains define planning problems and supply structured criteria for downstream learning. In both systems, fixed quality-control signals shape the synthesized tasks, but the task-generation mechanism is not independently optimized.

The third class uses *competitive performance feedback*. In the frozen-opponent setting of *SAGE* [256], a Setter generates problems that it can solve but a fixed Opponent cannot. The competitive outcome therefore supplies a direct signal of task difficulty near the capability boundary. Only the Setter is trained in this setting, so the system does not establish co-improvement between the two roles.

In summary, these self-improving methods improve synthesized tasks along four main dimensions: relevance, validity, difficulty, and diversity. Diagnostic feedback increases relevance by directing

 doi:10.20944/preprints202608.0051.v1

generation toward observed capability gaps. Fixed quality-control signals improve validity and correctness by excluding inconsistent or poorly specified tasks. Similarity filtering also limits redundancy and helps preserve diversity. Competitive feedback adjusts difficulty toward the agent's current capability boundary, thereby keeping the generated tasks informative as the agent improves.

Trajectory synthesis methods generate agent learning experience for specified tasks. One group obtains such trajectories through competitive interaction. *SPIN* [257] contrasts outputs from the current model with those from a frozen previous iteration and fixed human references. *Arena Learning* [258] converts battles against fixed opponents into preference data using a fixed judge. The resulting trajectories become more challenging as the target model improves, but the interaction protocols and evaluation mechanisms remain fixed.

A second group generates trajectories through refinement or recovery. *EVOLVE* [259] turns initial and refined answers into preference data, whereas *RISE* [260] formulates iterative answer improvement as a multi-turn learning problem. *DNPO* [261] constructs preference pairs through dynamic labeling and controlled noise. *PLD* [262] instead collects recovery trajectories from specialized policies and distills them into a generalist model. Although these methods construct different forms of experience, they all apply a fixed trajectory-generation protocol around an improving model.

To sum up, these self-improving trajectory synthesis methods improve the usefulness of experience along several dimensions, including correctness, difficulty, and diversity. Competitive interaction produces more difficult and behaviorally diverse trajectories as the target model becomes stronger. Refinement-based methods improve the correctness and quality of candidate solutions, while recovery-based methods concentrate experience on observed failures. Preference construction and rubric-based assessment further increase the reliability of the supervision extracted from these trajectories.

4.1.3. Verification and Quality Assurance

Verification and quality assurance protect the data pipeline from errors that could otherwise accumulate across self-improvement iterations. *Verification* assesses whether an individual task, trajectory, label, or outcome satisfies a specified correctness or consistency criterion. *Quality assurance* has a broader scope. It filters, ranks, repairs, or annotates data according to validity, reliability, difficulty, and provenance. These functions are essential because self-generated data can amplify specification errors, incorrect labels, and biased judgments when used for training. Reliable verification and quality assurance preserve the integrity of both learning signals and evaluation results.

In this subsection, self-improving verification refers to cases in which the verifier, judge, or quality-control policy is itself updated across iterations. This boundary excludes fixed verification mechanisms. Executable checks [361], agreement-based filters [362,363], and frozen critic signals [364] can stabilize a self-improving data loop, but they do not constitute self-improving verification if their own decision rules remain fixed.

Self-improving verification means that improvement occurs within the verification module. *Self-Improving Vision-Language Model (VLM) Judges* [263] iteratively trains a VLM judge with self-synthesized multimodal data at different quality levels. This process forms a single-module self-improvement loop for verification because the judge itself is refined over successive iterations. *Escaping Model Collapse* [365] provides a complementary analysis of verifier-guided retraining. It shows that an external verifier can provide useful information and yield short-term gains. However, repeated optimization may eventually pull the model toward the verifier's own knowledge boundary. This result highlights a central risk for self-improving verification: if the quality signal is imperfect, accepted data can reinforce verifier biases and narrow the learning distribution.

In summary, self-improving verification improves the accuracy, coverage, and robustness of quality control. Its main value is to reduce incorrect acceptance or rejection, expand coverage of failure modes, and prevent noisy data from propagating through later training iterations.

 doi:10.20944/preprints202608.0051.v1

4.2. Self-Improvement in Agent Data Utilization

The previous subsection focused on self-improvement in agent data production, where environment, task, and trajectory generation expand the pool of available training and evaluation data. This subsection instead focuses on self-improvement in agent data utilization. Rather than generating new data, it improves how the produced data are used for model training through curriculum design and adaptation. *Curriculum design* determines how available training data are organized, typically by selecting and ordering environments, tasks, or trajectories according to the coverage of task categories, difficulty levels, or the model's capability gaps. *Curriculum adaptation* focuses on the curriculum adaptor that expands, updates, and refines the curriculum during training. Together, curriculum design and curriculum adaptation determine the training data recipe by adapting data utilization to the agent model's evolving capabilities.

Methods in self-improving curriculum adaptation can be organized into two levels. The first distinction is whether the curriculum adaptor is rule-based or learned. Rule-based methods can still be adaptive because they update the curriculum in response to the model trainer's feedback. Learned curriculum adaptation methods go further by optimizing the curriculum adaptor itself. Within learned curriculum adaptation, a second distinction is whether the curriculum adaptor is learned before deployment or updated online during agent training.

Rule-based Curriculum Adaptation.

These methods adapt the curriculum, but the adaptation rule itself is predefined. The changing state may be a difficulty parameter, a memory of past strategies, or a compact playbook of curated knowledge. Within this class, methods differ in what aspect of the training distribution they adapt. *TRUSTEE* [264] trains a tool-calling agent on trajectories from a fully simulated environment. It steers task difficulty with a scalar parameter that is updated by a predefined threshold rule based on batch reward. This parameter conditions the task generator along five dimensions: tool count, interaction turns, system-prompt specificity, user expertise, and evaluation strictness. As a result, the generated tasks can follow the current capability boundary of the agent. *EvoCurr* [265] follows the same general pattern. A prompted designer agent lowers task difficulty when the solver struggles and raises it when the solver succeeds. *Learning What to Learn* [266] focuses on content selection rather than difficulty control. It shows that a carefully selected subset of tasks can approach the performance obtained from the full dataset. The method uses frozen LLMs as a generator, a reflector, and a curator to distill a compact playbook.

Learned Curriculum Adaptation.

Within learned curriculum adaptation, the first subtype uses an *offline meta-learned curriculum adaptor*. The curriculum adaptation strategy is optimized during a meta-training stage and is then deployed as an adaptive strategy. *AMC-TSI* [267] jointly meta-learns a curriculum adaptor and a learning algorithm. It trains three neural networks: a difficulty estimator, a meta-policy that selects operators and allocates compute, and a process reward model. After meta-training, the learned strategy adjusts its test-time decisions to observed learning dynamics. The key difference from rule-based curriculum adaptation is the source of the adaptation strategy. The curriculum adaptor is learned before deployment rather than manually predefined. The second learned subtype uses an *online learned curriculum adaptor*. In this subtype, the selector is itself a learning module inside the current training loop. *Actor-Curator* [268] trains a neural curator to select problems that maximize expected policy improvement. The actor is updated through reinforcement learning, and changes in actor performance provide feedback for refining the curator. The curriculum therefore moves with the actor's capability boundary. Meanwhile, the curriculum adaptor continues to improve during agent training through its own learning objective.

In summary, these methods improve the data curriculum for agent model training. They make the data curriculum more relevant to the current agent model's weaknesses and adjust task difficulty

 doi:10.20944/preprints202608.0051.v1

as the agent model improves. Curriculum adaptation increases data and computational efficiency, while selective coverage reduces redundancy without sacrificing diversity. These objectives must be balanced to avoid excessive focus on recent failures and the loss of previously acquired capabilities.

4.3. Multi-Module Co-Improvement in Agent Data System

The previous subsections 4.1 and 4.2 examine cases in which each module is optimized independently. This subsection focuses on the collaborative improvement of two or more interdependent data system modules within a feedback loop, as shown in Figure 6. In other words, each module both shapes and benefits from the improvement of another module, making their improvements mutually dependent.

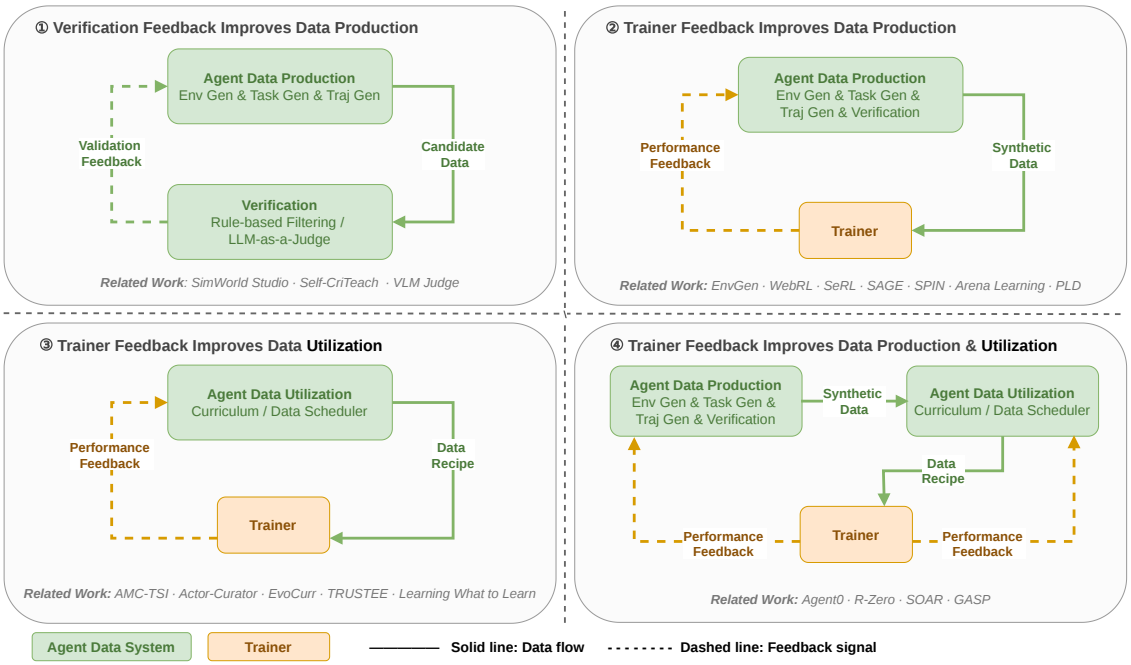

Figure 6. Self-improvement of data system modules in feedback loops.

Collaborative improvement addresses a key limitation of optimizing each module independently. As the agent model's capabilities evolve, a data system module that remains fixed can become a bottleneck, preventing the rest of the data system from adapting effectively. Collaborative improvement allows every data system module in the feedback loop to adapt to the agent model's evolving capabilities, ensuring that training data remain appropriately challenging, experience coverage continues to expand, and low-quality data are progressively filtered out.

We organize the following discussion according to where the feedback loop is established within the agent data system. *Co-Improvement within Data Production* (Section 4.3.1) covers feedback loops among data-production modules, including environment generation or simulation, task and trajectory synthesis, and verification. In these systems, the outputs of one production module provide feedback for improving another, thereby enhancing the quality and informativeness of the generated data for subsequent agent learning. *Co-Improvement of Data Production and Utilization* (Section 4.3.2) extends these feedback loops to connect data production with data utilization, completing a closed-loop feedback cycle among data production, data utilization, and model training.

4.3.1. Co-Improvement within Data Production

Multiple modules within data production can interact through a feedback loop, where the improvement of one module provides signals or resources that facilitate the improvement of others. This subsubsection reviews co-improvement methods within the data production stage and categorizes existing approaches based on the production modules that are jointly improved. The first category

 doi:10.20944/preprints202608.0051.v1

characterizes collaborative improvement between synthesis and verification, where generated solutions expose limitations in the assessment mechanism and the improved verifier provides more informative feedback for subsequent synthesis. The second category examines collaborative improvement between environment construction and task synthesis, where agent failures guide the generation of new tasks and the refinement of interactive environments.

Co-Improvement of Synthesis and Verification.

The coupled data synthesis and verification modules within the data system can jointly improve through their interactions. The agent model produces new behaviors that expose a changing error distribution. The assessment mechanism refines its judgments in response, and the improved mechanism then supplies more informative learning signals back to the agent model. The key property is that the assessment mechanism has its own learning objective and is not merely a frozen judge.

Agent0-VL [269] realizes this coupling within a shared vision-language model. Its solver produces increasingly challenging tool-integrated reasoning traces, while its verifier uses tool-grounded critique to provide fine-grained rewards. The two roles improve together even though they share parameters. *ACE* [270] uses a single backbone model that alternates between the roles of code solver and adversarial test generator. Unlike ground-truth tests, which contain both inputs and expected outputs, the adversarial unit tests generated by ACE contain only test inputs and are used to expose execution-level failures such as corner cases, boundary violations, and runtime fragility. Across training rounds, candidate programs that pass both ground-truth tests and adversarial tests are selected for supervised fine-tuning of the solver. In addition, adversarial tests are labeled according to their execution effects: tests that distinguish robust programs from fragile ones are treated as desirable, whereas tests that fail to reveal useful differences are treated as undesirable. These labels are then used as preference signals to optimize the adversarial test generator through Kahneman-Tversky Optimization.

These methods share a common structure: the agent model reveals limitations in the current assessment, while the refined assessment provides stronger signals for further learning. By coupling these processes, the system postpones the saturation of a static verifier and maintains supervision that tracks the agent model's shifting error distribution. The task and environment sources remain unchanged; improvement occurs specifically at the evaluation interface.

Co-Improvement of Environment and Task Synthesis.

Agent-World [271] jointly improves the agent policy, training tasks, and interactive environments. It builds a large collection of tool-using environments from real-world databases and web services. After each training round, the updated policy is evaluated in these environments. A diagnosis agent analyzes failure traces to identify weak environments and produces guidelines for task generation. These guidelines steer subsequent task synthesis and increase database complexity within the identified weak environments, so that new tasks and more difficult environments are produced in response to the policy's observed weaknesses. The newly generated tasks and expanded environments are then used by the reinforcement-learning trainer to update the policy, forming a feedback loop among policy learning, task generation, and environment refinement under a fixed improvement rule.

Across these systems, co-improvement within data production makes the generated data increasingly aligned with the agent's evolving training needs. Because production modules exchange feedback with each other, errors observed in trajectories, environments, or verification results can be converted into signals to improve subsequent data generation. These feedback loops help maintain an appropriate level of difficulty, expand coverage over experience and failure modes that the agent has not yet mastered, fix errors in data production modules, and reduce the propagation of low-quality data. As a result, the data production stage can provide more valid, diverse, and informative data without relying on repeated human intervention.

 doi:10.20944/preprints202608.0051.v1

4.3.2. Co-Improvement of Data Production and Utilization

This subsubsection focuses on closed-loop systems in which data production, data utilization, and model training mutually influence each other through continuous feedback. The data production modules synthesize tasks, environments, and trajectories. These data are then selected and incorporated into the model training by the data utilization modules. In turn, the resulting training outcomes provide feedback signals that guide the collaborative improvement of subsequent data production and utilization.

We classify related work according to the mechanisms through which data production and data utilization modules collaboratively improve. *Capability boundary matching* leverages the solver agent's signals of success, failure, or uncertainty to guide data production and utilization modules in generating and selecting tasks near the current capability boundary of the solver agent. *Goal-directed curriculum bridging* uses progress on target tasks to guide the generation and selection of intermediate training data that help the agent reach externally specified goals.

Capability Boundary Matching.

Capability frontier matching uses the solver agent's current capability boundary to determine which data instances should be generated and selected for the next training round. Before training, data production modules generate candidate tasks, and data utilization modules perform multiple rollouts on these candidate tasks. The resulting rollout signals, such as correctness, uncertainty, or reward, provide feedback on whether tasks have appropriate difficulty near the solver's capability boundary. These signals then guide the improvement of both the data production and data utilization modules.

Agent0 [272] trains a curriculum agent to generate increasingly difficult tasks, while an executor agent improves its tool-integrated reasoning on those tasks. The curriculum agent receives rewards according to whether its generated tasks are near the executor's current capability boundary. Thus, utilization outcomes from executor training directly improve the next round of task production. *R-Zero* [273] implements the same principle: a challenger is rewarded for proposing problems near the solver's current boundary, and the solver is trained on the resulting curriculum without pre-existing tasks or labels. In both systems, task production and task utilization are coupled through the agent model's changing performance profile: harder and better calibrated tasks improve the solver, and the improved solver reshapes the reward landscape for subsequent task generation.

Goal-directed Curriculum Bridging.

Goal-directed curriculum bridging uses target tasks to determine which intermediate data should be generated and selected for training. After the model is trained on the generated intermediate tasks, progress on the target tasks indicates whether these data provide useful stepping stones toward the target capability. This progress signal then guides subsequent data production and selection toward curricula that better bridge the gap between the model's current abilities and the target goal. *SOAR* [274] trains a teacher to generate synthetic stepping-stone problems and rewards it according to measurable student improvement on hard target problems. This links curriculum quality to learning progress instead of a proxy for difficulty alone. *Guided Asymmetric Self-Play (GASP)* [275] uses hard coding problems as goalposts. Its teacher generates easier lemmas and progressively harder lifts that move the student toward those goalposts, while execution feedback trains the student. In both systems, external targets provide relevance, but the learned teacher determines which intermediate data should be produced and scheduled for training. The loop therefore improves data production by measuring downstream utilization value, and improves utilization by supplying curricula that are increasingly aligned with target-task progress.

In summary, these methods share the same central mechanism, in which model training outcomes are transformed into signals for both data production and curriculum adaptation. The improved data production and utilization then jointly reshape the training distribution for the next iteration.

 doi:10.20944/preprints202608.0051.v1

Capability-frontier matching emphasizes calibrating task difficulty to the solver's current capability boundary, whereas goal-directed curriculum bridging emphasizes designing intermediate training data that connect current capabilities with target capabilities. Together, these patterns show how data production and data utilization can collaboratively improve within a feedback loop while keeping the generated data adaptive, informative, and aligned with model training needs.

Table 3. Representative self-improvement methods in the agent data system.

Work	Self-Improvement Process		Domain	Level
	Object	Evidence		
Data Production				
EnvGen [246]	Environment generation	Success rate	Embodied agent	L3
Adaptive Env Gen [247]	Environment generation	LLM analysis + Physical check	Embodied agent	L3
ACCEL [248]	Environment generation	Positive value loss	Procedural RL	L3
DRED [249]	Environment generation	Value loss scoring	Procedural RL	L3
SimWorld Studio [250]	Environment generation	Rule + VLM	Embodied agent	L3
WebRL [252]	Task synthesis	Critic score	Web agent	L3
SeRL [254]	Task synthesis	Majority-voting + Difficulty	Math reasoning	L3
Self-CriTeach [255]	Task spec + Trajectory	Planner check + Reward	Robotic planning	L3
SAGE [256]	Task synthesis	Code execution / LLM judge	Reasoning	L3
CoEvolve [253]	Task synthesis	Env. execution + Binary reward	Tool-use agent	L3
SPIN [257]	Trajectory synthesis	Self-play	General	L3
Arena Learning [258]	Trajectory synthesis	LLM judge	Dialogue	L3
EVOLVE [259]	Trajectory synthesis	RM	General	L3
DNPO [261]	Trajectory synthesis	LLM judge	General	L3
PLD [262]	Trajectory synthesis	Binary reward + Success rate	Embodied agent	L3
SI VLM Judges [263]	Judge model	Accuracy	Multimodal RM	L3
Data Utilization				
AMC-TSI [267]	Curriculum	Accuracy + Transfer regret	Math reasoning	L3
EvoCurr [265]	Curriculum + Memory	Rollout win rate	Game	L3
TRUSTEE [264]	Curriculum	Eval pass rate	Tool-use agent	L3
Actor-Curator [268]	Curriculum	Policy improvement	Math reasoning	L3
Co-Improvement				
Agent0-VL [269]	Verifier + Trajectory	Self-verification	VL reasoning	L3
ACE [270]	Adversary + Trajectory + Unit tests	Adversarial testing	Code generation	L3
Agent-World [271]	Synthesized env/task	Rubrics/Validators	Tool-use agent	L3
Agent0 [272]	Curriculum model + Tasks + Trajectory	Self-consistency + Majority vote	Reasoning	L3
R-Zero [273]	Challenger model + Tasks + Trajectory	Majority-vote + Uncertainty	Reasoning	L3

5. Agent Trainer Self-Improvement

Building on the agent-system representation in Section 2, this section studies the trainer $\mathcal{T}$. The trainer governs how training data are turned into model updates. It consists of three internal components. The supervision design derives learning signals from the available training evidence. The optimization strategy determines how those signals change the model parameters. The training infrastructure executes and coordinates the resulting updates.

We define trainer self-improvement as an evidence-driven revision of persistent state that determines how the trainer operates. Evidence produced during training or evaluation must induce a persistent change to either the trainer or, at the meta level, the improvement mechanism that revises it. The changed state must be retained and used in subsequent training or later trainer-improvement cycles.

Figure 7 places this definition within the overall trainer framework. The left side shows a training experiment, in which the supervision design, optimization strategy, and training infrastructure work together to update the model. The right side abstracts the improvement process into four functional roles. *Diagnose* examines training and evaluation evidence to identify system bottlenecks, failure modes, or opportunities for improvement. *Propose* converts the diagnosis into concrete and runnable modifications to the trainer. *Evaluate* measures the effects of these candidates through experiments or controlled comparisons and determines which changes have adequate empirical support. *Integrate* commits an accepted modification to persistent state so that it governs subsequent training or trainer

 doi:10.20944/preprints202608.0051.v1

improvement. These roles may be implemented by fixed rules, learned controllers, or explicit agents. A system may combine several roles within the same procedure or component.

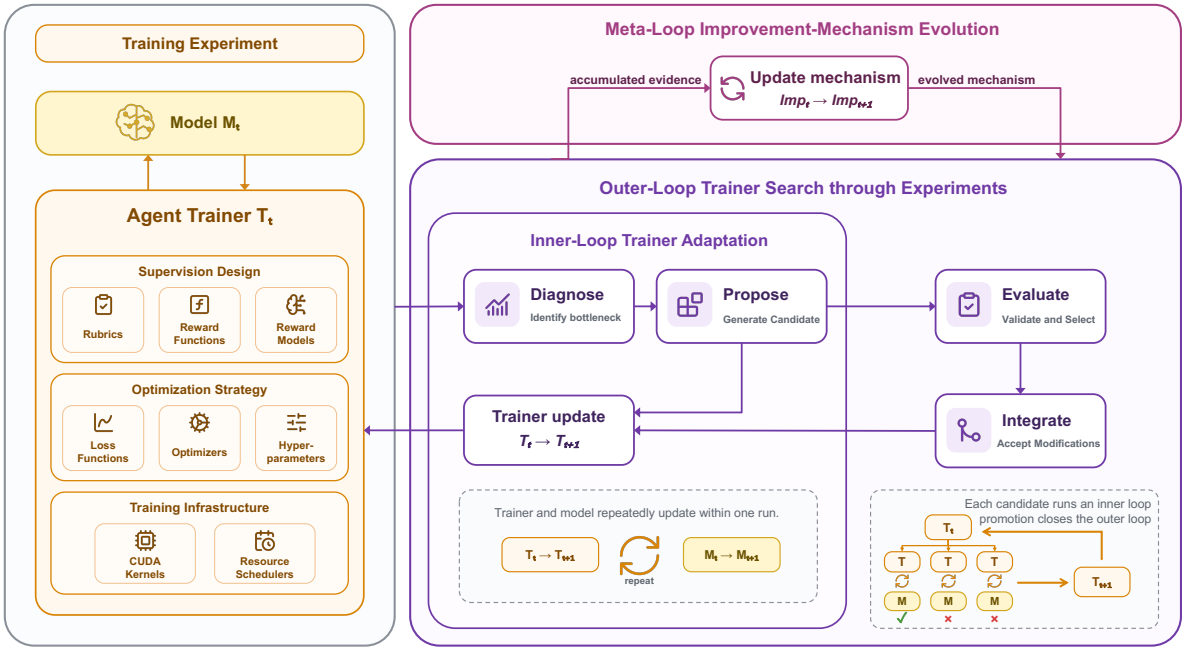

Figure 7. Agent trainer and the three loops of trainer self-improvement. The loops distinguish processes that adapt the active trainer within a training run, select and retain a trainer version through experiments, or revise the improvement mechanism that governs later experiments.

We organize the remainder of this section around three loops of trainer self-improvement, summarized in Table 4. The three loops are:

- *Inner-Loop Trainer Adaptation* operates within an active training lineage. Evidence from the current run directly changes persistent state in the supervision design, optimization strategy, or training infrastructure. The revised trainer state then governs subsequent model updates in the same lineage. The improvement mechanism remains fixed.
- *Outer-Loop Trainer Search through Experiments* operates across training experiments. Within each experiment, a trainer and a model interact through a training loop; across experiments, a fixed improvement mechanism uses the resulting evidence to propose runnable trainer candidates and evaluate them in bounded experiments. Based on the results, it rejects a candidate or integrates it as the retained trainer version for later experiments.
- *Meta-Loop Improvement-Mechanism Evolution* operates across successive trainer-improvement cycles. Within each cycle, the current improvement mechanism governs diagnosis, candidate proposal, evaluation, and integration. Across cycles, evidence from model-training experiments updates the persistent mechanism from Imp_t to Imp_{t+1}. The successor mechanism then governs these processes in later trainer-improvement cycles.

Table 4. Representative Trainer self-improvement methods across three loops, summarized by target object, evidence, revision mechanism, and RSI level.

Work	Self-Improvement Process			Level
	Object	Evidence	Mechanism	
Inner-Loop Trainer Adaptation				
ACE [270]	Test generator	Code–test execution matrix	Preference optimization	L3
AdaLRS [312]	Learning rate	Loss-descent velocity	Probe with retain/rollback	L3
ADMIRE [279]	Milestone criteria	Successful trajectories	Milestone distillation	L3
ASL [294]	Reward model	Rule-verified outcomes	Alternating role RL	L3
AI Training Manager [311]	Training recipe	Training telemetry	Schema-bounded LLM edits	L3

 doi:10.20944/preprints202608.0051.v1

Work	Self-Improvement Process			Level
	Object	Evidence	Mechanism	
ARBOR [278]	Rubric memory	Trajectory contrasts	Rubric write–prune	L3
ARCO [285]	Rubric generator	Binary terminal outcomes	Joint rubric–policy RL	L3
ATLAS [310]	DPO coef.; reference anchor	Telemetry; reference KL	Bounded update; anchor swap	L3
CoNL [295]	Critic	Critique-induced gains	Segment-rewarded RL	L3
COPUS [314]	Batch size; parallelism	Gradient noise; throughput	Goodput-driven resharding	L3
CoVerRL [296]	Verifier	Consensus pseudo-labels	Alternating role RL	L3
DR Tulu [276]	Rubric memory	Trajectory contrasts	Rubric write–prune	L3
DYNAMIX [319]	Batch-size controller	Training + system metrics	PPO controller training	L3
EARL [321]	Parallelism config.	Context length; system load	Profile lookup; live switch	L3
ECHO [291]	Critic	Critique-induced gains	Gain-rewarded RL	L3
EvoLM [282]	Rubric generator	Temporal contrasts; margins	Alternating role RL	L3
EPI [313]	Parameter mask	Per-param squared gradients	Periodic mask refresh	L3
EvoRubric [284]	Rubric generator; archive	Rubric validity; consensus	Role RL; rubric archiving	L3
EvoRubrics [281]	Rubric generator	Cross-evaluation scores	Adversarial GRPO update	L3
ICRL [292]	Critic	Critique-induced gains	Alternating role RL	L3
JarvisEvo [290]	Reward model	Human target assessments	Alternating role RL	L3
KungFu [315]	Batch size; workers; topology	Gradient noise; throughput	Rule-based reconfiguration	L3
MagicGUI-RMS [289]	Reward model	Reward-model disagreements	Data write-back; RM retraining	L3
Mutual-Taught [288]	Reward model	Pre/post-update preferences	Alternating DPO updates	L3
ONES [318]	Batch size; GPU allocation	Epoch progress; cluster state	Evolutionary scheduling	L3
Optimus [322]	Resource allocation	Loss history; throughput	Model refit; reallocation	L3
Pollux [316]	Batch size; GPU allocation	Gradient noise; throughput	Goodput co-optimization	L3
Q-Evolve [301]	Value critic	Transitions; terminal reward	IQL critic training	L3
RFT-FM [320]	Training configuration	Telemetry fault fingerprints	Diagnosis-guided repair	L3
RLAC [293]	Critic	Validator outcomes	Alternating DPO updates	L3
RLAnything [300]	Process reward model	Terminal outcomes; consistency	Alternating RM–policy RL	L3
RLAR [286]	Reward-tool library	Tool verification results	Synthesize, verify, commit	L3
RLCER [283]	Rubric generator	Rubric–answer correlation	Correlation-rewarded RL	L3
RL Tango [299]	Verifier	Final-answer correctness	Interleaved GRPO updates	L3
rStar-Math [302]	Process preference model	MCTS values; code execution	Pairwise preference training	L3
Rubick [323]	Execution plan	Measured throughput	Model refit; reconfiguration	L3
RubricEM [277]	Rubric memory; meta-policy	Trajectory contrasts	Write–prune; reflection RL	L3
SAVE [287]	Reward model	On-policy reward–value gaps	Value-anchored self-training	L3
SCORE [297]	Evaluator	Cross-rollout consistency	Consistency-rewarded RL	L3
Sia [317]	Batch size; GPU allocation	Gradient noise; cluster load	Goodput co-optimization	L3
Skill-SD [308]	Teacher state; skill bank	Trajectory rewards; utility	Skill writing; teacher sync	L3
SPARD [306]	Reward weights	Reward progress; dispersion	Mirror-descent reweighting	L3
StepORLM [303]	Process reward model	Solver outcomes; critiques	Iterative RM fine-tuning	L3
STRIDE [304]	Verifier	Final-answer correctness	Verifier fine-tuning	L3
Evaluative Thinking [280]	Evaluation rubric	Responses; RM scores	Meta-reward-model rewrite	L3
UCOB [309]	Teacher-side skill memory	Return gaps; skill utility	Credit-aware skill write–prune	L3
UI-Genie [305]	Reward model	Outcome/continuation labels	Iterative RM fine-tuning	L3
URPO [298]	Reward model	Preference rankings	Rank-rewarded GRPO update	L3
ZeroCoder [307]	Test generator; selector prior	Code–test execution matrix	Role RL; prior recalibration	L3
Outer-Loop Trainer Search through Experiments				
AIDE [353]	Training script	Validation score; run errors	LLM tree search	L3
AIRA-Design [354]	Training script	Validation loss; executability	LLM tree search	L3
Auto Research [328]	Training script	Eval scores; crashes; runtime	LLM code edits; promotion	L3
AutoTrainess [326]	Fine-tuning recipe	Benchmark scores	LLM recipe assembly	L3
CARD [335]	Reward code	Trajectory preferences	LLM code edits; gated trials	L3
CliffSearch [329]	Optimizer code	Validation loss; reviews	Reviewer-gated evolution	L3
DiscoPOP [348]	Loss function	Post-training scores	LLM generation; selection	L3
LLM-RL HPO [347]	Hyperparameters	Reward/KL dynamics; cost	Multi-fidelity Bayesian opt.	L3
Eureka [334]	Reward code	Fitness; component traces	Evolution; elite retention	L3
FT-Dojo [327]	Fine-tuning recipe	Validation errors; loss curves	LLM proposals; gated trials	L3
Highway Reward Evolution [351]	Reward code	Driving metrics	Evolution with RL trials	L3
LaRes [340]	Reward code	Task success; learning curves	Evolution; elite retention	L3
LERO [352]	Reward code	Team returns; convergence	Evolution with RL trials	L3
LLM-ALSO [336]	Reward-shaping config.	Returns; branch stability	Fork-validated promotion	L3
LLMZero [324]	Training configuration	Validation telemetry	LLM tree search	L3
MLEvolve [338]	Training pipeline	Task metric; run failures	Graph search with memory	L3
OptiCo [330]	Distributed-training config.	Throughput; runtime failures	Multi-agent search; memory	L3
POISE [333]	Policy-optimization code	Dynamics; held-out scores	Archive-guided LLM edits	L3
R* [341]	Reward code; coefficients	Fitness; component traces	Evolution with RL trials	L3
REvolve [337]	Reward code	Human rollout preferences	Island evolution; migration	L3
RF-Agent [342]	Reward code	Return; component feedback	LLM tree search	L3

 doi:10.20944/preprints202608.0051.v1

Work	Self-Improvement Process			Level
	Object	Evidence	Mechanism	
RHyVE [346]	Reward schedule	Fork margins; agreement	Fork-validated promotion	L3
ROSKA [343]	Reward code; fusion ratio	Fitness; component traces	Evolution; tuned policy fusion	L3
Reward Synthesis [350]	Reward code	Executability; task F1	Ranked generation; ensemble	L3
Self-Evolving Recommender [339]	Training + reward code	Offline loss; A/B metrics	LLM code edits; staged gates	L3
SMCEvolve [356]	Training script	Validation loss; run failures	SMC resampling; mutation	L3
TREX [332]	Fine-tuning recipe	Validation score; run failures	LLM tree search	L3
Meta-Loop Improvement-Mechanism Evolution				
AutoScientists [331]	Role-allocation policy	Validation deltas; stagnation	Evidence-triggered reorg.	L4
Bilevel Autoresearch [238]	Search runner code	Proposal–outcome traces	LLM-generated replacement	L4
DERL [349]	Reward meta-optimizer	Inner-policy validation	GRPO meta-training	L4
EvoTrainer [325]	Diagnostic harness	Diagnostic gaps; branch scores	Retained harness revision	L4
GEAR [355]	Search controller code	Crossover failures	Source-level repair	L4
Execution-Grounded AI Research [357]	Ideation policy	Validation score; run status	Execution-rewarded RL	L4

5.1. Inner-Loop Trainer Adaptation

Inner-Loop Trainer Adaptation occupies the innermost loop in Figure 7. Evidence from the active training process changes persistent trainer state from $\mathcal{T}_t$ to $\mathcal{T}_{t+1}$. The updated state then controls subsequent model updates in the same training lineage. The current training process continues without first evaluating the proposed change as a separate trainer candidate.

We group these methods by the part of the trainer that changes. *Supervision Design Evolution* revises the labels, rewards, critiques, or teacher targets used to train the model. *Optimization Strategy Evolution* revises the objective, update algorithm, or training controls that convert supervision into parameter updates. *Training Infrastructure Evolution* revises the execution substrate that carries out those updates, including low-level CUDA kernels, distributed-training runtimes, parallel execution plans, and resource-scheduling state. In all three groups, evidence directly changes the active trainer within the ongoing training lineage, while the improvement mechanism remains fixed.

5.1.1. Supervision Design Evolution

Supervision Design Evolution changes the persistent trainer-side state that determines how experience becomes supervision for the model. This state may define what behavior should count as successful, implement the mechanism that judges observed behavior, determine where a judgment should be assigned within a trajectory, or control how the resulting signal is delivered to and absorbed by the model. A method belongs to this category only when such a change is retained and affects later training. Producing different scores, critiques, or advantages for different samples with an otherwise fixed supervision pipeline is not sufficient. A supervisor may share parameters with the policy, but its role must receive an identifiable training signal, expose a distinct supervisory output, and use the updated function to alter later model training.

We organize this pipeline around four successive questions. *Evaluation Criteria Evolution* asks what should count as good behavior. *Evaluator and Verifier Evolution* asks how observed behavior is converted into a judgment. *Process-Level Credit Assignment Evolution* asks where within a multi-step process that judgment should be assigned. *Training-Signal Routing and Internalization* asks how the resulting supervision reaches the model and becomes available without its original supervisory support.

Evaluation Criteria Evolution

Evaluation Criteria Evolution changes the standards by which behavior is judged. It updates either the criteria themselves or the mechanism that generates them, allowing supervision to remain informative as the model's behavior evolves. Its defining effect is semantic: after the update, later behavior is evaluated against a revised conception of quality, correctness, progress, or failure.

 doi:10.20944/preprints202608.0051.v1

What changes is the standard itself, not how a fixed standard is applied: updating a judge or verifier without revising the meaning of success is a matter for the next category. Criteria evolution therefore answers what should be rewarded, not yet whether a particular behavior satisfies the standard or which part of that behavior deserves credit.

One route treats criteria as editable memory rather than model parameters. DR Tulu contrasts on-policy research responses to add positive and negative criteria, then retains the items whose rewards remain discriminative for later GRPO batches [276]. RubricEM applies a related write–score–prune cycle to stage-local rubric buffers, while ARBOR adds cross-query consolidation, reuse statistics, and retirement to maintain a longer-lived process-rubric memory [277,278].

Criteria can also encode progress through a trajectory. Adaptive Milestone Reward distills only a superior successful GUI trajectory into a revised per-instruction milestone sequence; a fixed matching rule then turns progress through those milestones into dense rewards for later successful and failed rollouts [279]. Toward Evaluative Thinking instead rewrites the textual rubric used by a reward model: periodic meta-analysis of recent responses and scores revises its criteria, examples, and scoring ranges before the next policy batches are evaluated [280]. In both cases, the semantic specification changes while the mechanism that applies it remains fixed.

A second route trains the producer of criteria. EvoRubrics gives the rubric generator a dedicated adapter and updates it from cross-evaluation evidence, so the successor adapter generates the criteria used to reward later policy batches [281]. EvoLM trains a shared-backbone rubric role from temporal response contrasts and frozen-judge score margins, whereas RLCER gives its shared-parameter rubricator a separate reward based on rubric validity and discrimination against answer correctness [282,283]. These systems share a criterion-generation interface, although their parameter sharing and external anchors differ.

Generator evolution can be coupled to additional persistent state. EvoRubric verifies response-conditioned atomic criteria, removes uninformative items, and archives accepted rubrics for later generation and reward construction [284]. ARCO jointly updates a per-step criterion generator and a criterion-conditioned scorer from terminal outcomes; the successor rubric model then supplies new step-specific criteria and reward-to-go signals to later policy batches [285]. Its scoring head participates in the loop, but the adaptive supervision first enters through the generated criteria.

Evaluator and Verifier Evolution

Evaluator and Verifier Evolution changes the mechanism that converts behavior and available evidence into an evaluative judgment. Evaluators may produce learned assessments, preferences, critiques, or scalar rewards. Verifiers may instead ground judgments in execution results, environment feedback, formal checks, or reusable reward tools. In either case, the evolving object is a retained model, program, tool collection, test generator, or verification state that changes how subsequent behavior is judged.

What evolves is the judge, not the standard: when the meaning of success itself changes, the method belongs to *Evaluation Criteria Evolution*.

Executable evidence can update verifier state without learning a scalar reward model. ACE trains an adversarial unit-test generator from a code–test execution matrix; the retained adversary adapter produces later tests that determine which solver programs enter supervised fine-tuning [270]. RLAR instead detects missing reward coverage, synthesizes or retrieves a callable checker or reward-model wrapper, verifies it, and commits the accepted tool to a library used on later training samples [286]. The first mechanism evolves a test generator, whereas the second evolves the available verification toolkit.

Separate reward models can follow the changing policy distribution directly. SAVE updates a persistent reward-and-value model from value-anchored on-policy feedback before using the successor model to score later response groups [287]. Mutual-Taught constructs pseudo-preferences between responses sampled before and after a policy update, MagicGUI-RMS refluxes disagreements with a fixed general-purpose verifier into a domain reward model, and JarvisEvo aligns an evaluator role to

 doi:10.20944/preprints202608.0051.v1

human target assessments before reusing it as the editor's reward source [288–290]. Their evidence sources differ, but each closes the same write-back path from current policy behavior to a successor evaluator and then to later policy supervision.

Other critics learn from the downstream usefulness of their judgments. ECHO rewards a separate critic by the improvement obtained when the policy follows its critique [291]. ICRL uses the same causal test with a shared backbone: revision success supplies a role-specific critic reward, and the updated critic conditions later training trajectories while the token-calibration rule remains fixed [292]. RLAC grounds an adversarial critic in fixed fact-checking or execution outcomes, training it to surface rubrics that the current generator is likely to violate and thereby changing where later verification is applied [293].

Several systems encode an explicit supervisor role in the same checkpoint as the policy. Agentic Self-Learning trains a generative reward-model role from verifiable labels, CoNL assigns distinct rewards to critique and ranking segments, and CoVerRL explicitly trains a verifier role that filters pseudo-labels for later updates [294–296]. Self-Evolving Deep Research updates an evaluator role from report-consistency evidence, changing the rubric dimensions and scores supplied to subsequent solver optimization [297]. URPO is the limiting shared-state case: a Kendall-τ reward trains an identifiable referee interface from ranked preferences, and the same updated model later scores open-ended response groups [298]. In these methods, the evolving object is the trained supervision function and its downstream interface, not generic improvement of the shared policy.

Process-Level Credit Assignment Evolution

Process-Level Credit Assignment Evolution changes how outcome-level or trajectory-level evidence is attributed to intermediate decisions. It converts an available judgment into supervision for steps, actions, stages, turns, or tokens, thereby identifying which parts of a process should be reinforced or corrected. The evolving object may be an attribution rule, a process evaluator, a decomposition mechanism, or another retained structure that improves the localization of credit under later model behavior.

The defining question is where evidence should be assigned, not whether the overall behavior is successful: a mechanism belongs here when its main evolving output is the placement of credit over a process, even when a learned model implements that mechanism. A system whose primary change instead improves the underlying correctness or quality judgment belongs to *Evaluator and Verifier Evolution*.

Outcome-grounded process verifiers learn to replace a terminal verdict with denser judgments. RL Tango alternates a generator with a separately trained generative verifier: known final-answer correctness updates the verifier, whose successor step judgments become rewards for later generator updates [299]. RLAnything similarly trains a generative process reward model from terminal outcomes and self-consistency evidence, then combines its revised step evaluations with outcome feedback in subsequent policy updates [300].

Credit can also be represented through values or process preferences. Q-Evolve trains a persistent Q/V critic on hybrid trajectories and converts its value estimates through generalized advantage estimation into step-wise supervision for the next policy phase [301]. rStar-Math derives correct–incorrect step pairs from verified search trajectories to train a process-preference model, whose successor scores guide later MCTS nodes and policy-training targets [302]. StepORLM refines a generative process reward model from outcome-labeled trajectories; its revised scores determine both the preference pairs and pair weights used by the next policy update [303].

Explicit localization methods identify where a trajectory can still be corrected. STRIDE trains a verifier from final correctness, uses its language feedback to identify the first erroneous step, and constructs redirected prefixes for subsequent generator training [304]. UI-Genie instead launches continuation rollouts from intermediate GUI states to label whether completion remains possible; the resulting process labels update its reward model, whose successor judgments guide later trajectory selection and supervision [305].

 doi:10.20944/preprints202608.0051.v1

Training-Signal Routing and Internalization

Training-Signal Routing and Internalization changes how already available supervision is selected, weighted, transferred, or made usable by the model. Routing determines which supervision channels, examples, teacher views, critiques, or reward components reach a learning decision and with what relative influence. Internalization transfers information available through a teacher, critic, privileged view, or temporary scaffold into a model that can later act without that auxiliary source. This category begins where the previous three end: the standard, the judgment, and its placement are already fixed, and what changes is how the resulting signal reaches the model and outlives its original source.

Routing state determines which already defined signals dominate the next update. SPARD maintains moving estimates of progress and dispersion for fixed reward dimensions, then updates their scalarization weights so later responses receive a different reward mixture [306]. ZeroCoder's Dynamic B4 mechanism recalibrates Bayesian selector priors from current-policy execution matrices and a small labeled set; the active priors then change which consensus outputs become coder and tester rewards in the next update [307]. The former continuously reweights reward channels, while the latter changes which proxy labels are admitted as supervision.

Internalization methods change the privileged teacher state from which training targets are drawn. Skill-SD writes successful strategies, mistakes, and workflows into a utility-tracked skill bank; a selected skill conditions a refreshed teacher whose token distribution supervises the unaugmented student [308]. UCOB maintains task- and state-level skill memories, compares skilled and unskilled returns at matched states, and distills from the locally stronger view while updating each skill's future retrieval utility [309]. In both systems, the persistent skill-and-teacher state changes later training targets; an inference-only memory would leave those targets unchanged.

Together, these four interfaces trace the supervision path from defining a standard, through producing and localizing a judgment, to delivering that judgment to the model. The next subsection turns from the supervision path to the mechanisms that use its output to form parameter updates.

5.1.2. Optimization Strategy Evolution

Optimization-strategy evolution begins after supervision has been produced. It changes persistent trainer state that determines either what objective is optimized or how that objective becomes a parameter update. We therefore separate two interfaces: *Objective Adaptation* changes the mathematical target assembled from existing supervision, whereas *Parameter-Update Adaptation* changes the controls that realize this target as parameter changes. In both cases, evidence from the active training process is written into an explicit objective or update-control state, and the revised state governs later updates in the same model lineage. Ordinary optimizer moments and schedules fixed by step count remain part of ordinary training because their transitions are already specified by a fixed update rule.

Objective Adaptation

Objective adaptation changes how available supervision is combined, normalized, or anchored before an optimizer consumes it. One route revises the relative influence of existing signals. SPARD tracks progress and dispersion for fixed reward dimensions and updates a persistent scalarization vector, so subsequent GRPO steps optimize a different reward mixture [306]. RL Tango maintains separate exponential-moving-average statistics for correct and incorrect generator outcomes; these statistics rescale the advantages used in later verifier updates and thereby rebalance the two outcome classes [299]. The two methods adapt objective weights at different resolutions: SPARD operates across reward dimensions, while RL Tango operates across outcome classes. Their separate data-routing and verifier-learning mechanisms belong to Supervision Design Evolution.

Reference-based objectives expose a second control surface. In ATLAS v3, phase telemetry updates the DPO coefficient β_{DPO}, and a fixed utility–proxy-KL gate decides whether a policy checkpoint should become the reference for the next EvoDPO phase [310]. The coefficient controls the strength of reference-relative optimization, while the accepted checkpoint changes the anchor used in later

 doi:10.20944/preprints202608.0051.v1

likelihood ratios. This is objective-state adaptation within one continuing lineage; preference-pair admission is an upstream supervision decision.

Some systems expose several bounded objective controls through one recipe interface. AI Training Manager reads active-run telemetry and prior intervention outcomes, proposes schema-constrained changes to loss weights or regularization, and persists only changes that pass deterministic range, freshness, and safety checks [311]. The accepted recipe governs later updates, while the manager model, action schema, verifier, and application protocol remain fixed.

Parameter-Update Adaptation

Parameter-update adaptation preserves the established target and changes how it is converted into parameter movement. AdaLRS provides the clearest example of evidence-conditioned step-size control: when loss-descent velocity slows, it probes a higher learning rate, keeps the probe if the velocity improves, and otherwise rolls back and applies a lower rate [312]. The retained learning-rate state controls later steps in the same run, while the probe and rollback rules themselves remain fixed.

The legal support of an update can evolve independently of its magnitude. Evolving Parameter Isolation accumulates squared-gradient evidence into a sensitivity estimate and periodically converts it into a protection mask. The mask suppresses AdamW updates and weight decay on newly important parameters, while parameters whose estimated importance has faded can re-enter the trainable set [313]. The persistent mask therefore determines which coordinates later updates may modify.

Batch adaptation changes the stochastic gradient estimator and often couples that change to the effective step size. COPUS and KungFu use online gradient-noise evidence to revise the batch regime used by subsequent updates; COPUS additionally writes micro-batch size, accumulation state, and a coupled Adam learning rate [314,315]. Pollux and Sia mediate the same optimizer-facing decision through learned statistical-efficiency or goodput models, selecting later batch sizes and associated learning-rate scaling [316,317]. These placements concern only batch and learning-rate state. Their independent changes to GPU allocation, parallel execution, placement, and worker topology belong to Training Infrastructure Evolution.

ONES and DYNAMIX provide cross-domain precedents for broader batch controllers. ONES uses online evolutionary scheduling to select a batch assignment and applies a coupled learning-rate scaling to later gradient steps [318]. DYNAMIX trains a PPO controller from training and systems evidence; the retained controller then chooses per-worker batch sizes for later update windows [319]. The former evolves candidate schedules, whereas the latter learns the update controller itself, but both leave their governing schedule-search or PPO procedure fixed.

Finally, RFT-FM shows how a diagnosis can feed a bounded update-control interface: its intervention stage receives an identified fault, writes a constrained corrective configuration, and resumes or restarts reinforcement fine-tuning under it [320]. The fault analysis itself belongs to Experiment Diagnosis in Section 5.2.1; what changes here is the configuration that controls subsequent updates.

Across these methods, the evolving object is a bounded objective or parameter-update state. The next subsection moves to the executable substrate: training code, parallel layout, runtime, placement, and resource scheduling.

5.1.3. Training Infrastructure Evolution

After supervision has been shaped into a parameter update, the training infrastructure determines how that update is actually executed: which workers participate, where model shards and computation are placed, how devices communicate, and how the workload is decomposed and scheduled across the cluster. *Training Infrastructure Evolution* occurs when an active training lineage rewrites this persistent execution state on the basis of its own measurements, so that later updates run under a configuration the run itself has selected. The evidence is typically gradient noise, throughput, or cluster load; the writeback is a new parallel layout, a resized worker set, or a revised resource allocation, often mediated by online performance estimates that training continues to refine. The boundary of the category lies in the origin of the decision rather than the form of the controller: reconfiguration that follows

 doi:10.20944/preprints202608.0051.v1

a predetermined schedule remains ordinary training, whereas even a fixed rule qualifies once live measurements decide which configuration is written back. Because a single reconfiguration often adjusts batch size or learning rate as well, Section 5.1.2 treats the optimization semantics of those changes, and this subsection follows their execution-side effects.

COPUS provides the clearest direct LLM example. During one pretraining run, it combines an online, 3D-parallel-aware gradient-noise estimate with a measured throughput table and ranks feasible global-batch, micro-batch, and data-, tensor-, and pipeline-parallel configurations under a fixed Goodput objective. When another topology offers sufficient projected gain after accounting for switching cost, COPUS pauses at an optimizer-step boundary, rebuilds process groups, reshards model and optimizer state, and resumes the same run under the selected layout [314]. The infrastructure-side successor is therefore an enacted parallel configuration that governs later updates in the same model lineage.

Earlier systems expose narrower run-internal control surfaces. KungFu embeds monitoring and control operators in the training dataflow, allowing a fixed user-authored adaptation policy to use live gradient or throughput measurements to resize the worker set or replace the collective-communication topology [315]. The policy remains fixed, while its evidence-dependent output becomes the distributed state used by subsequent iterations. EARL specializes this pattern to agentic LLM reinforcement learning: startup profiling maps context-length ranges to parallel configurations, and the current context length and system load select the layout used by later rollout and experience-preparation stages [321]. Its selector provides bounded infrastructure adaptation; the lookup rule and layout-aware dispatcher remain fixed, and EARL does not dynamically reconfigure the gradient-update stage.

At cluster scope, training progress can be combined with the state of several active jobs. Optimus repeatedly fits per-job convergence and resource–speed models from observed loss and throughput, then changes worker and parameter-server allocations according to their predicted effect on completion time [322]. ONES adds an online evolutionary scheduler: epoch-level progress and cluster state score a population of job–GPU mappings, and the selected mapping changes worker topology and placement before the affected jobs continue [318]. These systems establish predictive and evolutionary resource scheduling as broader model-training precedents, while keeping the scheduling objective and update rule fixed.

Pollux couples the job and cluster levels through Goodput. Each job continually estimates statistical efficiency and throughput from gradient and timing measurements; a cluster scheduler uses the resulting functions to revise GPU allocations, after which the same job continues under the new resource state [316]. Sia extends this feedback path to heterogeneous clusters by learning per-job performance across GPU types and selecting GPU type, count, and placement at successive scheduling rounds [317]. Both systems also coordinate batch-related controls, but their infrastructure contribution is the persistent reassignment of workers and devices from evidence collected during ongoing training.

Rubick broadens the mutable object from resource quantity to the execution plan itself: runtime measurements refine a joint model of resource allocations and parallelization strategies, and a fixed scheduler applies the selected resource–plan pair through checkpoint and resume while preserving the global batch size [323]. Subsequent updates may therefore use a different organization of computation, communication, and memory without changing the learning objective.

Across these systems, measured evidence selects an infrastructure state that is written directly into an active training lineage. Profiling, analytical enumeration, or surrogate search inside the controller does not create the experiment boundary used in Section 5.2. That boundary appears when an infrastructure change becomes a distinct runnable trainer candidate, receives downstream evidence from a declared training experiment, and is then promoted, rejected, or rolled back. We next examine this experimental search loop in *Outer-Loop Trainer Search through Experiments*.

5.2. Outer-Loop Trainer Search through Experiments

Outer-Loop Trainer Search through Experiments organizes trainer self-improvement as a cross-experiment process of candidate construction, evaluation, and retention, as shown in Figure 7. A

 doi:10.20944/preprints202608.0051.v1

proposed change is instantiated as a runnable trainer candidate and evaluated through a training experiment. Evidence from the experiment determines whether the candidate is rejected, evaluated further, or promoted. A promoted candidate becomes the retained trainer version used in subsequent experiments.

We organize this literature around three functions. *Experiment Diagnosis* interprets experiment evidence and explains the observed result. *Trainer Candidate Search* constructs runnable alternatives and organizes their version relationships. *Trainer Candidate Evaluation* gathers evidence and decides whether a candidate should be rejected, continued, or promoted. Together, these functions use evidence from training experiments to evaluate trainer candidates and determine which version is retained for subsequent experiments.

5.2.1. Experiment Diagnosis

Experiment Diagnosis converts artifacts from a completed or bounded training experiment into an evidence-linked account of its outcome. The account identifies the observed failure, a likely cause, the affected trainer-side object, the strength and uncertainty of the evidence, and a hypothesis that a later experiment could falsify. The affected object may be supervision code, an objective or optimizer, a training configuration, or execution state. A scalar score can rank experiments or trigger inspection, but ranking alone does not explain a result. Diagnosis reads the observation surface produced by the experiment. Changes to future instrumentation belong to Meta-Loop Improvement-Mechanism Evolution, while runnable alternatives and promotion decisions belong to Trainer Candidate Search and Trainer Candidate Evaluation.

We organize diagnosis as three successive transformations. *Diagnostic Evidence Synthesis* structures heterogeneous artifacts while preserving their experimental provenance. *Trainer Fault Localization* maps the resulting evidence to a failure mechanism and an affected trainer-side object. *Diagnostic Hypothesis Validation* qualifies that account through historical or controlled comparison and records what later evidence could refute it.

Diagnostic Evidence Synthesis

Diagnostic Evidence Synthesis turns heterogeneous run artifacts into a representation from which a failure can be inferred. Useful evidence records when an observation occurred, which component produced it, which behavior it describes, and which experiment or trainer version it belongs to. This provenance is essential because the same terminal outcome can arise from sparse supervision, ineffective updates, invalid output formatting, evaluator failure, or broken execution.

Run-local synthesis assembles optimization telemetry and behavior from the same experiment. Reward, KL, entropy, policy-loss, and response-length traces provide a temporal fingerprint of RFT failure modes [320]. Validation, gradient-norm, and clipping trajectories add checkpoint-level evidence for adaptive-strategy diagnosis [324]. Post-run views add losses and failed-output examples from the same experiment [327], so a terminal regression is represented through temporal and behavioral signatures rather than as a single scalar event. These signatures remain correlational: when a controller changes several settings together, they bound plausible explanations but do not identify a unique cause.

Version-bound artifact packets add provenance across code and execution. Metrics, rollouts, configurations, logs, diffs, legality outcomes, and runtime traces are attached to the candidate or distributed configuration that produced them, separating a valid negative result from an invalid run or measurement failure [325,328,330]. Within this family, EvoTrainer interprets its artifact inputs through four diagnostic layers—score, signal, behavior, and version [325].

Persistent lineages add comparative context by preserving parent deltas, bad-case attributions, effect estimates, failed ranges, and dead ends around an incumbent. Their common role is to relate the current outcome to known version changes and contrary outcomes, not to claim causal identification from history alone [331–333].

 doi:10.20944/preprints202608.0051.v1

Trainer Fault Localization

Evidence becomes diagnosis when it is mapped to an affected trainer-side object and a failure mechanism. A useful localization has the form *artifact pattern* → *failure mode* → *affected object* → *predicted consequence*. The output remains an explanatory claim: constructing the corresponding modification begins in Trainer Candidate Search.

Learning-side localization is most defensible when it maps a repeatable observation pattern to a bounded trainer object. Fine-grained fault schemas and layered evidence connect temporal anomalies or behavioral failures to reward computation, generation, optimization, credit assignment, tool interaction, exposed training controls, or harness code [320,324,325]. These mappings define a plausible intervention surface, but LLMZero's coordinated multi-parameter transitions do not isolate a single causal control. OptiCo extends the same schema-based localization to distributed-runtime failures, mapping configuration and execution evidence to a root-cause category, implicated fields, constraints, and an uncalibrated, agent-reported confidence used to route later search [330].

Known version deltas provide another narrowing device. Parent–child trajectories, sibling and historical comparisons, and champion-relative dead-end records restrict a hypothesis to what changed while preserving counterevidence; they still leave simultaneous edits and inherited state as possible confounders [331–333].

At the supervision interface, cross-domain reward-design systems connect policy behavior to specific signal defects. Component trajectories expose inactive or dominant terms, trajectory preferences expose ordering violations, and evidence-keyed critiques name a bounded shaping failure before candidate generation [334–336]. These are transferable localization interfaces, not direct evidence of general LLM trainer diagnosis. Conversely, a free-form reflection produced jointly with its mutation does not by itself establish localization; the affected trainer object and its supporting observations must remain separable from the next candidate.

Diagnostic Hypothesis Validation

Diagnostic Hypothesis Validation asks whether a localized explanation is sufficiently discriminating to guide a later test. A complete hypothesis states the suspected fault, the affected trainer-side object, the supporting and contradicting observations, and an outcome that would refute the proposed cause. This stage interprets available comparisons and specifies the required test. The execution of a new branch and the decision to promote it remain in Trainer Candidate Evaluation.

Evidence strength increases from historical recurrence to matched intervention. Version histories can reveal whether a proposed relation recurs or encounters counterexamples across parent–child changes and sibling branches, but they do not remove confounding from seed, checkpoint state, horizon, or coupled edits [325,332,333].

Matched-state probes provide sharper local tests. A same-checkpoint fork can compare a shaping update with a no-change continuation and reject a near tie [336]. A richer matched-state design clones policy, critic, and optimizer state across reward hypotheses, then uses winner margin, repeated-fork agreement, entropy, and abstention to express when the contrast is non-discriminating [346]. These cross-domain probes qualify a hypothesis under a bounded horizon; their execution, budget allocation, and promotion remain in Trainer Candidate Evaluation.

Implementation and measurement claims require a separate falsification surface. Single-factor branches, historical-rollout backtests, evaluator or harness audits, legality checks, code review, and progressive mini-runs can expose leakage, invalid code, constraint violations, or a failure that does not improve when its implicated object changes [325,327–329]. These checks can reject an explanation without turning a successful repair into proof of a general causal mechanism.

Experiment Diagnosis therefore ends with an evidence-linked, uncertainty-qualified, and falsifiable trainer hypothesis. It records what failed, why a particular trainer-side object is implicated, which observations support or contradict the account, and what result would disconfirm it. Section 5.2.2

 doi:10.20944/preprints202608.0051.v1

materializes that hypothesis as a runnable trainer candidate. Section 5.2.3 executes discriminating tests and decides whether the candidate should be rejected, continued, or promoted.

5.2.2. Trainer Candidate Search

Trainer Candidate Search turns a diagnosed hypothesis into a runnable alternative to the retained trainer: a candidate specification that states what changes, which parent version and training state it starts from, and how to execute it as a training experiment. Whether the candidate should be kept is not decided here; that is the task of Trainer Candidate Evaluation.

We distinguish three classes of candidates by the depth of the required change. *Training-Configuration Search* stays within the existing trainer interface and chooses values, schedules, or pre-implemented options. *Learning-Objective and Signal Search* changes what the model is trained to optimize—rewards, losses, targets, or credit assignment—while the surrounding training procedure remains intact. *Executable Training-Procedure Search* rewrites the training computation itself, from update rules and control flow to complete training pipelines. A candidate that spans several levels is classified by the deepest change it requires; the finer boundaries are drawn within each class below.

Training-Configuration Search

Training-Configuration Search constructs candidates by assigning values, schedules, or named alternatives already exposed by a fixed trainer interface. Its scope includes hyperparameters, static mixtures over available data sources, batch and sampling settings, checkpoint anchors, fixed reward coefficients, resource allocations, stopping conditions, and schedules over already defined training stages. The candidate may coordinate many fields and may define a multi-stage recipe, but the meaning of every field and the executable implementation behind every option remain fixed.

The category is bounded by the exposed interface, not by the apparent size or file format of the intervention: selecting a pre-implemented optimizer from a registry, retuning existing reward coefficients, or scheduling already defined stages are all configuration choices, whereas a candidate leaves the category as soon as it changes what a field means or how an option is implemented. A horizon or data allocation belongs to the candidate only when it is retained as part of the trainer recipe; a temporary budget used to screen candidates belongs to the evaluation contract, and generating or curating the underlying data belongs to the data system.

At its narrowest, candidate construction is conventional hyperparameter search over a declared schema. Efficient HPO for LLM RL uses a cost-aware acquisition rule to propose GRPO configurations from a fixed parameter space [347]. The inner loop of Bilevel Autoresearch instead asks an LLM search operator to make incumbent-relative parameter edits, but each first-order candidate is still an assignment within the unchanged training program [238]. Both systems can instantiate a candidate without introducing a new learning objective or executable operator.

Natural-language recipe construction remains configuration search when the backend defines the available training operations. AutoTrainess and FT-Dojo both assemble runnable post-training recipes within fixed execution interfaces [326,327]. AutoTrainess selects supported stages and methods and emits a complete LlamaFactory configuration, whereas FT-Dojo revises an incumbent SFT or LoRA recipe through exposed controls such as the update mode, optimizer settings, formatting, and duration. The proposal language is open-ended, but the candidate vocabulary is bounded by the backend.

Configuration candidates may also bind parameter choices to inherited training state. LLMZero constructs each child phase from a coordinated configuration, a parent checkpoint or scratch initialization, and a phase budget, so successive children form an adaptive multi-stage recipe within the same GRPO pipeline [324].

The same boundary extends beyond single-model recipes. OptiCo applies diagnosis-guided, rule-constrained transformations to distributed-training configurations while leaving the underlying runtime implementation fixed [330]. As a cross-domain boundary case, LLM-ALSO searches a predefined potential-based reward-shaping space [336]. Selecting terms and coefficients from that fixed basis is a configuration choice, even though the configured fields affect reward shaping.

 doi:10.20944/preprints202608.0051.v1

Learning-Objective and Signal Search

Learning-Objective and Signal Search changes what the learner is trained to optimize or how supervision is assigned to its behavior, while the surrounding training scaffold stays in place. Its scope includes objective and loss definitions, reward and shaping functions, target construction rules, advantage transformations, supervision weights, and mechanisms that distribute credit across examples, tokens, actions, or trajectories.

Classification follows the functional role of the signal, not its surface form: a signal may be written declaratively, symbolically, in natural language, or as executable reward or loss code, and it belongs here as long as an otherwise fixed learner consumes it through a stable interface. Selecting among pre-implemented objectives or tuning their coefficients remains *Training-Configuration Search*, and a candidate stops being a signal candidate once realizing it requires changing the learner's own computation or control flow. This category also concerns the objective used to train the model, not the criterion used to certify the candidate: a metric that only compares completed candidates belongs to Trainer Candidate Evaluation, whereas any quantity consumed during optimization belongs to the candidate, even when it is also reported as an experiment metric.

Executable syntax alone does not make a candidate a general training procedure. DiscoPOP uses an LLM to write and revise preference-optimization losses as Python functions [348], while POISE instantiates policy-optimization candidates through a shared interface for losses, advantage transformations, and regularization [333]. In both cases, the code changes the mathematical signal consumed by an otherwise stable training scaffold.

Signal candidates differ in how tightly their semantics are constrained. Differentiable Evolutionary RL searches compositions of predefined symbolic reward modules and their parameters [349]. Search-Driven Reward Synthesis instead generates free-form reward code and may combine selected rewards into an ensemble [350]. The former searches a structured signal grammar; the latter enlarges the space of expressible reward functions while preserving a fixed reward-call interface.

Cross-domain reward-design systems can be grouped by how completed policy evidence produces the next signal candidate. Incumbent-centered methods feed trajectory ordering, reward-component behavior, or downstream task metrics back into revision of a retained reward program [334,335,351]. Population methods maintain several reward programs and generate descendants from selected or retained elites [337,340,352]. Their diversification mechanisms range from crossover and mutation to island-based migration. Structured variants preserve richer cross-candidate state [341–343]. They use this state to recombine reward modules and tune their coefficients, revisit alternative tree paths, or carry both reward and policy state into the next search round. These mechanisms organize alternatives over a stable signal interface; they do not by themselves turn the candidate into a general executable training procedure.

Executable Training-Procedure Search

Executable Training-Procedure Search treats the training computation itself as the mutable candidate. Rather than filling fields in a fixed interface or swapping a bounded signal function, it writes or revises executable operators and their composition: update rules, sampling and rollout control, data flow and stage transitions, training-state management, and entire training scripts or pipelines. The category turns on which logic must change, not on whether code appears: code that only computes a reward remains a signal candidate and a script that only serializes settings remains a configuration candidate, whereas a candidate belongs here once its effect requires new control flow, a changed update implementation, or a recomposition of trainer components. Throughout, the improvement mechanism that generates and validates these patches remains fixed; when experiment evidence begins to revise that mechanism itself, the mutable object belongs to Section 5.3.

Procedure search begins when a candidate must change the executable training computation. AIDE and the Autoresearch task in AIRA-Design both organize draft, debug, and improve operations over runnable training scripts [353,354]. AIDE grows a tree through atomic script revisions, while

 doi:10.20944/preprints202608.0051.v1

AIRA-Design explores branches that can alter training-loop and optimizer logic. TREX extends this pattern from local script edits to a Researcher–Executor workflow that turns a training plan into an executable fine-tuning scheme [332]; we discuss its trainer-side implementations here, while its data-only branches belong to the data system (Section 4).

Repository-scale systems retain an executable incumbent and construct later procedures relative to it. Auto Research and AutoScientists both branch program edits from a current champion and share the resulting lineage across later experiments [328,331]. EvoTrainer materializes each proposed intervention in an isolated worktree and records it as a versioned trainer branch [325]. The candidate is therefore a versioned implementation whose inherited code state is fixed before evaluation begins.

Population operators offer different ways to generate such implementations. GEAR applies genetic mutation and crossover to a bounded frontier of complete trainer programs [355]. SMCEvolve treats complete pretraining scripts as particles and constructs descendants through sequential local edits or full rewrites [356]. Towards Execution-Grounded Automated AI Research separates ideation from implementation, using an executor to materialize population proposals as runnable pre-training or post-training procedures [357]. The search operator differs across these systems, but each candidate is executable before its training outcome is judged.

A procedure candidate need not span an entire repository: CliffSearch evolves executable optimizer classes whose update equations and internal state transitions replace the incumbent optimizer implementation within an otherwise fixed training stack [329]. The changed object is the optimization operator itself, not a registry choice or a coefficient within a fixed operator.

Cross-domain pipeline systems provide secondary precedents for broader procedure representations. ML-Agent edits versioned ML scripts, while MLE-Ideator separates a strategic proposal from the agent that implements it as an executable solution [344,345]. MLEvolve represents complete ML pipelines as nodes in a progressively expanded graph [338], and the Self-Evolving Recommendation System uses specialized personas to materialize optimizer and reward-definition code changes [339]. These works demonstrate transferable candidate representations and search operators.

Across all three classes, Trainer Candidate Search ends once a runnable candidate has been materialized together with its parent version, inherited state, and declared scope of change. These outputs specify what differs between candidates; Trainer Candidate Evaluation supplies the comparison contract and determines whether that difference is beneficial.

5.2.3. Trainer Candidate Evaluation

Trainer Candidate Evaluation decides, from measured training consequences, whether a runnable candidate is rejected, evaluated further, or promoted. The measurement itself is simple in form: train the candidate from a declared initial state under a specified budget, then score the resulting model on a designated evaluation set. What gives the score meaning is the evaluation contract—the initial state, budget, evaluation data, comparison baseline, and promotion rule; change any of these and the score supports a different claim.

We distinguish three evaluation regimes by the role their evidence plays. *Full-Run Evaluation* compares candidates at target fidelity under the declared contract. *Multi-Fidelity Evaluation* uses cheaper approximations to decide which candidates deserve more compute. *Robust Validation* checks that an apparent advantage is really caused by the candidate and survives beyond the conditions where it was first observed. A system may combine these regimes or skip unnecessary ones; what characterizes it is the strongest evidence it requires before promotion.

Full-Run Evaluation

Full-Run Evaluation judges a candidate by what it produces: the candidate is trained to completion under the target evaluation contract, and the resulting model or policy is assessed by a criterion external to the candidate itself. Internal rewards, training losses, or self-reported rationales may guide learning, but they do not certify the candidate; promotion follows the measured utility of the trained system. “Full-run” is relative to the declared contract rather than to the largest experiment one could

 doi:10.20944/preprints202608.0051.v1

conduct: a bounded experiment still counts when its completed outcome directly decides rejection or promotion under the declared rule.

Fix an evaluator-owned target contract. A target contract keeps mutable code separate from evaluator-owned data, budgets, legality checks, and metric extraction. Whether candidates are generated through a specialist lineage, a particle population, or a general proposal–executor split, each must materialize as a runnable training recipe and return measured status and utility under the same external contract [328,356,357]. Search topology therefore changes how candidates are proposed, not what Full-Run evidence means.

Measure the candidate through its downstream learner. For bounded rewards or optimizers, utility still passes through the learner they produce. A reward candidate can be scored by validation performance of its trained inner policy, with the selected reward later training a fresh policy for held-out evaluation [349]. More generally, reward programs and optimizer implementations are compared through policies or models trained under a common scaffold [329,334]. The reward-program case is a cross-domain precedent; the transferable principle is that internal reward values and proposal rationales do not certify candidate utility.

Write the completed outcome into version state. Completed evidence changes trainer evolution only when subsequent experiments inherit a new baseline or an explicit negative record. Champion replacement, best-configuration retention, keep-or-discard decisions, and branch dispositions such as keep, prune, revert, or merge write measured results into version state, with consequential EvoTrainer promotion remaining human-gated [238,325,327,331]. Full-run evidence therefore establishes utility only under the declared contract; cheaper evidence determines whether that contract should be reached, and additional controls determine how strongly its outcome should be trusted.

Multi-Fidelity Evaluation

Multi-Fidelity Evaluation compares or filters candidates using evidence collected at systematically related levels of cost and approximation to the target training contract. Fidelity may vary with training horizon, data or model scale, rollout count, environment realism, the amount of newly collected experience, or the directness of the evaluation signal. Lower-fidelity observations provide inexpensive but imperfect estimates; their defining role is to decide whether a candidate should be rejected, continued, or granted a more expensive evaluation.

Replay-based proxies, partial training, checkpoint-based short runs, smaller-scale experiments, learning-curve prediction, successive halving, and adaptive budget allocation all qualify when they forecast target-fidelity utility. Low cost alone is not the criterion: a uniformly short experiment without such gating is simply a full run under a smaller contract, and a static validity check establishes executability rather than likely training benefit.

Reuse existing evidence. Candidate screening can begin before a new training run. EvoTrainer backtests proposed reward and filtering changes on historical rollouts, using group-level reward behavior and output audits to expose dead groups, reward leakage, or undesirable behavior before launching a new branch [325]. As a cross-domain reward-design precedent, CARD rejects reward candidates whose scores fail to preserve the ordering of stored high- and low-quality trajectories [335]. FT-Dojo combines schema and data checks with a reduced mini-run before full fine-tuning [327]. The static gates establish that a candidate can run; replay or reduced-run evidence becomes multi-fidelity evaluation only when it forecasts training value and controls whether the candidate receives a larger budget.

Run controlled low-cost probes. When retained evidence cannot rank candidates, RHyVE clones the same policy, critic, and optimizer state into short PPO forks, changes the reward hypothesis, and compares the resulting trajectories before committing to a longer phase [346]. The shared checkpoint reduces both repeated training cost and variation in the candidates' starting states. Related cross-domain systems strengthen this comparison in different ways: LLM-ALSO adds a no-change branch from the same checkpoint and abstains when the short-run advantage is unstable, while ROSKA gives each reward–policy candidate approximately 200 PPO updates before extending selected candi-

 doi:10.20944/preprints202608.0051.v1

dates [336,343]. Efficient HPO for LLM RL treats model scale and training budget as fidelity variables, using proxy trials to decide which configuration–checkpoint pairs merit target-model training [347]. A probe belongs to this regime because its result gates the next allocation of compute, not merely because the run is short.

Allocate budget as evidence arrives. LLMZero evaluates a branch while it trains: at fixed intervals, a frozen early-stopper compares the branch's validation trajectory and diagnostic curves with those of the incumbent, stopping the branch once further compute is unlikely to change the ordering [324]. A stopped branch writes its score back to the search tree, discouraging repeated allocation to the same low-potential direction. Efficient HPO for LLM RL similarly applies fixed gates to reward and KL dynamics, terminates failing trials, and resumes surviving checkpoints at higher fidelity [347].

Account for fidelity risk. The goal of these approximations is to preserve the candidate ordering that matters at target fidelity, rather than to predict every final score precisely. That ordering can still invert: a slow-starting candidate may be pruned too early, a configuration that succeeds on a proxy model may fail to transfer, and a short-fork gain may reflect noise or a favorable checkpoint. Multi-fidelity evaluation therefore decides where additional compute should go; robust validation determines whether the resulting advantage is reliable enough for promotion.

Robust Validation

Robust Validation asks whether a candidate's apparent advantage is attributable, reproducible, and persistent enough to support promotion. Controlled comparison first limits confounding: candidate and reference start from matched or explicitly comparable states, receive comparable budgets, and are scored under the same data and metric conditions. Repeated seeds, shared-checkpoint forks, no-change controls, uncertainty-aware thresholds, and abstention rules then separate a repeatable intervention effect from experimental noise.

Validation may further require the advantage to survive a second evaluation boundary, such as held-out tasks or seeds, a different horizon or model scale, a shifted data distribution, human assessment, or deployment traffic. "Robust" therefore describes the reliability of the evaluation conclusion, not only adversarial robustness of the trained model. The boundary with the previous regime lies in purpose rather than technique: a scale change or held-out score that economizes screening belongs to *Multi-Fidelity Evaluation*, while the same instrument counts as robust evidence only when it adds a confirmation boundary beyond the evidence used to select the candidate.

Construct a matched counterfactual. LLM-ALSO creates every proposed reward-shaping branch together with a mandatory no-change branch from the same policy checkpoint. Its stability-aware comparison penalizes transient peaks, promotes a new shaping configuration and endpoint only when the margin is clear, and otherwise preserves the incumbent [336]. RHyVE likewise clones policy, critic, and optimizer state across reward-hypothesis forks, then examines winner margins, repeated-fork agreement, and winner entropy before committing to a reward or phase schedule; ambiguous evidence produces abstention rather than forced promotion [346]. The short forks allocate compute in Multi-Fidelity Evaluation, while their matched starting state, stability tests, and abstention rule support attribution here. EvoTrainer applies the same logic at version scale through single-factor branches and targeted audits, including rejection of a score gain that disappeared after Git-history leakage was removed [325].

Calibrate promotion to experimental noise. AutoScientists estimates a seed-induced noise floor from paired executions of identical code and compares each candidate's validation gain with a calibrated band [331]. A candidate with a clear gain may replace the shared champion directly; one with a marginal gain must improve under a second seed as well, and a candidate that fails this confirmation is retained as a near miss rather than promoted. This rule turns repeated outcomes into a candidate-level decision and prevents one favorable stochastic run from becoming the baseline for all later experiments.

Cross an independent confirmation boundary. Efficient HPO for LLM RL uses proxy models and partial budgets to allocate search compute, then trains the selected configuration on the target model to test whether the proxy ordering transfers at the intended scale [347]. As a production-ML precedent,

 doi:10.20944/preprints202608.0051.v1

the Self-Evolving Recommendation System sends offline survivors through versioned training and guarded online A/B evaluation; safety violations can abort the candidate, while delayed north-star metrics and run status are written to the experiment journal [339]. Such a second boundary can confirm the apparent advantage, restrict it to the conditions where it survives, or leave the incumbent unchanged. Robust validation therefore strengthens a promotion claim through attribution, repetition, or transfer, while leaving the validation and promotion rules themselves fixed.

Robust Validation supplies the strongest candidate-level basis for promotion while keeping the improvement mechanism fixed. Evaluating whether a mutable mechanism will produce better future candidates instead crosses into Meta-Loop Improvement-Mechanism Evolution.

5.3. Meta-Loop Improvement-Mechanism Evolution

Meta-Loop Improvement-Mechanism Evolution begins when evidence from actual model-training experiments changes the persistent decision mechanism that governs later trainer experiments. A qualifying cycle evaluates a trainer candidate, observes the resulting training outcome, and transforms Imp_t into a successor Imp_{t+1} whose changed decisions affect a later cycle. The writeback may alter diagnosis, proposal and search, evaluation, or promotion. Accumulating logs, context, memories, archives, populations, posteriors, or tree statistics under unchanged decision rules leaves the improvement mechanism fixed.

The successor mechanism may be realized through persistent change at two non-exclusive implementation levels. In *Improvement-Model Evolution*, model-training evidence updates the parameters of a learned model that makes later improvement decisions; this model is distinct from the downstream model trained within each experiment. In *Improvement-Harness Evolution*, the evidence revises the persistent operational logic that structures and executes those decisions, including executable analyzers, reusable skills, runner or controller code, and role-allocation policies.

5.3.1. Improvement-Model Evolution

An Improvement Model evolves when outcomes from trainer experiments become learning signals for the model that conducts later improvement. Differentiable Evolutionary Reinforcement Learning provides a direct bilevel example: its Meta-Optimizer composes symbolic reward functions from a fixed primitive vocabulary, and each candidate reward governs a separate inner GRPO run [349]. Held-out performance of the trained inner policy is the outer reward that updates the Meta-Optimizer, so the successor proposes different reward structures for later policy-training experiments while the primitive vocabulary and training protocol remain fixed.

The same writeback can train a research policy that proposes configurations or executable trainer changes. In *Towards Execution-Grounded Automated AI Research*, an executor turns proposed pre-training or post-training ideas into code patches and obtains rewards from bounded model-training runs. Its RL branch uses these rewards to train the Ideator with GRPO, so the successor Ideator governs later proposals; its archive-conditioned evolutionary branch keeps the proposal procedure fixed and remains Outer-Loop Trainer Search through Experiments [357].

5.3.2. Improvement-Harness Evolution

An Improvement Harness evolves when training evidence revises persistent operational logic around the Improvement Model. The successor may be executable search code, an analysis procedure, a reusable skill, or an organizational policy that changes how later experiments are interpreted, generated, or allocated.

Executable controller revision. Bilevel Autoresearch makes the writeback explicit at the runner boundary. Its inner loop edits the active GPT training program and evaluates each edit through a bounded pre-training run. The outer level then reads the accumulated trace and current

`runner.py`, generates a replacement search procedure, validates that procedure, and installs it or restores the incumbent. An accepted runner changes the exploration operators applied to subsequent

 doi:10.20944/preprints202608.0051.v1

trainer experiments; the outer schedule, import gate, validation metric, and inner keep-or-discard rule remain fixed [238]. GEAR-Evolve performs a narrower controller repair: observed crossover failures produce retained source-level changes to parent eligibility, ancestry traversal, exhausted-pair blocking, and expansion rules. The elite frontier and ancestry records remain ordinary search state; the controller patches are the successor mechanism that governs later language-model training candidates [355].

Diagnostic and organizational revision. *EvoTrainer* provides a representative instantiation of this pattern. Each isolated RL branch binds scores, training signals, behaviors, configurations, code differences, and rollouts to an explicit version. The harness interprets this packet through score, signal, behavior, and version layers, while a version ledger, case memory, skill library, and search trace preserve what was tried and why it succeeded or failed. Candidate interventions are materialized in isolated worktrees and normally change one factor—such as reward, data, hyperparameters, or filtering—so branch evidence can support keep, prune, revert, or merge decisions. When the available evidence cannot explain an outcome, distinguish competing hypotheses, or justify the next intervention, EvoTrainer records a diagnostic gap. The retained response may expand metrics, specialize an analyzer, revise a procedure, retrieve external evidence, or add a validated reusable skill. These changes alter how later branches are diagnosed and constructed; the ledger and memories remain evidence stores rather than the evolving mechanism itself, and costly runs or consequential promotions remain human-gated [325]. Figure 8 maps this distinction onto the system architecture.

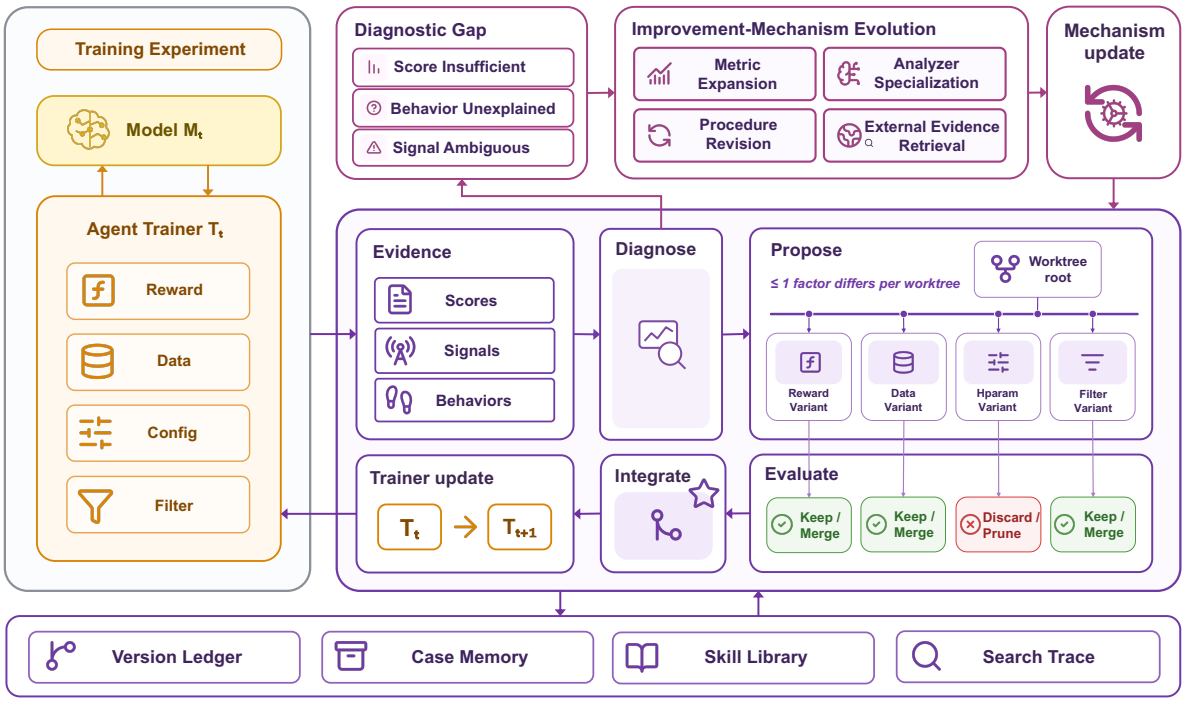

Figure 8. EvoTrainer as a representative coupling between trainer-version search and improvement-mechanism evolution. Versioned training branches expose scores, signals, behaviors, configurations, code differences, and rollouts for evaluating single-factor trainer candidates that may be retained, merged, pruned, or reverted. Diagnostic gaps can persistently revise metrics, analyzers, procedures, retrieval, and reusable skills, thereby changing how later branches are interpreted and constructed; consequential runs and promotions remain human-gated [325].

AutoScientists revises the organizational interface through which improvement work is assigned. Model-training evidence can trigger persistent creation, merging, splitting, retirement, or rebalancing of specialist teams, after which the revised roster and allocation policy control who investigates each search axis and launches later experiments. Its queues, champion history, and dead-end records supply evidence but do not independently constitute mechanism evolution [331].

 doi:10.20944/preprints202608.0051.v1

Taken together, the three loops trace a single progression: persistent change moves from the active trainer, to retained trainer versions, to the improvement mechanism itself. Their maturity declines along the same path. Inner-Loop Trainer Adaptation rests on the most direct evidence, since each revision acts and is observed within the same training lineage. Outer-Loop Trainer Search has runnable systems in all three candidate classes, but its bottleneck is the evaluation contract: target-fidelity experiments are expensive, cheap proxies can invert the candidate ordering, and few systems implement the matched controls, noise calibration, and confirmation boundaries that Robust Validation requires, so promotion evidence lags behind candidate generation. Meta-Loop Improvement-Mechanism Evolution remains the least developed: current systems evolve bounded parts of *Imp* under fixed higher-order boundaries—a protected evaluation signal, a permitted modification interface, an execution budget, or human release authority—that keep successor mechanisms comparable and auditable. Trainer self-improvement today is therefore predominantly self-improvement under a fixed mechanism, with recursion realized only in bounded and audited forms.

The trainer loop also does not close the full self-improvement cycle on its own: its supervision consumes the tasks, trajectories, and verification signals supplied by the data system, its revisions take effect only through the model they update, and a promoted trainer version changes which future experience is worth collecting. Section 6 therefore moves from this component-level view to a unified treatment in which the model, harness, data system, trainer, and improvement mechanism improve as one coupled system.

6. Co-Improvement of Multiple System Components

Sections 3, 4, and 5 review how the agent harness, data system, and trainer improve individually. The component-level view is useful, but it is not sufficient for sustained improvement in agent systems. In a coupled system, any component can become the main constraint on overall performance. If one component improves while the others remain fixed, further improvement of that component may produce diminishing returns. System-level bottlenecks are dynamic rather than static, and they can shift as the capabilities and components of the agent system change. Therefore, single-component self-improvement cannot fully identify or address the moving constraints that arise in an evolving agent system.

Following the unified research framework in Section 2.3, we further study collaborative improvement of agent systems. From a system-level perspective, co-improvement refers to the joint improvement of multiple system components through their dependency relationships. Based on the components involved in the collaborative improvement, this section organizes related work into three categories.

6.1. Harness-Trainer Co-Improvement

Harness-trainer co-improvement couples the improvement of harness modules with the training of the agent model. In this setting, improved skills, tools, workflows, or scaffolds make training signals more informative, while training outcomes reveal which harness modules should be revised next.

Training Conditioned on Agent Skills and Experience.

Accumulated skills or experience records can serve as conditioning signals, curriculum guides, or distillation targets during agent model training. SkillRL and ARISE maintain skill libraries that condition policy learning. After training, the updated policy generates new trajectories that are used to expand or refine the skill library [366,367].

Harness Refinement Driven by Training Feedback.

Trainer-side evidence, including policy performance, failure patterns, and reward signals, can guide revisions to harness code, workflow logic, or agent architecture. SIA places scaffold updates and weight updates into a single feedback-driven improvement loop [368]. HarnessForge models the agent

 doi:10.20944/preprints202608.0051.v1

system as an interdependent harness-policy system, where fault-guided harness tailoring is combined with harness-conditioned policy adaptation [359].

6.2. Harness-Data Co-Improvement

Harness-data co-improvement couples the improvement of harness modules with the data pipeline. Harness modules change the data that the agent system can produce or access, while data-side signals drive the creation and refinement of those harness modules.

Co-Improving Skills and Trajectory Generation.

Agent skills extracted by the harness can become the source of new training tasks or environments. Task outcomes then provide evidence for further skill refinement. CODESKILL converts historical trajectories into procedural skills and feeds the resulting library into code-repair task construction and solving loops [369]. VOYAGER demonstrates this pattern in an embodied setting, where an executable code-skill library enables exploration of progressively harder environments in Minecraft without changing model weights [370].

Co-Improving Harness Code and Trajectory Generation.

Harness code improvement and trajectory generation influence each other through mutual dependencies. Better tools and harness code expand the agent's action space and improve execution trajectories. Execution failures and debugging records from trajectory synthesis reveal capability gaps and guide the creation and refinement of new tools and harness code. Live-SWE-agent creates, debugs, and reuses executable custom tools during software-engineering tasks. The resulting tools change later interaction traces, and failures in those traces guide further tool debugging or tool creation [371]. AutoHarness synthesizes executable code harnesses, including action filters, verifiers, and code policies, from environment feedback. The accepted harness modifications alter the agent's future interaction capabilities, thereby affecting the distribution of trajectories collected in subsequent episodes [372].

6.3. Data-Trainer Co-Improvement

Data-trainer co-improvement creates a feedback loop where better training produces better data, which in turn improves training. The environment simulator and policy can improve each other through iterative interaction. Some existing studies train world models to serve as environment simulators. Real trajectories refine the world model, and the improved world model helps synthesize experience in regions exposed by the current policy. WebEvolver trains a world model that predicts web observations and produces imagined trajectories for policy learning [373]. VLAW establishes an analogous loop for robotic manipulation, where real rollouts improve an action-conditioned video world model that then generates synthetic rollouts for further policy training [374].

7. Open Problems and Future Research Directions

Despite rapid progress in self-improving agent systems, many fundamental problems on the path toward recursive self-improvement remain unsolved. These open problems concern the conditions that enable agent systems to become reliable, scalable, general, safe, and beneficial to humans as they progress toward recursive self-improvement.

7.1. Long-Horizon Evaluation of Improvements in Real-World Production

Current agent evaluations primarily measure task performance within a single evaluation run. In contrast, evaluating self-improving agent systems requires assessing whether repeated self-modifications consistently improve the reliability, robustness, and overall usefulness of subsequent system versions. To measure the effects of multi-round self-modification, future benchmarks should simulate long-horizon real-world workloads, such as maintaining large code repositories, conducting iterative scientific research, and operating long-running services. Beyond measuring capability gains,

 doi:10.20944/preprints202608.0051.v1

these benchmarks should assess whether agent systems preserve previously acquired capabilities, learn from unsuccessful modifications, and remain effective as goals, budgets, tools, and environments evolve. They should further evaluate the improvement process itself by examining whether systems become better at diagnosing their own bottlenecks, selecting beneficial modifications, and carrying improvements forward to future tasks.

7.2. Observable, Scalable, and Modifiable Training and Inference Infrastructure

RSI requires infrastructure that exposes the agent system's behavior and update process through representations that agents can inspect and improve. Existing infrastructure is typically designed and implemented as isolated components. Although many systems already record useful information, such as execution traces, training records, and branch histories, these artifacts are distributed across different storage systems, accessed through diverse interfaces, and governed by inconsistent access control mechanisms. Consequently, they do not provide a unified, agent-friendly interface through which inference-time behavior, data generation, model training, validation, and self-modification can be fully observed, reproduced, compared, and revised.

Future infrastructure should tightly integrate agents with the training and inference infrastructure by making the entire improvement process observable through standardized system representations, enabling agents to dynamically allocate and reconfigure computational resources, and allowing them to safely analyze, evaluate, and revise different parts of the training and inference infrastructure. This creates a positive feedback loop in which agents improve the infrastructure, and the improved infrastructure, in turn, enables agents to improve themselves more efficiently and reliably in subsequent iterations.

7.3. From Bounded RSI to General RSI

The capability grading standard in Section 2.2 distinguishes bounded RSI at L4 from general RSI at L5. Recent systems have demonstrated self-improvement within specific domains, such as automatically repairing code, optimizing training infrastructure, or refining task-specific improvement procedures. However, these results do not yet constitute general RSI, as the effectiveness of their improvement mechanisms remains closely tied to particular tasks, environments, tools, and evaluation criteria. The key challenge in advancing from L4 to L5 is to achieve transferable self-improvement, where the system can preserve and adapt its improvement capabilities across diverse domains. Future research should identify the fundamental principles underlying reusable improvement mechanisms while preventing the inappropriate transfer of domain-specific strategies that may perform well in one setting but fail when applied to different scenarios.

7.4. Safety and Controllability Under Recursive Self-Modification

Safety and controllability become substantially more challenging when an RSI system can modify not only task-level policies, but also the infrastructure and improvement mechanisms that govern future self-modification. As an RSI system becomes more capable, it may propose modifications to evaluators, permission rules, and version management policies. The challenge is to ensure that governance mechanisms can evolve while preventing successive system versions from weakening the safeguards that keep continued self-improvement trustworthy.

Future research should distinguish capability-oriented components from protected governance components. Capability-oriented components that primarily affect performance, efficiency, or adaptability should remain modifiable. By contrast, modifications to components that enforce safety, legal, or privacy requirements must undergo a stricter verification before acceptance. In addition, RSI systems should continuously monitor performance degradation, evaluator failures, reward hacking, privacy leakage, unsafe tool generation, and system failures. They should also preserve immutable records of rollback versions, safety alerts, and incident logs to maintain traceability across successive system versions and support human review.

 doi:10.20944/preprints202608.0051.v1

7.5. Human-Expert and Agent Co-Improvement

Self-improvement should not be studied only as the replacement of human iteration by autonomous iteration. In many high-impact domains, a more practical and valuable objective is to establish a process of co-improvement between human and agent systems. Human experts contribute domain knowledge, value judgments, contextual understanding, and accountability, whereas agent systems offer scalable exploration, persistent memory, automated experimentation, and efficient modification. The central research challenge is therefore to develop mechanisms that enable agents to improve through expert interaction while enhancing human capabilities through better evidence, more effective tools, and stronger decision support.

Future self-improving agent systems should learn how to identify situations that require expert involvement, determine the appropriate form of interaction, provide interpretable evidence for human review, and convert expert feedback into reusable improvement resources. Accordingly, the evaluation of agent systems should move beyond task-level performance metrics and examine whether human-agent collaboration leads to higher-quality decisions, greater transparency, and stronger generalization to previously unseen scenarios.

8. Conclusion

This survey studies self-improving agent systems, which autonomously transform experience and evaluation feedback into persistent updates to their own components. We begin by formalizing the problem and introducing a capability grading standard ranging from manual improvement to general RSI. Furthermore, we propose a unified research framework that analyzes these agent systems as a whole, characterizing both the scope of autonomous improvement and the improvement process. We organize existing work into a taxonomy covering self-improvement within the agent harness, data system, and trainer, as well as co-improvement across components. Additionally, we identify open problems that motivate progress toward more autonomous, reliable, and general self-improving agent systems. We hope that this survey lays the foundation for studying this emerging area and provides a roadmap for future research.

References

1. METR. Measuring AI Ability to Complete Long Tasks. https://metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/, 2025.
2. Kwa, T.; West, B.; Becker, J.; Deng, A.; Garcia, K.; Hasin, M.; Jawhar, S.; Kinniment, M.; Rush, N.; Arx, S.V.; et al. Measuring AI Ability to Complete Long Software Tasks, 2026, [arXiv:cs.AI/2503.14499].
3. Qwen Team. Qwen3.7: The Agent Frontier, 2026.
4. Xu, A.; Lin, B.; Xue, B.; Wang, B.; Xu, B.; Wu, B.; Zhang, B.; Lin, C.; Dong, C.; Ling, C.; et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348* **2026**.
5. Blakeman, A.; Thomas, A.; Jhunjhunwala, A.; Gupta, A.; Khattar, A.; Rajfer, A.; Renduchintala, A.; Asif, A.; Vavre, A.; Miranda, A.F.; et al. Nemotron 3 Ultra: Open, Efficient Mixture-of-Experts Hybrid Mamba-Transformer Model for Agentic Reasoning. *arXiv preprint arXiv:2606.15007* **2026**.
6. Jimenez, C.E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; Narasimhan, K. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?, 2024, [arXiv:cs.CL/2310.06770].
7. Deng, X.; Da, J.; Pan, E.; He, Y.Y.; Ide, C.; Garg, K.; Lauffer, N.; Park, A.; Pasari, N.; Rane, C.; et al. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?, 2025, [arXiv:cs.SE/2509.16941].
8. Merrill, M.A.; Shaw, A.G.; Carlini, N.; Li, B.; Raj, H.; Bercovich, I.; Shi, L.; Shin, J.Y.; Walshe, T.; Buchanan, E.K.; et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces, 2026, [arXiv:cs.SE/2601.11868].
9. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models, 2023, [arXiv:cs.CL/2210.03629].
10. Yang, J.; Jimenez, C.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.; Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* **2024**, *37*, 50528–50652.

 doi:10.20944/preprints202608.0051.v1

11. Anthropic. Claude Code. https://docs.anthropic.com/en/docs/claude-code, 2025. AI coding agent harness.
12. OpenAI. OpenAI Codex CLI. https://github.com/openai/codex, 2025. Lightweight coding agent running in the terminal.
13. Blakeman, A.; Grattafiori, A.; Basant, A.; Gupta, A.; Khattar, A.; Renduchintala, A.; Vavre, A.; Shukla, A.; Bercovich, A.; Ficek, A.; et al. NVIDIA Nemotron 3: Efficient and Open Intelligence. *arXiv preprint arXiv:2512.20856* **2025**.
14. Zhao, J.; Chen, G.; Meng, F.; Li, M.; Chen, J.; Xu, H.; Sun, Y.; Zhao, W.X.; Song, R.; Zhang, Y.; et al. Immersion in the github universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892* **2026**.
15. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* **2017**.
16. Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Kunc, Y.; et al. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300* **2024**.
17. Sheng, G.; Zhang, C.; Ye, Z.; Wu, X.; Zhang, W.; Zhang, R.; Peng, Y.; Lin, H.; Wu, C. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256* **2024**.
18. Zhu, Z.; Xie, C.; Lv, X.; slime Contributors. slime: An LLM post-training framework for RL Scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.
19. Zhang, L.; Chen, M.; Cao, R.; Chen, J.; Zhou, F.; Xu, Y.; Yang, J.; Ma, Z.; Chen, L.; Luo, C.; et al. MegaFlow: Large-Scale Distributed Orchestration System for the Agentic Era, 2026, [arXiv:cs.DC/2601.07526].
20. Wu, R.; Wang, X.; Mei, J.; Cai, P.; Fu, D.; Yang, C.; Wen, L.; Yang, X.; Shen, Y.; Wang, Y.; et al. EvolveR: Self-Evolving LLM Agents through an Experience-Driven Lifecycle, 2025, [arXiv:cs.CL/2510.16079].
21. Novikov, A.; Vu, N.; Eisenberger, M.; Dupont, E.; Huang, P.S.; Wagner, A.Z.; Shirobokov, S.; Kozlovskii, B.; Ruiz, F.J.R.; Mehrabian, A.; et al. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery, 2025, [arXiv:cs.AI/2506.13131].
22. Lin, J.; Liu, S.; Pan, C.; Lin, L.; Dou, S.; Huang, X.; Yan, H.; Han, Z.; Gui, T. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. *CoRR* **2026**, *abs/2604.25850*, [2604.25850]. https://doi.org/10.48550/ARXIV.2604.25850.
23. Zhang, J.; Zhao, B.; Yang, W.; Foerster, J.N.; Clune, J.; Jiang, M.; Devlin, S.; Shavrina, T. Hyperagents. *CoRR* **2026**, *abs/2603.19461*, [2603.19461]. https://doi.org/10.48550/ARXIV.2603.19461.
24. Good, I.J. Speculations Concerning the First Ultraintelligent Machine; Elsevier, 1966; Vol. 6, *Advances in Computers*, pp. 31–88. https://doi.org/https://doi.org/10.1016/S0065-2458(08)60418-0.
25. Schmidhuber, J. Gödel Machines: Fully Self-referential Optimal Universal Self-improvers. In *Artificial General Intelligence*; Goertzel, B.; Pennachin, C., Eds.; Cognitive Technologies, Springer, 2007; pp. 199–226. https://doi.org/10.1007/978-3-540-68677-4_7.
26. Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: language agents with verbal reinforcement learning. In Proceedings of the Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023; Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; Levine, S., Eds., 2023.
27. Zhao, A.; Huang, D.; Xu, Q.; Lin, M.; Liu, Y.; Huang, G. ExpeL: LLM Agents Are Experiential Learners. *CoRR* **2023**, *abs/2308.10144*, [2308.10144]. https://doi.org/10.48550/ARXIV.2308.10144.
28. Chen, S.; Lin, S.; Gu, X.; Shi, Y.; Lian, H.; Yun, L.; Chen, D.; Sun, W.; Cao, L.; Wang, Q. SWE-Exp: Experience-Driven Software Issue Resolution. *CoRR* **2025**, *abs/2507.23361*, [2507.23361]. https://doi.org/10.48550/ARXIV.2507.23361.
29. Ouyang, S.; Yan, J.; Hsu, I.; Chen, Y.; Jiang, K.; Wang, Z.; Han, R.; Le, L.T.; Daruki, S.; Tang, X.; et al. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. *CoRR* **2025**, *abs/2509.25140*, [2509.25140]. https://doi.org/10.48550/ARXIV.2509.25140.
30. Cai, Z.; Guo, X.; Pei, Y.; Feng, J.; Chen, J.; Zhang, Y.; Ma, W.; Wang, M.; Zhou, H. FLEX: Continuous Agent Evolution via Forward Learning from Experience. *CoRR* **2025**, *abs/2511.06449*, [2511.06449]. https://doi.org/10.48550/ARXIV.2511.06449.
31. Shen, M.; Zha, K.; He, Z.; Hong, Z.; Ouyang, S.; Ryu, J.J.; Sattigeri, P.; Diggavi, S.N.; Wornell, G.W. Decocted Experience Improves Test-Time Inference in LLM Agents. *CoRR* **2026**, *abs/2604.04373*, [2604.04373]. https://doi.org/10.48550/ARXIV.2604.04373.
32. Packer, C.; Fang, V.; Patil, S.G.; Lin, K.; Wooders, S.; Gonzalez, J.E. MemGPT: Towards LLMs as Operating Systems. *CoRR* **2023**, *abs/2310.08560*, [2310.08560]. https://doi.org/10.48550/ARXIV.2310.08560.

 doi:10.20944/preprints202608.0051.v1

33. Xu, W.; Liang, Z.; Mei, K.; Gao, H.; Tan, J.; Zhang, Y. A-MEM: Agentic Memory for LLM Agents. *CoRR* **2025**, *abs/2502.12110*, [2502.12110]. https://doi.org/10.48550/ARXIV.2502.12110.
34. Liu, J.; Su, Y.; Xia, P.; Han, S.; Zheng, Z.; Xie, C.; Ding, M.; Yao, H. SimpleMem: Efficient Lifelong Memory for LLM Agents. *CoRR* **2026**, *abs/2601.02553*, [2601.02553]. https://doi.org/10.48550/ARXIV.2601.02553.
35. Dai, Z.; Deng, S.; Guan, S.; Tian, Y.; Yao, X.; Yan, X.; Cheng, J. RecMem: Recurrence-based Memory Consolidation for Efficient and Effective Long-Running LLM Agents. *CoRR* **2026**, *abs/2605.16045*, [2605.16045]. https://doi.org/10.48550/ARXIV.2605.16045.
36. Jin, Y.; Zhang, S.; Wang, H.; Qin, L.; Zhang, Y.; Zhang, W. EXG: Self-Evolving Agents with Experience Graphs. *CoRR* **2026**, *abs/2605.17721*, [2605.17721]. https://doi.org/10.48550/ARXIV.2605.17721.
37. Fang, J.; Xu, B.; Wang, Z.; Cao, H.; Deng, X.; Dong, B.; Zhu, H.; Huang, R.; Yu, G.; Wei, Y.; et al. Rethinking Memory as Continuously Evolving Connectivity. *CoRR* **2026**, *abs/2605.28773*, [2605.28773]. https://doi.org/10.48550/ARXIV.2605.28773.
38. Ji, S.; Wu, B.; Wang, Z.; Xia, L.; Li, Q.; Wang, R.; Ding, W.; Zhu, Z.; Li, B.; Dai, G.; et al. Infini Memory: Maintainable Topic Documents for Long-Term LLM Agent Memory. 2026.
39. Fei, T.; Song, M.; Zheng, M.; Yu, X. Memory Beyond Recall: A Dual-Process Cognitive Memory System for Self-Evolving LLM Agents. 2026.
40. Ye, C.; Liu, Y.; Wang, Y.; Yu, H.; Zhao, Y.; Liu, G.; McAuley, J.J.; You, J. Auto-Dreamer: Learning Offline Memory Consolidation for Language Agents. *CoRR* **2026**, *abs/2605.20616*, [2605.20616]. https://doi.org/10.48550/ARXIV.2605.20616.
41. Zhang, Y.; Wu, Y.; Yu, Y.; Wu, Q.; Wang, H. Live-Evo: Online Evolution of Agentic Memory from Continuous Feedback. *CoRR* **2026**, *abs/2602.02369*, [2602.02369]. https://doi.org/10.48550/ARXIV.2602.02369.
42. Liao, J.; Shi, H.; Zhou, R.; Wang, J.; Zhang, S.; Zhang, W.; Wen, Y.; Li, Z.; Xiong, F.; Tang, B.; et al. MemQ: Integrating Q-Learning into Self-Evolving Memory Agents over Provenance DAGs. *CoRR* **2026**, *abs/2605.08374*, [2605.08374]. https://doi.org/10.48550/ARXIV.2605.08374.
43. Wang, X.; Mao, W.; Wu, J.; Wang, X.; He, X. R^2-Mem: Reflective Experience for Memory Search. *CoRR* **2026**, *abs/2605.13486*, [2605.13486]. https://doi.org/10.48550/ARXIV.2605.13486.
44. Zhang, Y.; Li, Y.; Payani, A.; Wang, L. AdaMEM: Test-Time Adaptive Memory for Language Agents. 2026.
45. Kim, K.; Kang, M.; Kim, T.; Yang, Y.; Ren, M.; Hwang, S.J. Memory Transfer Learning: How Memories are Transferred Across Domains in Coding Agents. *CoRR* **2026**, *abs/2604.14004*, [2604.14004]. https://doi.org/10.48550/ARXIV.2604.14004.
46. Cheng, Y.; Zhou, J.; Hu, Y.; Chen, Y.; Zhou, H.; Chen, M.; Zhang, Z.; Shao, K.; Xie, Y.; Yin, Z. TAME: A Trustworthy Test-Time Evolution of Agent Memory with Systematic Benchmarking. *CoRR* **2026**, *abs/2602.03224*, [2602.03224]. https://doi.org/10.48550/ARXIV.2602.03224.
47. Wang, S.; Brahma, D.; Henao, R. SAGE: A Novelty Gate for Efficient Memory Evolution in Agentic LLMs. *CoRR* **2026**, *abs/2605.30711*, [2605.30711]. https://doi.org/10.48550/ARXIV.2605.30711.
48. Song, Y.; Xin, Q. D-MEM: Dopamine-Gated Agentic Memory via Reward Prediction Error Routing. *CoRR* **2026**, *abs/2603.14597*, [2603.14597]. https://doi.org/10.48550/ARXIV.2603.14597.
49. Zhang, G.; Ren, H.; Zhan, C.; Zhou, Z.; Wang, J.; Zhu, H.; Zhou, W.; Yan, S. MemEvolve: Meta-Evolution of Agent Memory Systems. *CoRR* **2025**, *abs/2512.18746*, [2512.18746]. https://doi.org/10.48550/ARXIV.2512.18746.
50. Xiong, Y.; Hu, S.; Clune, J. Learning to Continually Learn via Meta-learning Agentic Memory Designs. *CoRR* **2026**, *abs/2602.07755*, [2602.07755]. https://doi.org/10.48550/ARXIV.2602.07755.
51. Pan, W.; Liu, S.; Zhou, X.; Zhang, S.; Shi, W.; Xu, M.; Jia, X. M*: Every Task Deserves Its Own Memory Harness. *CoRR* **2026**, *abs/2604.11811*, [2604.11811]. https://doi.org/10.48550/ARXIV.2604.11811.
52. Liu, J.; Ye, X.; Xia, P.; Zheng, Z.; Xie, C.; Ding, M.; Yao, H. EvolveMem:Self-Evolving Memory Architecture via AutoResearch for LLM Agents. *CoRR* **2026**, *abs/2605.13941*, [2605.13941]. https://doi.org/10.48550/ARXIV.2605.13941.
53. Liu, Q.; Wang, G.; Wu, W.; Huang, J.; Tao, X.; Song, D.; Zhou, J.; He, L. MemPro: Agentic Memory Systems as Evolvable Programs. 2026.
54. Zhang, H.; Long, Q.; Bao, J.; Feng, T.; Zhang, W.; Yue, H.; Wang, W. MemSkill: Learning and Evolving Memory Skills for Self-Evolving Agents. *CoRR* **2026**, *abs/2602.02474*, [2602.02474]. https://doi.org/10.48550/ARXIV.2602.02474.
55. Yang, Y.; Liu, T.; Zhu, W.B.; Shi, T.; Song, L.; Jia, R. Self-Evolving LLM Memory Extraction Across Heterogeneous Tasks. *CoRR* **2026**, *abs/2604.11610*, [2604.11610]. https://doi.org/10.48550/ARXIV.2604.11610.

 doi:10.20944/preprints202608.0051.v1

56. Qian, C.; Han, C.; Fung, Y.R.; Qin, Y.; Liu, Z.; Ji, H. CREATOR: Disentangling Abstract and Concrete Reasonings of Large Language Models through Tool Creation. *CoRR* **2023**, *abs/2305.14318*, [2305.14318]. https://doi.org/10.48550/ARXIV.2305.14318.
57. Cai, T.; Wang, X.; Ma, T.; Chen, X.; Zhou, D. Large Language Models as Tool Makers. *CoRR* **2023**, *abs/2305.17126*, [2305.17126]. https://doi.org/10.48550/ARXIV.2305.17126.
58. Ding, H.; Tao, S.; Pang, L.; Wei, Z.; Gao, J.; Ding, B.; Shen, H.; Cheng, X. ToolCoder: A Systematic Code-Empowered Tool Learning Framework for Large Language Models. *CoRR* **2025**, *abs/2502.11404*, [2502.11404]. https://doi.org/10.48550/ARXIV.2502.11404.
59. Liu, X.; Yin, D.; Wu, Z.; Feng, Y. RefTool: Enhancing Model Reasoning with Reference-Guided Tool Creation. *CoRR* **2025**, *abs/2505.21413*, [2505.21413]. https://doi.org/10.48550/ARXIV.2505.21413.
60. Zheng, B.; Fatemi, M.Y.; Jin, X.; Wang, Z.Z.; Gandhi, A.; Song, Y.; Gu, Y.; Srinivasa, J.; Liu, G.; Neubig, G.; et al. SkillWeaver: Web Agents can Self-Improve by Discovering and Honing Skills. *CoRR* **2025**, *abs/2504.07079*, [2504.07079]. https://doi.org/10.48550/ARXIV.2504.07079.
61. Wang, C.; Yu, Z.; Xie, X.; Yao, W.; Fang, R.; Qiao, S.; Cao, K.; Zheng, G.; Qi, X.; Zhang, P.; et al. SkillX: Automatically Constructing Skill Knowledge Bases for Agents. *CoRR* **2026**, *abs/2604.04804*, [2604.04804]. https://doi.org/10.48550/ARXIV.2604.04804.
62. Zhang, Y.; Han, X.; Jiang, X.; Wang, R. Workflow-to-Skill: Skill Creation via Routing-Workflow-Semantics-Attachments Decomposition. 2026.
63. Xiao, C.; Jiao, Z.; Wang, S.; Wang, W.; Zhao, B.; Wei, H.; Zhang, L.; Qu, L. Socratic-SWE: Self-Evolving Coding Agents via Trace-Derived Agent Skills. 2026.
64. Pan, Q.; Yang, Y.; Li, J.; Zhou, J.; Chen, K.; Li, X.; Chen, Q.; He, L. Anything2Skill: Compiling External Knowledge into Reusable Skills for Agents. 2026.
65. Yan, Z.; Song, D.; Zhang, H.; Liang, W.; Zhang, Y.; Dai, Y.; He, L.; Yu, P.S.; Xu, R.; Li, X.; et al. OpenSkill: Open-World Self-Evolution for LLM Agents. 2026.
66. Shen, S.; Cheng, W.; Ma, M.; Turcan, A.; Zhang, M.J.; Ma, J. SKILLFOUNDRY: Building Self-Evolving Agent Skill Libraries from Heterogeneous Scientific Resources. *CoRR* **2026**, *abs/2604.03964*, [2604.03964]. https://doi.org/10.48550/ARXIV.2604.03964.
67. Qiu, Z.; Song, K.; Tang, S.; Qiao, S.; Liang, L.; Chen, H.; Deng, S. Unsupervised Skill Discovery for Agentic Data Analysis. 2026.
68. Lin, H.; Li, P.; Song, J.; Jiang, F.; Zhang, T. MUSE-Autoskill: Self-Evolving Agents via Skill Creation, Memory, Management, and Evaluation. *CoRR* **2026**, *abs/2605.27366*, [2605.27366]. https://doi.org/10.48550/ARXIV.2605.27366.
69. Ouyang, S.; Yan, J.; Chen, Y.; Han, R.; Wang, Z.; Mishra, B.D.; Meng, R.; Li, C.; Jiao, Y.; Zha, K.; et al. SkillOS: Learning Skill Curation for Self-Evolving Agents. *CoRR* **2026**, *abs/2605.06614*, [2605.06614]. https://doi.org/10.48550/ARXIV.2605.06614.
70. Ma, Z.; Yang, S.; Ji, Y.; Wang, X.; Wang, Y.; Hu, Y.; Huang, T.; Chu, X. SkillClaw: Let Skills Evolve Collectively with Agentic Evolver. *CoRR* **2026**, *abs/2604.08377*, [2604.08377]. https://doi.org/10.48550/ARXIV.2604.08377.
71. Yue, M.; Liu, Z.; Yang, L.; Zhang, J.; Liu, Z.; Chen, H.; Yao, Z.; Savarese, S.; Xiong, C.; Heinecke, S.; et al. ToolLibGen: Scalable Automatic Tool Creation and Aggregation for LLM Reasoning. *CoRR* **2025**, *abs/2510.07768*, [2510.07768]. https://doi.org/10.48550/ARXIV.2510.07768.
72. Yang, Y.; Gong, Z.; Huang, W.; Yang, Q.; Zhou, Z.; Huang, Z.; Li, Y.; Gao, X.; Dai, Q.; Liu, B.; et al. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *CoRR* **2026**, *abs/2605.23904*, [2605.23904]. https://doi.org/10.48550/ARXIV.2605.23904.
73. Wang, Y.; Zhou, Y.; Liang, Y.; Zhang, C.; Liu, F.; Zhou, J.; Yao, H. Not All Skills Help: Measuring and Repairing Agent Knowledge. 2026.
74. Zhang, Q.; Feng, Z.; Shi, X.; Hu, X.; Liu, C.; Xie, P.; Wang, X.; Ye, J.; Hooi, B.; Wang, H.; et al. SkillComposer: Learning to Evolve Agent Skills for Specification and Generalization. 2026.
75. Shi, H.; Yuan, X.; Liu, B. Evolving Programmatic Skill Networks. *CoRR* **2026**, *abs/2601.03509*, [2601.03509]. https://doi.org/10.48550/ARXIV.2601.03509.
76. Gautam, S.; Radhakrishna, A.; Gulwani, S. SkillAxe: Sharpening LLM-Authored Agent Skills Through Evaluation-Guided Self-Refinement. 2026.
77. Gao, H.; Chen, H.; Wang, C.; Guo, S.; Pang, L.; Liu, Z.; Shen, H.; Cheng, X. SkillAudit: Ground-Truth-Free Skill Evolution via Paired Trajectory Auditing. 2026.

 doi:10.20944/preprints202608.0051.v1

78. Ma, Y.; Huang, Y.; Bao, H.; Zhuang, H.; Shukla, S.; Galley, M.; Zhang, X.; Feuerriegel, S. SkillGen: Verified Inference-Time Agent Skill Synthesis. *CoRR* **2026**, *abs/2605.10999*, [2605.10999]. https://doi.org/10.48550/ARXIV.2605.10999.
79. Zhang, H.; Fan, S.; Zou, H.P.; Chen, Y.; Wang, Z.; Zhou, J.; Li, C.; Huang, W.; Yao, Y.; Zheng, K.; et al. CoEvoSkills: Self-Evolving Agent Skills via Co-Evolutionary Verification. *CoRR* **2026**, *abs/2604.01687*, [2604.01687]. https://doi.org/10.48550/ARXIV.2604.01687.
80. Yang, Y.; Bhatt, N.P.; Wang, K.; Tetteh, S.; Wang, Z.; Topcu, U. VASO: Formally Verifiable Self-Evolving Skills for Physical AI Agents. 2026.
81. Lu, J.; Kong, Z.; Wang, Y.; Fu, R.; Wan, H.; Yang, C.; Lou, W.; Sun, H.; Wang, L.; Jiang, Y.; et al. Beyond Static Tools: Test-Time Tool Evolution for Scientific Reasoning. *CoRR* **2026**, *abs/2601.07641*, [2601.07641]. https://doi.org/10.48550/ARXIV.2601.07641.
82. Wei, S.; Min, H.S.; Dong, X.; Lin, X.; Cui, S.; Jiang, B.; Dai, Z.; Kuang, K.; Xu, G.; Wu, F.; et al. MetaForge: A Self-Evolving Multimodal Agent that Retrieves, Adapts, and Forges Tools On Demand. 2026.
83. Wei, Y.; Huang, Z.; Lu, S.; Qian, J.; Wang, Q.; Wu, C.; He, L. SkillSmith: Co-Evolving Skills and Tools for Self-Improving Agent Systems. 2026.
84. Wang, J.; Yan, Q.; Wang, Y.; Tian, Y.; Mishra, S.S.; Xu, Z.; Gandhi, M.; Xu, P.; Cheong, L.L. Reinforcement Learning for Self-Improving Agent with Skill Library. *CoRR* **2025**, *abs/2512.17102*, [2512.17102]. https://doi.org/10.48550/ARXIV.2512.17102.
85. Wong, S.; Qi, Z.; Wang, Z.; Hu, N.; Lin, S.; Ge, J.; Gao, E.; Chen, W.; Du, Y.; Yu, M.; et al. Confucius Code Agent: Scalable Agent Scaffolding for Real-World Codebases. *CoRR* **2025**, *abs/2512.10398*, [2512.10398]. https://doi.org/10.48550/ARXIV.2512.10398.
86. Yang, C.; Wang, X.; Lu, Y.; Liu, H.; Le, Q.V.; Zhou, D.; Chen, X. Large Language Models as Optimizers. *CoRR* **2023**, *abs/2309.03409*, [2309.03409]. https://doi.org/10.48550/ARXIV.2309.03409.
87. Fernando, C.; Banarse, D.; Michalewski, H.; Osindero, S.; Rocktäschel, T. Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution. *CoRR* **2023**, *abs/2309.16797*, [2309.16797]. https://doi.org/10.48550/ARXIV.2309.16797.
88. Xiang, J.; Zhang, J.; Yu, Z.; Teng, F.; Tu, J.; Liang, X.; Hong, S.; Wu, C.; Luo, Y. Self-Supervised Prompt Optimization. *CoRR* **2025**, *abs/2502.06855*, [2502.06855]. https://doi.org/10.48550/ARXIV.2502.06855.
89. Peng, D.; Zhou, Y.; Chen, Q.; Liu, J.; Chen, J.; Qin, L. DLPO: Towards a Robust, Efficient, and Generalizable Prompt Optimization Framework from a Deep-Learning Perspective. *CoRR* **2025**, *abs/2503.13413*, [2503.13413]. https://doi.org/10.48550/ARXIV.2503.13413.
90. Lin, Z.J.; Letham, B.; Dooley, S.; Balandat, M.; Bakshy, E. Embedding by Elicitation: Dynamic Representations for Bayesian Optimization of System Prompts. *CoRR* **2026**, *abs/2605.19093*, [2605.19093]. https://doi.org/10.48550/ARXIV.2605.19093.
91. Singhal, R.; Tambwekar, P.; Maamari, K. PrefPO: Pairwise Preference Prompt Optimization. *CoRR* **2026**, *abs/2603.19311*, [2603.19311]. https://doi.org/10.48550/ARXIV.2603.19311.
92. Wang, F.; Si, S.; Hsieh, C.J.; Dhillon, I.S. APEX: Automated Prompt Engineering eXpert with Dynamic Data Selection. 2026.
93. Kang, E.H.; Yoganarasimhan, H. Bayesian Optimization in Language Space: An Eval-Efficient AI Self-Improvement Framework. *CoRR* **2025**, *abs/2511.12063*, [2511.12063]. https://doi.org/10.48550/ARXIV.2511.12063.
94. Lee, Y.; Boen, J.; Finn, C. Feedback Descent: Open-Ended Text Optimization via Pairwise Comparison. *ArXiv* **2025**, *abs/2511.07919*.
95. Lu, M.; Feng, C.; Han, H.; Lu, G.; Sun, Y.; Ding, X.; Long, S.; Li, F.; Motwani, T. SPEAR: Code-Augmented Agentic Prompt Optimization. *CoRR* **2026**, *abs/2605.26275*, [2605.26275]. https://doi.org/10.48550/ARXIV.2605.26275.
96. Fernandes, R.C.; Fehring, L.; Eimer, T.; Lindauer, M.; Feurer, M. Environment-Grounded Automated Prompt Optimization for LLM Game Agents. *arXiv preprint arXiv:2606.17838* **2026**.
97. Khattab, O.; Singhvi, A.; Maheshwari, P.; Zhang, Z.; Santhanam, K.; Vardhamanan, S.; Haq, S.; Sharma, A.; Joshi, T.T.; Moazam, H.; et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *CoRR* **2023**, *abs/2310.03714*, [2310.03714]. https://doi.org/10.48550/ARXIV.2310.03714.
98. Opsahl-Ong, K.; Ryan, M.J.; Purtell, J.; Broman, D.; Potts, C.; Zaharia, M.; Khattab, O. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. *CoRR* **2024**, *abs/2406.11695*, [2406.11695]. https://doi.org/10.48550/ARXIV.2406.11695.

 doi:10.20944/preprints202608.0051.v1

99. Spiess, C.; Vaziri, M.; Mandel, L.; Hirzel, M. AutoPDL: Automatic Prompt Optimization for LLM Agents. *CoRR* **2025**, *abs/2504.04365*, [2504.04365]. https://doi.org/10.48550/ARXIV.2504.04365.
100. Lin, S.; Hua, W.; Li, L.; Wang, Z.; Zhang, Y. ADO: Automatic Data Optimization for Inputs in LLM Prompts. *CoRR* **2025**, *abs/2502.11436*, [2502.11436]. https://doi.org/10.48550/ARXIV.2502.11436.
101. Shankaranarayanan, A.; Venkataraman, A.N.; Nikolakopoulos.; Kumaraswamy, V.S.; Zhang, T.; Chander, S.; Saboo, R.R.; Khan, S.A. A FRAMEWORK FOR PROMPT OPTIMIZATION AND TRANSLATION A CROSS FOUNDATION MODELS.
102. Yüksekgönül, M.; Bianchi, F.; Boen, J.; Liu, S.; Huang, Z.; Guestrin, C.; Zou, J. TextGrad: Automatic "Differentiation" via Text. *CoRR* **2024**, *abs/2406.07496*, [2406.07496]. https://doi.org/10.48550/ARXIV.2406.07496.
103. Agrawal, L.A.; Tan, S.; Soylu, D.; Ziems, N.; Khare, R.; Opsahl-Ong, K.; Singhvi, A.; Shandilya, H.; Ryan, M.J.; Jiang, M.; et al. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *ArXiv* **2025**, *abs/2507.19457*.
104. Chen, M.; Deng, W.; Zou, J.; Yu, H.; Li, X. Textual Equilibrium Propagation for Deep Compound AI Systems. *CoRR* **2026**, *abs/2601.21064*, [2601.21064]. https://doi.org/10.48550/ARXIV.2601.21064.
105. Rishav, R.; Pujari, P.; Rastogi, P. ContraPrompt: Contrastive Prompt Optimization via Dyadic Reasoning Trace Analysis. *CoRR* **2026**, *abs/2604.17937*, [2604.17937]. https://doi.org/10.48550/ARXIV.2604.17937.
106. Zhu, T.; Yao, T.; Kuwarananchoren, K.; Singh, A.; Lai, Y.; Mohan, D.A.; Bhargava, S. Graph-based Target Back-Propagation for Context Adaptation in Multi-LLM Agentic Systems. 2026.
107. Li, W.; Song, Y.; Zhao, M.; Jin, B.; Li, W. Unifying Temporal and Structural Credit Assignment in LLM-Based Multi-Agent Prompt Optimization. *CoRR* **2026**, *abs/2605.30227*, [2605.30227]. https://doi.org/10.48550/ARXIV.2605.30227.
108. Xu, W.; Liu, S.; Wang, M. EEVEE: Towards Test-time Prompt Learning in the Real World for Self-Improving Agents. 2026.
109. Li, H.; He, R.; Zhang, Q.; Ji, C.; Mang, Q.; Chen, X.; Agrawal, L.A.; Liao, W.; Yang, E.; Cheung, A.; et al. Combee: Scaling Prompt Learning for Self-Improving Language Model Agents. *CoRR* **2026**, *abs/2604.04247*, [2604.04247]. https://doi.org/10.48550/ARXIV.2604.04247.
110. Suzgun, M.; Yüksekgönül, M.; Bianchi, F.; Jurafsky, D.; Zou, J. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. *CoRR* **2025**, *abs/2504.07952*, [2504.07952]. https://doi.org/10.48550/ARXIV.2504.07952.
111. Zhang, Q.; Hu, C.; Upasani, S.; Ma, B.; Hong, F.; Kamanuru, V.; Rainton, J.; Wu, C.; Ji, M.; Li, H.; et al. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *CoRR* **2025**, *abs/2510.04618*, [2510.04618]. https://doi.org/10.48550/ARXIV.2510.04618.
112. Pei, Z.; Zhen, H.; Kai, S.; Pan, S.J.; Wang, Y.; Yuan, M.; Yu, B. SCOPE: Prompt Evolution for Enhancing Agent Effectiveness. *CoRR* **2025**, *abs/2512.15374*, [2512.15374]. https://doi.org/10.48550/ARXIV.2512.15374.
113. Vassilyev, N.; Berrios, W.; Zhang, R.; Han, B.; Kiela, D.; Mehri, S. Reflective Context Learning: Studying the Optimization Primitives of Context Space. *CoRR* **2026**, *abs/2604.03189*, [2604.03189]. https://doi.org/10.48550/ARXIV.2604.03189.
114. Zhu, Z.; Hu, Y.; Dai, Y.; Fang, J.; Jiang, C.; Hu, S.; Zhao, Y. Unified Context Evolution for LLM Agents. 2026.
115. Parashar, J.; Bhandarkar, S. KACE: Knowledge-Adaptive Context Engineering for Mathematical Reasoning. 2026.
116. Huang, Z.; Kuncoro, A.; Feng, Q.; Shen, J.; Dery, L.M.; Szlam, A.; Ranzato, M. Context Training with Active Information Seeking. *CoRR* **2026**, *abs/2605.13050*, [2605.13050]. https://doi.org/10.48550/ARXIV.2605.13050.
117. Xie, Y.; Wang, K.; Cheng, B.; Yao, J.; Sha, Z.; Duffy, A.; Xi, Y.; Mei, H.; Tan, C.; Wei, C.; et al. MEMO: Memory-Augmented Model Context Optimization for Robust Multi-Turn Multi-Agent LLM Games. *CoRR* **2026**, *abs/2603.09022*, [2603.09022]. https://doi.org/10.48550/ARXIV.2603.09022.
118. Wu, Y.; Long, W.; Nguyen, C.T.; Wang, X.; et al. Contrastive Self-Refinement for Low-Cost Adaptation in Real-World Text-to-SQL.
119. Zha, J.; Wang, J.; Zhou, C.; Song, X. Trace2Policy: From Expert Behavior Traces to Self-Evolving Decision Agents. 2026.
120. Tao, W.; Wu, H.; Wong, W.F. SePO: Self-Evolving Prompt Agent for System Prompt Optimization. 2026.
121. Chen, X.; Xu, C.; Wang, Y.; Liu, B.; Yao, Z.; He, Y. Learning to Self-Evolve. *CoRR* **2026**, *abs/2603.18620*, [2603.18620]. https://doi.org/10.48550/ARXIV.2603.18620.
122. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.L.; Cao, Y.; Narasimhan, K. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *CoRR* **2023**, *abs/2305.10601*, [2305.10601]. https://doi.org/10.48550/ARXIV.2305.10601.

 doi:10.20944/preprints202608.0051.v1

123. Zhou, A.; Yan, K.; Shlapentokh-Rothman, M.; Wang, H.; Wang, Y. Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models. *CoRR* **2023**, *abs/2310.04406*, [2310.04406]. https://doi.org/10.48550/ARXIV.2310.04406.
124. Zhu, K.; Li, H.; Wu, S.; Xing, T.; Ma, D.; Tang, X.; Liu, M.; Yang, J.; Liu, J.; Jiang, Y.E.; et al. Scaling Test-time Compute for LLM Agents. *CoRR* **2025**, *abs/2506.12928*, [2506.12928]. https://doi.org/10.48550/ARXIV.2506.12928.
125. Antoniades, A.; Örwall, A.; Zhang, K.; Xie, Y.; Goyal, A.; Wang, W.Y. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. *CoRR* **2024**, *abs/2410.20285*, [2410.20285]. https://doi.org/10.48550/ARXIV.2410.20285.
126. Kim, J.; Yang, W.; Niu, K.; Zhang, H.; Zhu, Y.; Helenowski, E.; Silva, R.; Chen, Z.; Iyer, S.; Zaheer, M.; et al. Scaling Test-Time Compute for Agentic Coding. *CoRR* **2026**, *abs/2604.16529*, [2604.16529]. https://doi.org/10.48550/ARXIV.2604.16529.
127. Zeng, G.; Shen, M.; Chen, D.; Qi, Z.; Das, S.; Gutfreund, D.; Cox, D.; Wornell, G.W.; Lu, W.; Hong, Z.; et al. Satori-SWE: Evolutionary Test-Time Scaling for Sample-Efficient Software Engineering. *CoRR* **2025**, *abs/2505.23604*, [2505.23604]. https://doi.org/10.48550/ARXIV.2505.23604.
128. Lin, J.; Guo, Y.; Han, Y.; Hu, S.; Ni, Z.; Wang, L.; Chen, M.; Liu, H.; Chen, R.; He, Y.; et al. SE-Agent: Self-Evolution Trajectory Optimization in Multi-Step Reasoning with LLM-Based Agents. *CoRR* **2025**, *abs/2508.02085*, [2508.02085]. https://doi.org/10.48550/ARXIV.2508.02085.
129. Tan, D.Y.Y.; Chin, K.; Zhang, J. AgentGA: Evolving Code Solutions in Agent-Seed Space. *CoRR* **2026**, *abs/2604.14655*, [2604.14655]. https://doi.org/10.48550/ARXIV.2604.14655.
130. Fattha, A.D.; Chua, K.Y.; Jiang, L.; Wynter, L. Exploration Structure in LLM Agents for Multi-File Change Localization. 2026.
131. Zhang, S.; Wang, M.; Shi, Y.; Wang, Y.; Gu, X.; Yao, Y.; Fu, R.; Fu, S. FastContext: Training Efficient Repository Explorer for Coding Agents. 2026.
132. Han, H.; Xie, J.; Ma, X.; Zhu, W.; Zhang, Z.; Long, Z.; Chen, H.; Ye, Q. SWE-TRACE: Optimizing Long-Horizon SWE Agents Through Rubric Process Reward Models and Heuristic Test-Time Scaling. *CoRR* **2026**, *abs/2604.14820*, [2604.14820]. https://doi.org/10.48550/ARXIV.2604.14820.
133. Mao, C.; Lei, Y.; Wei, Z.; Liang, M.; Wang, Z.; Xu, J.; Chen, D.; Jiang, W.; Li, Y. EGSS: Entropy-guided Stepwise Scaling for Reliable Software Engineering. *CoRR* **2026**, *abs/2602.05242*, [2602.05242]. https://doi.org/10.48550/ARXIV.2602.05242.
134. Tan, B.; Deng, H.; Zhang, J.; Xu, J.; He, P.; Sun, Y. SWE-Manager: Selecting and Synthesizing Golden Proposals Before Coding. *CoRR* **2026**, *abs/2601.22956*, [2601.22956]. https://doi.org/10.48550/ARXIV.2601.22956.
135. Liu, M.; Chen, Z.; Pei, Z.; Wang, Z.; Wang, Y.; Zheng, Z. Architecture-Aware Multi-Design Generation for Repository-Level Feature Addition. *CoRR* **2026**, *abs/2603.01814*, [2603.01814]. https://doi.org/10.48550/ARXIV.2603.01814.
136. Ding, Y.; Zhang, L. SWE-Replay: Efficient Test-Time Scaling for Software Engineering Agents. *CoRR* **2026**, *abs/2601.22129*, [2601.22129]. https://doi.org/10.48550/ARXIV.2601.22129.
137. Zhang, T.; Popa, A.; Xu, Y.; Song, R.; Dimitriadis, D. PIVOT: Bridging Planning and Execution in LLM Agents via Trajectory Refinement. *CoRR* **2026**, *abs/2605.11225*, [2605.11225]. https://doi.org/10.48550/ARXIV.2605.11225.
138. Pan, R.; Wang, J.; Zhang, Q.; Zhu, Y.; Wu, L.; Yang, Z.; Zhang, Y.; Zhang, L.; Zhang, H. Persistent Cross-Attempt State Optimization for Repository-Level Code Generation. *CoRR* **2026**, *abs/2604.03632*, [2604.03632]. https://doi.org/10.48550/ARXIV.2604.03632.
139. Zhou, Z.; Cao, C.; Feng, X.; Li, X.; Li, Z.; Lu, X.; Yao, J.; Huang, W.; Cheng, T.; Zhang, J.; et al. AlphaApollo: A System for Deep Agentic Reasoning. 2025.
140. Chen, P.B.; Zhang, Y.; Roth, D.; Madden, S.; Andreas, J.; Cafarella, M.J. Log-Augmented Generation: Scaling Test-Time Reasoning with Reusable Computation. *CoRR* **2025**, *abs/2505.14398*, [2505.14398]. https://doi.org/10.48550/ARXIV.2505.14398.
141. Tran, Q.M.; Huang, Z.; Zhang, W.; Han, B.; Yatani, K.; Sugiyama, M.; Liu, T. Bifrost: Steering Strategic Trajectories to Bridge Contextual Gaps for Self-Improving Agents. *CoRR* **2026**, *abs/2602.05810*, [2602.05810]. https://doi.org/10.48550/ARXIV.2602.05810.
142. Yang, X.; Zhou, J.; Pacheco, M.; Zhu, W.; He, P.; Wang, S.; Liu, K.; Pan, R. Lingxi: Repository-Level Issue Resolution Framework Enhanced by Procedural Knowledge Guided Scaling. *CoRR* **2025**, *abs/2510.11838*, [2510.11838]. https://doi.org/10.48550/ARXIV.2510.11838.

 doi:10.20944/preprints202608.0051.v1

143. Ma, R.; Jiang, Y.; Zhang, S.; Ma, Z.; Feng, Y.; Ng, V.; Wang, Z.; Yue, X.; Li, C.; Lu, L. FailureMem: A Failure-Aware Multimodal Framework for Autonomous Software Repair. *CoRR* **2026**, *abs/2603.17826*, [2603.17826]. https://doi.org/10.48550/ARXIV.2603.17826.
144. Hu, H.; Xie, G.; Zhang, Q.; Liu, J.; Yu, S.; Fang, C.; Chen, Z.; Xiao, L. EvoRepair: Enhancing Vulnerability Repair Agents Through Experience-Based Self-Evolution. *CoRR* **2026**, *abs/2605.30105*, [2605.30105]. https://doi.org/10.48550/ARXIV.2605.30105.
145. Yu, S.; Chong, D.; Nandi, A.; Soylu, D.; Sun, J.; Manning, C.D.; Shi, W. Shepherd: A Runtime Substrate Empowering Meta-Agents with a Formalized Execution Trace. *CoRR* **2026**, *abs/2605.10913*, [2605.10913]. https://doi.org/10.48550/ARXIV.2605.10913.
146. Dong, Y.; He, J.; Hou, Y.; Du, D.; Xu, Z.; Yu, S.; Xia, Y.; Chen, H. DeltaBox: Scaling Stateful AI Agents with Millisecond-Level Sandbox Checkpoint/Rollback. *CoRR* **2026**, *abs/2605.22781*, [2605.22781]. https://doi.org/10.48550/ARXIV.2605.22781.
147. Zhang, X.; Wang, D.; Xu, K.; Zhu, Q.; Che, W. Scaling Laws for Agent Harnesses via Effective Feedback Compute. *CoRR* **2026**, *abs/2605.29682*, [2605.29682]. https://doi.org/10.48550/ARXIV.2605.29682.
148. Shang, Y.; Li, Y.; Zhao, K.; Ma, L.; Liu, J.; Xu, F.; Li, Y. AgentSquare: Automatic LLM Agent Search in Modular Design Space. *CoRR* **2024**, *abs/2410.06153*, [2410.06153]. https://doi.org/10.48550/ARXIV.2410.06153.
149. Li, Y.; Li, L.; Wu, Z.; Liao, Q.; Hao, J.; Shao, K.; Xu, F.; Li, Y. AgentSwift: Efficient LLM Agent Design via Value-guided Hierarchical Search. *CoRR* **2025**, *abs/2506.06017*, [2506.06017]. https://doi.org/10.48550/ARXIV.2506.06017.
150. Chen, J.; Shen, J.; Kang, H.; Hong, Z.; Jiang, Q.; Bose, S.; Zhang, Y.; Leng, L.; Vyas, A.K.; Mao, L.; et al. AgentSpec: Understanding Embodied Agent Scaffolds Through Controlled Composition. 2026.
151. Chen, G.; Dong, S.; Shu, Y.; Zhang, G.; Sesay, J.; Karlsson, B.F.; Fu, J.; Shi, Y. AutoAgents: A Framework for Automatic Agent Generation. *CoRR* **2023**, *abs/2309.17288*, [2309.17288]. https://doi.org/10.48550/ARXIV.2309.17288.
152. Yuan, S.; Song, K.; Chen, J.; Tan, X.; Li, D.; Yang, D. EvoAgent: Towards Automatic Multi-Agent Generation via Evolutionary Algorithms. *CoRR* **2024**, *abs/2406.14228*, [2406.14228]. https://doi.org/10.48550/ARXIV.2406.14228.
153. Hu, S.; Lu, C.; Clune, J. Automated Design of Agentic Systems. *CoRR* **2024**, *abs/2408.08435*, [2408.08435]. https://doi.org/10.48550/ARXIV.2408.08435.
154. Ye, R.; Tang, S.; Ge, R.; Du, Y.; Yin, Z.; Chen, S.; Shao, J. MAS-GPT: Training LLMs to Build LLM-based Multi-Agent Systems. *CoRR* **2025**, *abs/2503.03686*, [2503.03686]. https://doi.org/10.48550/ARXIV.2503.03686.
155. Xu, A.; Tai, Y. Meta-Agent: From Task Descriptions to Verified Multi-Agent Systems. *CoRR* **2026**, *abs/2605.25233*, [2605.25233]. https://doi.org/10.48550/ARXIV.2605.25233.
156. Chen, X.; Liu, Y.; Wei, H.; Ding, K. LEMON: Learning Executable Multi-Agent Orchestration via Counterfactual Reinforcement Learning. *CoRR* **2026**, *abs/2605.14483*, [2605.14483]. https://doi.org/10.48550/ARXIV.2605.14483.
157. Feng, Y.; Luo, H.; Lin, Z.; Sun, Y.; Wei, P.; Hsieh, L.B.; Luu, A.T. OrchMAS: Orchestrated Reasoning with Multi Collaborative Heterogeneous Scientific Expert Structured Agents. *CoRR* **2026**, *abs/2603.03005*, [2603.03005]. https://doi.org/10.48550/ARXIV.2603.03005.
158. Hu, Y.; Zhang, Y.; Trager, M.; Zhang, Y.E.; Yang, S.; Xia, W.; Soatto, S.. Evolutionary Generation of Multi-Agent Systems. *ArXiv* **2026**, *abs/2602.06511*.
159. Zhang, Y.; Xu, T.; Dai, S.; Shao, Z.; Wu, Q.; Wang, H. EVOCHAMBER: Test-Time Co-evolution of Multi-Agent System at Individual, Team, and Population Scales. *CoRR* **2026**, *abs/2605.11136*, [2605.11136]. https://doi.org/10.48550/ARXIV.2605.11136.
160. Zhang, G.; Niu, L.; Fang, J.; Wang, K.; Bai, L.; Wang, X. Multi-agent Architecture Search via Agentic Supernet. *CoRR* **2025**, *abs/2502.04180*, [2502.04180]. https://doi.org/10.48550/ARXIV.2502.04180.
161. Ma, B.; Li, H.; Hu, Z.; Gui, X.; Liu, L.; Liu, S. AutoMaAS: Self-Evolving Multi-Agent Architecture Search for Large Language Models. *CoRR* **2025**, *abs/2510.02669*, [2510.02669]. https://doi.org/10.48550/ARXIV.2510.02669.
162. Yao, T.; Li, Z.; Shen, Z. HieraMAS: Optimizing Intra-Node LLM Mixtures and Inter-Node Topology for Multi-Agent Systems. *CoRR* **2026**, *abs/2602.20229*, [2602.20229]. https://doi.org/10.48550/ARXIV.2602.20229.
163. Guo, J.; Xue, X.; Zhang, L.; Xu, W.; Chen, S.; Torr, P.; Ouyang, W.; Bai, L.; Yin, Z. SciOrch: Learning to Orchestrate Expert LLMs for Solving Frontier Multimodal Scientific Reasoning Tasks. 2026.

 doi:10.20944/preprints202608.0051.v1

164. Fang, W.; Yuan, L.; Lan, G.; Han, D.; Brinton, C.G. Iterative Critique-and-Routing Controller for Multi-Agent Systems with Heterogeneous LLMs. *CoRR* **2026**, *abs/2605.08686*, [2605.08686]. https://doi.org/10.48550/ARXIV.2605.08686.
165. Ye, R.; Liu, X.; Wu, Q.; Pang, X.; Yin, Z.; Bai, L.; Chen, S. X-MAS: Towards Building Multi-Agent Systems with Heterogeneous LLMs. *CoRR* **2025**, *abs/2505.16997*, [2505.16997]. https://doi.org/10.48550/ARXIV.2505.16997.
166. Feng, Y.; Du, J.; Hong, Y.; Wang, Q.; Yu, L. PASS: Probabilistic Agentic Supernet Sampling for Interpretable and Adaptive Chest X-Ray Reasoning. *CoRR* **2025**, *abs/2508.10501*, [2508.10501]. https://doi.org/10.48550/ARXIV.2508.10501.
167. Su, J.; Xia, Y.; Lan, Q.; Song, X.; Chen, C.; Yang, J.; He, L.; Shi, T. Difficulty-Aware Agent Orchestration in LLM-Powered Workflows. *CoRR* **2025**, *abs/2509.11079*, [2509.11079]. https://doi.org/10.48550/ARXIV.2509.11079.
168. Zhuge, M.; Wang, W.; Kirsch, L.; Faccio, F.; Khizbullin, D.; Schmidhuber, J. Language Agents as Optimizable Graphs. *CoRR* **2024**, *abs/2402.16823*, [2402.16823]. https://doi.org/10.48550/ARXIV.2402.16823.
169. Li, Z.; Xu, S.; Mei, K.; Hua, W.; Rama, B.; Raheja, O.; Wang, H.; Zhu, H.; Zhang, Y. AutoFlow: Automated Workflow Generation for Large Language Model Agents. *CoRR* **2024**, *abs/2407.12821*, [2407.12821]. https://doi.org/10.48550/ARXIV.2407.12821.
170. Zhang, J.; Xiang, J.; Yu, Z.; Teng, F.; Chen, X.; Chen, J.; Zhuge, M.; Cheng, X.; Hong, S.; Wang, J.; et al. AFlow: Automating Agentic Workflow Generation. *CoRR* **2024**, *abs/2410.10762*, [2410.10762]. https://doi.org/10.48550/ARXIV.2410.10762.
171. Wang, Y.; Yang, L.; Li, G.; Wang, M.; Aragam, B. ScoreFlow: Mastering LLM Agent Workflows via Score-based Preference Optimization. *CoRR* **2025**, *abs/2502.04306*, [2502.04306]. https://doi.org/10.48550/ARXIV.2502.04306.
172. Niu, B.; Song, Y.; Lian, K.; Shen, Y.; Yao, Y.; Zhang, K.; Liu, T. Flow: A Modular Approach to Automated Agentic Workflow Generation. *CoRR* **2025**, *abs/2501.07834*, [2501.07834]. https://doi.org/10.48550/ARXIV.2501.07834.
173. Zheng, C.; Chen, J.; Lyu, Y.; Ng, W.Z.T.; Zhang, H.; Ong, Y.; Tsang, I.W.; Yin, H. MermaidFlow: Redefining Agentic Workflow Generation via Safety-Constrained Evolutionary Programming. *CoRR* **2025**, *abs/2505.22967*, [2505.22967]. https://doi.org/10.48550/ARXIV.2505.22967.
174. Wang, Y.; Liu, S.; Fang, J.; Meng, Z. EvoAgentX: An Automated Framework for Evolving Agentic Workflows. *CoRR* **2025**, *abs/2507.03616*, [2507.03616]. https://doi.org/10.48550/ARXIV.2507.03616.
175. Zhang, G.; Chen, K.; Wan, G.; Chang, H.; Cheng, H.; Wang, K.; Hu, S.; Bai, L. EvoFlow: Evolving Diverse Agentic Workflows On The Fly. *CoRR* **2025**, *abs/2502.07373*, [2502.07373]. https://doi.org/10.48550/ARXIV.2502.07373.
176. Liu, S.; Fang, J.; Zhou, H.; Wang, Y.; Meng, Z. SEW: Self-Evolving Agentic Workflows for Automated Code Generation. *CoRR* **2025**, *abs/2505.18646*, [2505.18646]. https://doi.org/10.48550/ARXIV.2505.18646.
177. Wei, Y.; Huang, Z.; Li, H.; Xing, W.W.; Lin, T.; He, L. VFlow: Discovering Optimal Agentic Workflows for Verilog Generation. *CoRR* **2025**, *abs/2504.03723*, [2504.03723]. https://doi.org/10.48550/ARXIV.2504.03723.
178. Hou, Z.; Tang, J.; Wang, Y. HALO: Hierarchical Autonomous Logic-Oriented Orchestration for Multi-Agent LLM Systems. *CoRR* **2025**, *abs/2505.13516*, [2505.13516]. https://doi.org/10.48550/ARXIV.2505.13516.
179. Xu, B.; Ye, Y.; Shen, C.; Zhou, Y.; Chen, C.; Chen, M. HyEvo: Self-Evolving Hybrid Agentic Workflows for Efficient Reasoning. *CoRR* **2026**, *abs/2603.19639*, [2603.19639]. https://doi.org/10.48550/ARXIV.2603.19639.
180. Zhu, R.; Jiang, B.; Mei, L.; Yang, F.; Wang, L.; Gao, H.; Bai, F.; Zhao, P.; Lin, Q.; Rajmohan, S.; et al. AdaptFlow: Adaptive Workflow Optimization via Meta-Learning. *CoRR* **2025**, *abs/2508.08053*, [2508.08053]. https://doi.org/10.48550/ARXIV.2508.08053.
181. Wang, J.; Xu, S.; Liu, H.; Wang, J.; Luo, Y.; Di, S.; Zhang, M.; Chen, L. Learning to Compose for Cross-domain Agentic Workflow Generation. *CoRR* **2026**, *abs/2602.11114*, [2602.11114]. https://doi.org/10.48550/ARXIV.2602.11114.
182. Yuan, B.; Zhou, Y.; Xu, Z.; Ramnath, K.; Feng, A.; Srinivasan, B. BayesFlow: A Probability Inference Framework for Meta-Agent Assisted Workflow Generation. *CoRR* **2026**, *abs/2601.22305*, [2601.22305]. https://doi.org/10.48550/ARXIV.2601.22305.
183. Kong, M.; Qu, Z.; Zhou, Z.; Liang, P.; Li, X.; Shang, Z.; Hong, Z.; Huang, K.; Wang, Z.; Dai, Z. Workflow-R1: Group Sub-sequence Policy Optimization for Multi-turn Workflow Construction. *CoRR* **2026**, *abs/2602.01202*, [2602.01202]. https://doi.org/10.48550/ARXIV.2602.01202.
184. Ma, Z.; Zhao, Z.; Hua, C.; Berto, F.; Park, J. JudgeFlow: Agentic Workflow Optimization via Block Judge. *CoRR* **2026**, *abs/2601.07477*, [2601.07477]. https://doi.org/10.48550/ARXIV.2601.07477.

 doi:10.20944/preprints202608.0051.v1

185. Xu, S.; Zhang, J.; Di, S.; Luo, Y.; Yao, L.; Liu, H.; Zhu, J.; Liu, F.; Zhang, M. RobustFlow: Towards Robust Agentic Workflow Generation. *CoRR* **2025**, *abs/2509.21834*, [2509.21834]. https://doi.org/10.48550/ARXIV.2509.21834.
186. Shi, X.; Zheng, M.; Lou, Q. Learning Latency-Aware Orchestration for Parallel Multi-Agent Systems. *CoRR* **2026**, *abs/2601.10560*, [2601.10560]. https://doi.org/10.48550/ARXIV.2601.10560.
187. Li, J.; Hong, Z.; Shen, M.; Zhang, Y.; Gan, C. FlowCompile: An Optimizing Compiler for Structured LLM Workflows. *CoRR* **2026**, *abs/2605.13647*, [2605.13647]. https://doi.org/10.48550/ARXIV.2605.13647.
188. Li, A.; Yang, S.; Chen, F.; Xu, T.; Li, P.; Su, Z. GraphFlow: A Graph-Based Workflow Management for Efficient LLM-Agent Serving. *CoRR* **2026**, *abs/2605.22566*, [2605.22566]. https://doi.org/10.48550/ARXIV.2605.22566.
189. Li, J.; Zhang, E.; Zhou, D.; Chen, E.; Yan, Y. Learning to Hand Off: Provably Convergent Workflow Learning under Interface Constraints. *CoRR* **2026**, *abs/2605.19140*, [2605.19140]. https://doi.org/10.48550/ARXIV.2605.19140.
190. Liu, S.; Li, M.; Fu, D.; Wang, H.P.; Xia, Y.; Li, H.; Yan, H.; Li, P. Towards Direct Latent-Space Synthesis for Parallel Branches in LLM-Agent Workflows. 2026.
191. Hu, Y.; Cai, Y.; Du, Y.; Zhu, X.; Liu, X.; Yu, Z.; Hou, Y.; Tang, S.; Chen, S. Self-Evolving Multi-Agent Collaboration Networks for Software Development. *CoRR* **2024**, *abs/2410.16946*, [2410.16946]. https://doi.org/10.48550/ARXIV.2410.16946.
192. Zhang, G.; Yue, Y.; Sun, X.; Wan, G.; Yu, M.; Fang, J.; Wang, K.; Cheng, D. G-Designer: Architecting Multi-agent Communication Topologies via Graph Neural Networks. *CoRR* **2024**, *abs/2410.11782*, [2410.11782]. https://doi.org/10.48550/ARXIV.2410.11782.
193. Zhang, G.; Yue, Y.; Li, Z.; Yun, S.; Wan, G.; Wang, K.; Cheng, D.; Yu, J.X.; Chen, T. Cut the Crap: An Economical Communication Pipeline for LLM-based Multi-Agent Systems. *CoRR* **2024**, *abs/2410.02506*, [2410.02506]. https://doi.org/10.48550/ARXIV.2410.02506.
194. Li, B.; Zhao, Z.; Lee, D.; Wang, G. Adaptive Graph Pruning for Multi-Agent Communication. *CoRR* **2025**, *abs/2506.02951*, [2506.02951]. https://doi.org/10.48550/ARXIV.2506.02951.
195. Zhang, R.; Zhao, X.; Wang, R.; Chen, S.; Zhang, G.; Zhang, A.; Wang, K.; Wen, Q. SafeSieve: From Heuristics to Experience in Progressive Pruning for LLM-based Multi-Agent Communication. *CoRR* **2025**, *abs/2508.11733*, [2508.11733]. https://doi.org/10.48550/ARXIV.2508.11733.
196. Leong, H.Y.; Li, Y.; Wu, Y.; Ouyang, W.; Zhu, W.; Gao, J.; Han, W. AMAS: Adaptively Determining Communication Topology for LLM-based Multi-Agent System. *CoRR* **2025**, *abs/2510.01617*, [2510.01617]. https://doi.org/10.48550/ARXIV.2510.01617.
197. Jiang, E.H.; Wan, G.; Yin, S.; Li, M.; Wu, Y.; Liang, X.; Li, X.; Sun, Y.; Wang, W.; Chang, K.; et al. Dynamic Generation of Multi-LLM Agents Communication Topologies with Graph Diffusion Models. *CoRR* **2025**, *abs/2510.07799*, [2510.07799]. https://doi.org/10.48550/ARXIV.2510.07799.
198. Li, S.; Liu, Y.; Zheng, Y.; Li, M.; Nguyen, Q.V.H.; Pan, S. OFA-MAS: One-for-All Multi-Agent System Topology Design based on Mixture-of-Experts Graph Generative Models. *CoRR* **2026**, *abs/2601.12996*, [2601.12996]. https://doi.org/10.48550/ARXIV.2601.12996.
199. Wu, X.; Liu, X.; Lu, J.; Wang, S.; Qiu, X.; Shu, Y.; Hu, J.; Guo, C.; Yang, B. ST-EVO: Towards Generative Spatio-Temporal Evolution of Multi-Agent Communication Topologies. *CoRR* **2026**, *abs/2602.14681*, [2602.14681]. https://doi.org/10.48550/ARXIV.2602.14681.
200. Jiang, E.H.; Li, L.; Sun, R.; Liang, X.; Li, Y.; Wu, Y.; Luo, H.; Li, H.; Zhang, Z.; Kang, Z.; et al. Agent Q-Mix: Selecting the Right Action for LLM Multi-Agent Systems through Reinforcement Learning. *CoRR* **2026**, *abs/2604.00344*, [2604.00344]. https://doi.org/10.48550/ARXIV.2604.00344.
201. Zhang, Z.; Zhou, W.; Li, J.; Fei, H.; Wen, J.; Ji, W. RADAR: Redundancy-Aware Diffusion for Multi-Agent Communication Structure Generation. *CoRR* **2026**, *abs/2605.09907*, [2605.09907]. https://doi.org/10.48550/ARXIV.2605.09907.
202. Wu, X.; Lu, J.; Yan, S.; Qiu, X.; Hu, J.; Guo, C.; Yang, B. Differentiable Mixture-of-Agents Incentivizes Swarm Intelligence of Large Language Models. *CoRR* **2026**, *abs/2605.15706*, [2605.15706]. https://doi.org/10.48550/ARXIV.2605.15706.
203. Wang, S.; Lu, R.; Yang, Z.; Wang, Y.; Zhang, Y.; Xu, L.; Xu, Q.; Yin, G.; Chen, C.; Guan, X. AgentConductor: Topology Evolution for Multi-Agent Competition-Level Code Generation. *CoRR* **2026**, *abs/2602.17100*, [2602.17100]. https://doi.org/10.48550/ARXIV.2602.17100.
204. Xu, C.; Hu, Y.; Wang, R.; Lin, X.; Wang, W.; Liu, D.; Feng, F. TacoMAS: Test-Time Co-Evolution of Topology and Capability in LLM-based Multi-Agent Systems. *CoRR* **2026**, *abs/2605.09539*, [2605.09539]. https://doi.org/10.48550/ARXIV.2605.09539.

 doi:10.20944/preprints202608.0051.v1

205. Zhang, T.; Zhou, Z.; Wan, J.; Hu, T.; Wang, C.; He, X.; Hong, R. Learning Transferable Topology Priors for Multi-Agent LLM Collaboration Across Domains. *CoRR* **2026**, *abs/2605.17359*, [2605.17359]. https://doi.org/10.48550/ARXIV.2605.17359.
206. Wang, X.; Wang, J.; Zhang, F.; Hu, Y.; Zhang, D.; Ye, Y.; Ban, Y.; Han, J.; Wang, R. MasFACT: Continual Multi-Agent Topology Learning via Geometry-Aware Posterior Transfer. *CoRR* **2026**, *abs/2605.17361*, [2605.17361]. https://doi.org/10.48550/ARXIV.2605.17361.
207. Gou, W.; Liu, Z. Dynamic Trust-Aware Sparse Communication Topology for LLM-Based Multi-Agent Consensus. 2026.
208. Talluri, A.; Anne, P.; Pendiyala, B.C.; Chilukuri, R. Retrieval-Conditioned Topology Selection with Provable Budget Conservation for Multi-Agent Code Generation. *CoRR* **2026**, *abs/2605.05657*, [2605.05657]. https://doi.org/10.48550/ARXIV.2605.05657.
209. Tastan, N.; Iacob, A.; Sani, L.; Kurmanji, M.; Lane, N.D.; Horvath, S.; Nandakumar, K. Response-Conditioned Parallel-to-Sequential Orchestration for Multi-Agent Systems. *CoRR* **2026**, *abs/2605.15573*, [2605.15573]. https://doi.org/10.48550/ARXIV.2605.15573.
210. Wang, D.; Yin, D.; Desai, R.; Li, L.; Celikyilmaz, A.; Ni, A. Learning to Interrupt in Language-based Multi-agent Communication. *CoRR* **2026**, *abs/2604.06452*, [2604.06452]. https://doi.org/10.48550/ARXIV.2604.06452.
211. Yu, Y.; Liu, H.; Jin, H.; Yuan, X.; Kuang, P.; Wang, H. Learning to Communicate: Toward End-to-End Optimization of Multi-Agent Language Systems. *CoRR* **2026**, *abs/2604.21794*, [2604.21794]. https://doi.org/10.48550/ARXIV.2604.21794.
212. Xu, T.; Wen, H.; Li, M. Adapting the Interface, Not the Model: Runtime Harness Adaptation for Deterministic LLM Agents. *CoRR* **2026**, *abs/2605.22166*, [2605.22166]. https://doi.org/10.48550/ARXIV.2605.22166.
213. Chen, M.; Wang, J.; Liu, Z.; Wang, Y.; Wang, Q. From Failed Trajectories to Reliable LLM Agents: Diagnosing and Repairing Harness Flaws. 2026.
214. Wei, C.; Gao, M.; Han, Z.; Chen, K.; Zhuang, Y.; Guan, H.; Zhang, Y.; Cheng, Y.; He, J.; Chen, H.; et al. The World Leaks the Future: Harness Evolution for Future Prediction Agents. *CoRR* **2026**, *abs/2604.15719*, [2604.15719]. https://doi.org/10.48550/ARXIV.2604.15719.
215. Zhang, H.; Zhang, S.; Li, K.; Zhang, C.; Chen, Y.; Zhang, Y.; Bai, L.; Hu, S. Self-Harness: Harnesses That Improve Themselves. 2026.
216. Kakade, A.; Srivastava, V.; Karande, S.S. Polaris: A Gödel Agent Framework for Small Language Models through Experience-Abstracted Policy Repair. *CoRR* **2026**, *abs/2603.23129*, [2603.23129]. https://doi.org/10.48550/ARXIV.2603.23129.
217. Lee, Y.; Nair, R.; Zhang, Q.; Lee, K.; Khattab, O.; Finn, C. Meta-Harness: End-to-End Optimization of Model Harnesses. *CoRR* **2026**, *abs/2603.28052*, [2603.28052]. https://doi.org/10.48550/ARXIV.2603.28052.
218. Liu, H.; Shou, C.; Liu, X.; Wen, H.; Chen, Y.; Fang, R.J.; Feng, Y. Synthesizing Multi-Agent Harnesses for Vulnerability Discovery. *CoRR* **2026**, *abs/2604.20801*, [2604.20801]. https://doi.org/10.48550/ARXIV.2604.20801.
219. Zhang, D. AgentDevel: Reframing Self-Evolving LLM Agents as Release Engineering. *CoRR* **2026**, *abs/2601.04620*, [2601.04620]. https://doi.org/10.48550/ARXIV.2601.04620.
220. Nanda, R.; Maddila, C.; Jha, S.; Khan, E.M.; Paltenghi, M.; Chandra, S. Wink: Recovering from Misbehaviors in Coding Agents. *CoRR* **2026**, *abs/2602.17037*, [2602.17037]. https://doi.org/10.48550/ARXIV.2602.17037.
221. Mulian, H.; Zeltyn, S.; Levy, I.; Galanti, L.; Yaeli, A.; Shlomov, S. AgentFixer: From Failure Detection to Fix Recommendations in LLM Agentic Systems. *CoRR* **2026**, *abs/2603.29848*, [2603.29848]. https://doi.org/10.48550/ARXIV.2603.29848.
222. Bonagiri, A.; Borkar, D.; Anderias, G.J.; Rafatirad, S.; Homayoun, H. CausalFlow: Causal Attribution and Counterfactual Repair for LLM Agent Failures. *CoRR* **2026**, *abs/2605.25338*, [2605.25338]. https://doi.org/10.48550/ARXIV.2605.25338.
223. Zhang, G.; Wang, J.; Chen, J.; Zhou, W.; Wang, K.; Yan, S. AgenTracer: Who Is Inducing Failure in the LLM Agentic Systems? *CoRR* **2025**, *abs/2509.03312*, [2509.03312]. https://doi.org/10.48550/ARXIV.2509.03312.
224. Zhang, S.; Yin, M.; Zhang, J.; Liu, J.; Han, Z.; Zhang, J.; Li, B.; Wang, C.; Wang, H.; Chen, Y.; et al. Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems. *CoRR* **2025**, *abs/2505.00212*, [2505.00212]. https://doi.org/10.48550/ARXIV.2505.00212.
225. Zelikman, E.; Lorch, E.; Mackey, L.; Kalai, A.T. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation. *CoRR* **2023**, *abs/2310.02304*, [2310.02304]. https://doi.org/10.48550/ARXIV.2310.02304.

 doi:10.20944/preprints202608.0051.v1

226. Yin, X.; Wang, X.; Pan, L.; Lin, L.; Wan, X.; Wang, W.Y. Gödel Agent: A Self-Referential Agent Framework for Recursive Self-Improvement. In Proceedings of the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, 2025.
227. Robeyns, M.; Szummer, M.; Aitchison, L. A Self-Improving Coding Agent. *CoRR* **2025**, *abs/2504.15228*, [2504.15228]. https://doi.org/10.48550/ARXIV.2504.15228.
228. Cai, Q.; Zhang, Y.; Jia, X.; Zheng, H.; Xue, W.; Song, J.; Tian, X.; Guo, Y. MOSS: Self-Evolution through Source-Level Rewriting in Autonomous Agent Systems. *CoRR* **2026**, *abs/2605.22794*, [2605.22794]. https://doi.org/10.48550/ARXIV.2605.22794.
229. Robol, M.; Giorgini, P. Self-Evolving Software Agents. *CoRR* **2026**, *abs/2604.27264*, [2604.27264]. https://doi.org/10.48550/ARXIV.2604.27264.
230. Che, L.; Yang, Y.; Lin, P.; Wang, C.; Wang, X.; Su, J. DemoEvolve: Overcoming Sparse Feedback in Agentic Harness Evolution with Demonstrations. *CoRR* **2026**, *abs/2605.24539*, [2605.24539]. https://doi.org/10.48550/ARXIV.2605.24539.
231. Ho, M.; Liu, B.; Chen, J.; Wang, A.X.; Qin, L. SIGA: Self-Evolving Coding-Agent Adapters for Scientific Simulation. 2026.
232. Wang, R.; Huang, J.; Wang, P.; Liu, X.; Kong, L.; Zhang, T. Lean4Agent: Formal Modeling and Verification for Agent Workflow and Trajectory. 2026.
233. Zhang, S.; Yuan, C.; Guo, R.; Yu, X.; Xu, R.; Chen, Z.; Li, Z.; Yang, Z.; Guan, S.; Tang, Z.; et al. EvoFSM: Controllable Self-Evolution for Deep Research with Finite State Machines. *CoRR* **2026**, *abs/2601.09465*, [2601.09465]. https://doi.org/10.48550/ARXIV.2601.09465.
234. Zhang, J.; Hu, S.; Lu, C.; Lange, R.T.; Clune, J. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. In Proceedings of the The Fourteenth International Conference on Learning Representations, 2026.
235. Wang, W.; Piekos, P.; Li, N.; Laakom, F.; Chen, Y.; Ostaszewski, M.; Zhuge, M.; Schmidhuber, J. Huxley-Gödel Machine: Human-Level Coding Agent Development by an Approximation of the Optimal Self-Improving Machine. *CoRR* **2025**, *abs/2510.21614*, [2510.21614]. https://doi.org/10.48550/ARXIV.2510.21614.
236. Weng, Z.; Antoniades, A.; Nathani, D.; Zhang, Z.; Pu, X.; Wang, X.E. Group-Evolving Agents: Open-Ended Self-Improvement via Experience Sharing. *CoRR* **2026**, *abs/2602.04837*, [2602.04837]. https://doi.org/10.48550/ARXIV.2602.04837.
237. Zhang, J.; Gu, Y.; Ruan, J.; Song, M.; Peng, Y.; Han, Z.; Xiang, J.; Wang, Z.; Yang, C.; Ouyang, Y.; et al. Harnessing Agentic Evolution. *CoRR* **2026**, *abs/2605.13821*, [2605.13821]. https://doi.org/10.48550/ARXIV.2605.13821.
238. Qu, Y.; Lu, M. Bilevel Autoresearch: Meta-Autoresearching Itself. *CoRR* **2026**, *abs/2603.23420*, [2603.23420]. https://doi.org/10.48550/ARXIV.2603.23420.
239. Liu, Z.; Shi, Z.; Sang, Y.; He, B.; Lin, M.; Wei, T.; Wang, D.; Dumoulin, B.; Jin, W.; Lu, H. Adaptive Auto-Harness: Sustained Self-Improvement for Agentic System Deployment on Open-Ended Task Streams. 2026.
240. Liu, S.; Agarwal, S.; Maheswaran, M.; Cemri, M.; Li, Z.; Mang, Q.; Naren, A.; Boneh, E.; Cheng, A.; Pan, M.Z.; et al. EvoX: Meta-Evolution for Automated Discovery. *CoRR* **2026**, *abs/2602.23413*, [2602.23413]. https://doi.org/10.48550/ARXIV.2602.23413.
241. Wu, X.; Yin, S.; Kang, Y.; Zhang, X.; Xu, Q.; Chen, Z.; Zhang, W. SGM: A Statistical Godel Machine for Risk-Controlled Recursive Self-Modification. *CoRR* **2025**, *abs/2510.10232*, [2510.10232]. https://doi.org/10.48550/ARXIV.2510.10232.
242. Garralda-Barrio, M. Governed Evolution of Agent Runtimes through Executable Operational Cognition. *CoRR* **2026**, *abs/2605.27328*, [2605.27328]. https://doi.org/10.48550/ARXIV.2605.27328.
243. Hakim, S.B.; Guo, K.; Tan, W.; Velasquez, A.; Xu, S.; Song, H.H. ANNEAL: Adapting LLM Agents via Governed Symbolic Patch Learning. *CoRR* **2026**, *abs/2605.16309*, [2605.16309]. https://doi.org/10.48550/ARXIV.2605.16309.
244. Sahoo, S.; Chadha, A.; Jain, V.; Chaudhary, D. SAHOO: Safeguarded Alignment for High-Order Optimization Objectives in Recursive Self-Improvement. *CoRR* **2026**, *abs/2603.06333*, [2603.06333]. https://doi.org/10.48550/ARXIV.2603.06333.
245. Shi, D.; He, J.; Chen, J.; Wang, B.; Nakashima, Y. Towards Healthy Evolution: Exploring the Role and Mechanisms of Human-Agent Interaction in Self-Evolving Systems. 2026.
246. Zala, A.; Cho, J.; Lin, H.; Yoon, J.; Bansal, M. EnvGen: Generating and Adapting Environments via LLMs for Training Embodied Agents, 2024, [arXiv:cs.CL/2403.12014].

 doi:10.20944/preprints202608.0051.v1

247. Yeo, T.; Weerakoon, D.; Weerakoon, D.; Misra, A. Towards Adaptive Environment Generation for Training Embodied Agents, 2026, [arXiv:cs.RO/2602.06366].
248. Parker-Holder, J.; Jiang, M.; Dennis, M.; Samvelyan, M.; Foerster, J.; Grefenstette, E.; Rocktäschel, T. Evolving Curricula with Regret-Based Environment Design, 2022, [arXiv:cs.LG/2203.01302].
249. Garcin, S.; Doran, J.; Guo, S.; Lucas, C.G.; Albrecht, S.V. DRED: Zero-Shot Transfer in Reinforcement Learning via Data-Regularised Environment Design, 2024, [arXiv:cs.LG/2402.03479].
250. Kang, H.; Ye, X.; Liu, Y.; Mantri, S.H.; Mao, L.; Fleming, J.; Regmi, D.; Qin, L. SimWorld Studio: Automatic Environment Generation with Evolving Coding Agent for Embodied Agent Learning, 2026, [arXiv:cs.AI/2605.09423].
251. Chen, Z.; Zhao, Z.; Zhang, K.; Liu, B.; Qi, Q.; Wu, Y.; Kalluri, T.; Cao, S.; Xiong, Y.; Tong, H.; et al. Scaling Agent Learning via Experience Synthesis, 2025, [arXiv:cs.AI/2511.03773].
252. Qi, Z.; Liu, X.; Iong, I.L.; Lai, H.; Sun, X.; Zhao, W.; Yang, Y.; Yang, X.; Sun, J.; Yao, S.; et al. WebRL: Training LLM Web Agents via Self-Evolving Online Curriculum Reinforcement Learning, 2024, [arXiv:cs.CL/2411.02337].
253. Yang, S.; Ma, Z.; Huang, T.; Hu, Y.; Wang, Y.; Chu, X. CoEvolve: Training LLM Agents via Agent-Data Mutual Evolution. In Proceedings of the Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), San Diego, California, United States, 2026; pp. 23015–23036. https://doi.org/10.18653/v1/2026.acl-long.1055.
254. Fang, W.; Liu, S.; Zhou, Y.; Zhang, K.; Zheng, T.; Chen, K.; Song, M.; Tao, D. SeRL: Self-Play Reinforcement Learning for Large Language Models with Limited Data, 2026, [arXiv:cs.CL/2505.20347].
255. Huang, J.; Li, Z.; Hu, Y.; Zhang, Z.; Coates, M.; Quan, X.; Zhang, Y. Self-CriTeach: LLM Self-Teaching and Self-Critiquing for Improving Robotic Planning via Automated Domain Generation, 2026, [arXiv:cs.RO/2509.21543].
256. Amjith, S.; Wang, M.X.; Lynch, J.; Gundlach, H.; Thompson, N. SAGE: Self-play Adversarial Games Enhance Large Language Model Reasoning Capabilities.
257. Chen, Z.; Deng, Y.; Yuan, H.; Ji, K.; Gu, Q. Self-Play Fine-Tuning Converts Weak Language Models to Strong Language Models, 2024, [arXiv:cs.LG/2401.01335].
258. Luo, H.; Sun, Q.; Xu, C.; Zhao, P.; Lin, Q.; Lou, J.; Chen, S.; Tang, Y.; Chen, W. Arena Learning: Build Data Flywheel for LLMs Post-training via Simulated Chatbot Arena, 2024, [arXiv:cs.CL/2407.10627].
259. Zeng, Y.; Cui, X.; Jin, X.; Mi, Q.; Liu, G.; Sun, Z.; Yang, M.; Li, D.; Ma, W.; Yang, N.; et al. Evolving LLMs' Self-Refinement Capability via Synergistic Training-Inference Optimization, 2025, [arXiv:cs.CL/2502.05605].
260. Qu, Y.; Zhang, T.; Garg, N.; Kumar, A. Recursive Introspection: Teaching Language Model Agents How to Self-Improve, 2024, [arXiv:cs.LG/2407.18219].
261. Yang, H.; Le, K.; Hua, T.; Gao, S.; Xu, B.; Tang, Z.; Xu, J.; Chawla, N.V.; Jin, H.; Srinivasan, V. Dynamic Noise Preference Optimization: Self-Improvement of Large Language Models with Self-Synthetic Data, 2026, [arXiv:cs.CL/2502.05400].
262. Xiao, W.; Lin, H.; Peng, A.; Xue, H.; He, T.; Xie, Y.; Hu, F.; Wu, J.; Luo, Z.; Fan, L.J.; et al. Self-Improving Vision-Language-Action Models with Data Generation via Residual RL, 2025, [arXiv:cs.CV/2511.00091].
263. Lin, I.W.; Hu, Y.; Li, S.S.; Geng, S.; Koh, P.W.; Zettlemoyer, L.; Althoff, T.; Ghazvininejad, M. Self-Improving VLM Judges Without Human Annotations, 2025, [arXiv:cs.CV/2512.05145].
264. Tang, C.; Huang, H.Y.; Liu, W.; Zheng, J.; Yang, S.; Wu, Y. Democratizing Tool Learning with Environments Fully Simulated by a Free 8B Language Model. *arXiv preprint arXiv:2604.17739* **2026**.
265. Cheng, Y.; Wang, Z.; Ma, W.; Zhu, W.; Deng, Y.; Zhao, J. EvoCurr: Self-evolving Curriculum with Behavior Code Generation for Complex Decision-making, 2025, [arXiv:cs.AI/2508.09586].
266. Zhang, Q.; Ruan, S.; Upasani, S.; Hong, F.; Ji, C.; Hu, C.; Li, B.; Li, H.; Olukotun, K. Learning What to Learn: Curriculum Curation for Test-Time Agent Learning. In Proceedings of the ICLR 2026 Workshop on AI with Recursive Self-Improvement (RSI), 2026.
267. Bukkapatnam, K.; Lala, A.; Patel, L. Adaptive Meta-Curriculum for Test-Time Self-Improvement. In Proceedings of the ICLR 2026 Workshop on AI with Recursive Self-Improvement (RSI), 2026.
268. Gu, Z.; Light, J.; Astudillo, R.; Ye, Z.; He, L.; Zou, H.P.; Cheng, W.; Paternain, S.; Yu, P.S.; Yue, Y. Actor-Curator: Co-adaptive Curriculum Learning via Policy-Improvement Bandits for RL Post-Training, 2026, [arXiv:cs.LG/2602.20532].
269. Liu, J.; Xiong, K.; Xia, P.; Zhou, Y.; Ji, H.; Feng, L.; Han, S.; Ding, M.; Yao, H. Agent0-VL: Exploring Self-Evolving Agent for Tool-Integrated Vision-Language Reasoning, 2025, [arXiv:cs.CV/2511.19900].

 doi:10.20944/preprints202608.0051.v1

270. Huang, Y.; Yu, X.; Wei, Z. ACE: Self-Evolving LLM Coding Framework via Adversarial Unit Test Generation and Preference Optimization, 2026, [arXiv:cs.SE/2605.16299].
271. Dong, G.; Lu, J.; Huang, J.; Zhong, W.; Liu, L.; Huang, S.; Li, Z.; Zhao, Y.; Song, X.; Li, X.; et al. Agent-World: Scaling Real-World Environment Synthesis for Evolving General Agent Intelligence, 2026, [arXiv:cs.AI/2604.18292].
272. Xia, P.; Zeng, K.; Liu, J.; Qin, C.; Wu, F.; Zhou, Y.; Xiong, C.; Yao, H. Agent0: Unleashing Self-Evolving Agents from Zero Data via Tool-Integrated Reasoning, 2025, [arXiv:cs.LG/2511.16043].
273. Huang, C.; Yu, W.; Wang, X.; Zhang, H.; Li, Z.; Li, R.; Huang, J.; Mi, H.; Yu, D. R-Zero: Self-Evolving Reasoning LLM from Zero Data, 2025, [arXiv:cs.LG/2508.05004].
274. Sundaram, S.; Quan, J.; Kwiatkowski, A.; Ahuja, K.; Ollivier, Y.; Kempe, J. Teaching Models to Teach Themselves: Reasoning at the Edge of Learnability, 2026, [arXiv:cs.LG/2601.18778].
275. Jana, S.; Sancaktar, C.; Daniš, T.; Martius, G.; Orvieto, A.; Kolev, P. GASP: Guided Asymmetric Self-Play For Coding LLMs, 2026, [arXiv:cs.LG/2603.15957].
276. Shao, R.; Asai, A.; Shen, S.Z.; Ivison, H.; Kishore, V.; Zhuo, J.; Zhao, X.; Park, M.; Finlayson, S.G.; Sontag, D.; et al. DR Tulu: Reinforcement Learning with Evolving Rubrics for Deep Research, 2025, [arXiv:cs.CL/2511.19399].
277. Li, G.; Mishra, B.D.; Wang, Z.; Yan, J.; Chen, Y.; Li, C.L.; Le, L.T.; Han, R.; Lee, G.; Tong, H.; et al. RubricEM: Meta-RL with Rubric-guided Policy Decomposition beyond Verifiable Rewards, 2026, [arXiv:cs.CL/2605.10899].
278. Liu, Z.; Zhang, L.; Wang, X.; Xu, Z.; Zhan, S.; Shan, X.; Huang, W.; Dai, T.; Xia, S.T.; Huo, C.; et al. ARBOR: Online Process Rewards via a Reusable Rubric Buffer for Search Agents, 2026, [arXiv:cs.CL/2606.03239].
279. Zheng, C.; Mo, X.; Ma, X.; Lin, Q.; Zhao, Y.; Zhu, J.; Lou, X.; Wang, J.; Wang, Z.; Liu, W.; et al. Adaptive Milestone Reward for GUI Agents. *arXiv preprint arXiv:2602.11524* **2026**, [arXiv:cs.LG/2602.11524].
280. Kim, Z.M.; Park, C.; Raheja, V.; Kim, S.; Kang, D. Toward Evaluative Thinking: Meta Policy Optimization with Evolving Reward Models. *arXiv preprint arXiv:2504.20157* **2025**, [arXiv:cs.CL/2504.20157].
281. Ding, H.; Huang, B.; Fang, Y.; Liao, W.; Li, Z.; Zhang, J.; Wu, Z.; Zhao, J.; Wang, Y. EvoRubrics: Dynamic Rubrics as Rewards via Adversarial Co-Evolution for LLM Reinforcement Learning, 2026, [arXiv:cs.CL/2606.23038].
282. Li, S.S.; Xin, R.; Xiao, T.; Wang, Y.; Shao, R.; Hao, Z.; Sclar, M.; Oh, S.; Brahman, F.; Koh, P.W.; et al. EvoLM: Self-Evolving Language Models through Co-Evolved Discriminative Rubrics, 2026, [arXiv:cs.CL/2605.03871].
283. Sheng, L.; Ma, W.; Hong, R.; Wang, X.; Zhang, A.; Chua, T.S. Reinforcing Chain-of-Thought Reasoning with Self-Evolving Rubrics, 2026, [arXiv:cs.AI/2602.10885].
284. Guan, X.; Hu, X.; Huang, S.; Wang, Z.; Zhang, B.; Li, Z.; Xie, P.; Liu, B.; Cao, J. EvoRubric: Self-Evolving Rubric-Driven RL for Open-Ended Generation, 2026, [arXiv:cs.CL/2605.29847].
285. Tian, Z.; Zhang, J.; Li, R.; Bo, X.; Li, Y.; Chen, X. ARCO: Adaptive Rubric with Co-Evolution for Multi-Step LLM-Based Agents, 2026, [arXiv:cs.CL/2606.21262].
286. Feng, A.Z.; Wang, C.; Wen, B.; Wang, Y.; Luo, Y.; Wang, H.; Huang, M. RLAR: An Agentic Reward System for Multi-task Reinforcement Learning on Large Language Models, 2026, [arXiv:cs.CL/2603.00724].
287. Wang, X.; Wu, T.; Tang, M.; Li, J.; Liu, Q.; Zheng, Z. The Flip Side of RLHF: On-Policy Feedback for Reward Model Self-Supervised Improvement, 2026, [arXiv:cs.CL/2605.30888].
288. Shi, T.; Huang, C.; Wan, F.; Zhong, L.; Yang, Z.; Shen, W.; Quan, X.; Yan, M. Mutual-Taught for Co-adapting Policy and Reward Models, 2025, [arXiv:cs.CL/2506.06292].
289. Li, Z.; Cao, Z.; Huang, W.; Zhang, Y.; Qi, K.; Wang, R.; Zheng, Z.; Zhao, J.; Zhu, H.; Wu, H.; et al. MagicGUI-RMS: A Multi-Agent Reward Model System for Self-Evolving GUI Agents via Automated Feedback Reflux, 2026, [arXiv:cs.AI/2601.13060].
290. Lin, Y.; Wang, L.; Lin, K.; Lin, Z.; Gong, K.; Li, W.; Lin, B.; Li, Z.; Zhang, S.; Peng, Y.; et al. JarvisEvo: Towards a Self-Evolving Photo Editing Agent with Synergistic Editor-Evaluator Optimization, 2025, [arXiv:cs.CV/2511.23002].
291. Li, Z.; Jiang, L.; Hu, Y.; Zeng, X.; Li, Y.; Zhang, X.; Chen, G.; Pan, Z.; Li, X.; Liu, Y. No More Stale Feedback: Co-Evolving Critics for Open-World Agent Learning, 2026, [arXiv:cs.AI/2601.06794].
292. Lin, J.; Yu, X.; Xin, Y.; Guo, Y.; Jiang, Z.; Yue, Z.; Wang, W.; Zou, H.; Qin, C.; Xiong, H. ICRL: Learning to Internalize Self-Critique with Reinforcement Learning, 2026, [arXiv:cs.AI/2605.15224].
293. Wu, M.; Zhang, G.; Min, S.; Levine, S.; Kumar, A. RLAC: Reinforcement Learning with Adversarial Critic for Free-Form Generation Tasks. *arXiv preprint arXiv:2511.01758* **2025**, [arXiv:cs.LG/2511.01758].

 doi:10.20944/preprints202608.0051.v1

294. Sun, W.; Cheng, X.; Fan, J.; Yu, X.; Xu, Y.; He, S.; Zhao, J.; Liu, K. Towards Agentic Self-Learning LLMs in Search Environment, 2025, [arXiv:cs.AI/2510.14253].
295. Sui, Y.; Hooi, B. Conversation for Non-verifiable Learning: Self-Evolving LLMs through Meta-Evaluation, 2026, [arXiv:cs.CL/2601.21464].
296. Pan, T.; Yan, Y.; Wang, Z.; Zhang, R.; Hou, G.; Zhang, W.; Lu, W.; Xiao, J.; Shen, Y. CoVerRL: Breaking the Consensus Trap in Label-Free Reasoning via Generator-Verifier Co-Evolution. *arXiv preprint arXiv:2603.17775* **2026**, [arXiv:cs.CL/2603.17775].
297. Zhu, H.; Cai, C.; Song, Y.; Chen, X.; Han, S.; Guo, Y. Self-Evolving Deep Research via Joint Generation and Evaluation. *arXiv preprint arXiv:2606.04507* **2026**, [arXiv:cs.CL/2606.04507].
298. Lu, S.; Wang, H.; Chen, Z.; Tang, Y. URPO: A Unified Reward & Policy Optimization Framework for Large Language Models. *arXiv preprint arXiv:2507.17515* **2025**, [arXiv:cs.CV/2507.17515].
299. Zha, K.; Gao, Z.; Shen, M.; Hong, Z.W.; Boning, D.S.; Katabi, D. RL Tango: Reinforcing Generator and Verifier Together for Language Reasoning. *arXiv preprint arXiv:2505.15034* **2025**, [arXiv:cs.LG/2505.15034].
300. Wang, Y.; Xie, T.; Shen, K.; Wang, M.; Yang, L. RLAnything: Forge Environment, Policy, and Reward Model in Completely Dynamic RL System, 2026, [arXiv:cs.LG/2602.02488].
301. Zhang, Y.; Fang, M.; Chen, Z.; Pechenizkiy, M. Self-evolving LLM agents with in-distribution Optimization, 2026, [arXiv:cs.LG/2606.07367].
302. Guan, X.; Zhang, L.L.; Liu, Y.; Shang, N.; Sun, Y.; Zhu, Y.; Yang, F.; Yang, M. rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking. *arXiv preprint arXiv:2501.04519* **2025**, [arXiv:cs.CL/2501.04519].
303. Zhou, C.; Xu, T.; Lin, J.; Ge, D. StepORLM: A Self-Evolving Framework With Generative Process Supervision For Operations Research Language Models. *arXiv preprint arXiv:2509.22558* **2025**, [arXiv:cs.AI/2509.22558].
304. Zhang, J.; Ma, G.; Liu, S.; Hu, Z.; Jing, Y.; Lin, T.E.; Li, Y.; Tao, D. STRIDE: Learnable Stepwise Language Feedback for LLM Reasoning. *arXiv preprint arXiv:2605.18851* **2026**, [arXiv:cs.LG/2605.18851].
305. Xiao, H.; Wang, G.; Chai, Y.; Lu, Z.; Lin, W.; He, H.; Fan, L.; Bian, L.; Hu, R.; Liu, L.; et al. UI-Genie: A Self-Improving Approach for Iteratively Boosting MLLM-based Mobile GUI Agents, 2025, [arXiv:cs.CL/2505.21496].
306. Zhi, X.; zhou, P.; Lu, C.; Lv, H.; Liang, Y.; Zhang, R.; Gao, Y.; WU, Y.; Hu, Y.; Gu, H.; et al. SPARD: Self-Paced Curriculum for RL Alignment via Integrating Reward Dynamics and Data Utility. *arXiv preprint arXiv:2604.07837* **2026**, [arXiv:cs.AI/2604.07837].
307. Fan, L.; Chen, M.; Zhu, T.; Liu, K.; Xia, X.; Li, S.; Liu, Z. ZeroCoder: Can LLMs Improve Code Generation Without Ground-Truth Supervision? *arXiv preprint arXiv:2604.07864* **2026**, [arXiv:cs.SE/2604.07864].
308. Wang, H.; Wang, G.; Xiao, H.; Zhou, Y.; Pan, Y.; Wang, J.; Xu, K.; Wen, Y.; Ruan, X.; Chen, X.; et al. Skill-SD: Skill-Conditioned Self-Distillation for Multi-turn LLM Agents, 2026, [arXiv:cs.LG/2604.10674].
309. Tu, S.; Xu, C.; Zhang, Q.; Ma, Y.; Zhang, Y.; Li, L.; Li, D.; Lan, X.; Zhao, D. UCOB: Learning to Utilize and Evolve Agentic Skills via Credit-Aware On-Policy Bidirectional Self-Distillation, 2026, [arXiv:cs.AI/2606.29502].
310. Jeon, U.; Kwon, J.; Sullivan, M.A.; Lee, C.E.; Lin, G. ATLAS: Adaptive Self-Evolutionary Research Agent with Task-Distributed Multi-LLM Supporters, 2026, [arXiv:cs.AI/2602.02709].
311. Rao, A.; Advani, N.K. AI Training Manager: Bounded Closed-Loop Control of Adaptive Training Recipes, 2026, [arXiv:cs.AI/2606.29871].
312. Dong, H.; Yang, D.; Liang, X.; Feng, C.; Ran, J. AdaLRS: Loss-Guided Adaptive Learning Rate Search for Efficient Foundation Model Pretraining. In Proceedings of the Advances in Neural Information Processing Systems, 2025, [arXiv:cs.LG/2506.13274].
313. Lin, Z.; Xue, C.; Liang, D.; Han, X.; Liu, P.; Wu, X.; Jiang, L.; Lu, Y.; Shi, H.; Liang, S.; et al. Parameter Importance is Not Static: Evolving Parameter Isolation for Supervised Fine-Tuning. *arXiv preprint arXiv:2604.14010* **2026**, [arXiv:cs.LG/2604.14010].
314. Sakip, A.; Fuadi, E.H.; Sayedelahl, O.; Li, Z.; She, J.; Aji, A.F.; Liu, S.; Xing, E.; Ho, Q. COPUS: Co-adaptive Parallelism and Batch Size Selection in Large Language Model Training, 2026, [arXiv:cs.DC/2604.26687].
315. Mai, L.; Li, G.; Wagenländer, M.; Fertakis, K.; Brabete, A.O.; Pietzuch, P. KungFu: Making Training in Distributed Machine Learning Adaptive. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020, pp. 937–954.
316. Qiao, A.; Choe, S.K.; Subramanya, S.J.; Neiswanger, W.; Ho, Q.; Zhang, H.; Ganger, G.R.; Xing, E.P. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In Proceedings of the 15th USENIX

 doi:10.20944/preprints202608.0051.v1

Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 2021, pp. 1–18.
317. Subramanya, S.J.; Arfeen, D.; Lin, S.; Qiao, A.; Jia, Z.; Ganger, G.R. Sia: Heterogeneity-aware, Goodput-optimized ML-cluster Scheduling. In Proceedings of the Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 642–657. https://doi.org/10.1145/3600006.3613175.
318. Bian, Z.; Li, S.; Wang, W.; You, Y. Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters. In Proceedings of the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15, [2108.03645]. https://doi.org/10.1145/3458817.3480859.
319. Dai, Y.; He, K.; Wang, A. DYNAMIX: RL-based Adaptive Batch Size Optimization in Distributed Machine Learning Systems, 2025, [arXiv:cs.LG/2510.08522].
320. Zhang, L.; Jia, T.; Zhai, Y.; Fang, L.; Zheng, K.; Liu, H.; Huang, X.; Yu, P.S.; Li, Y. Towards Robust LLM Post-Training: Automatic Failure Management for Reinforcement Fine-Tuning. *arXiv preprint arXiv:2605.04431* **2026**, [arXiv:cs.LG/2605.04431].
321. Tan, Z.; Abdullahi, M.; Shi, T.; Yuan, H.; Xu, Z.; Yu, C.; Li, B.; Zhao, B. EARL: Efficient Agentic Reinforcement Learning Systems for Large Language Models, 2025, [arXiv:cs.DC/2510.05943].
322. Peng, Y.; Bao, Y.; Chen, Y.; Wu, C.; Guo, C. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In Proceedings of the Proceedings of the Thirteenth EuroSys Conference, 2018. https://doi.org/10.1145/3190508.3190517.
323. Zhang, X.; Zhao, H.; Xiao, W.; Jia, X.; Xu, F.; Li, Y.; Lin, W.; Liu, F. Rubick: Exploiting Job Reconfigurability for Deep Learning Cluster Scheduling, 2024, [arXiv:cs.DC/2408.08586].
324. Fang, H.; Zhu, W.; Han, B.; Zhang, A.; Pan, Z.; Yang, S.; Zhang, S.; Gai, J.; Tang, P.; Hu, C.; et al. LLMZero: Discovering Adaptive Training Strategies for RL Post-Training via LLM Agents. *arXiv preprint arXiv:2606.18388* **2026**, [arXiv:cs.LG/2606.18388].
325. Chen, G.; Shi, Y.; Li, Y.; Li, B.; Xu, X.; Wei, H.; Ni, S.; Yang, M.; Ye, J. EvoTrainer: Co-Evolving LLM Policies and Training Harnesses for Autonomous Agentic Reinforcement Learning, 2026, [arXiv:cs.AI/2606.03108].
326. Yu, Z.; Yin, P.; Gao, S.; He, S.; Cai, K.; Zhang, X.P. AutoTrainess: Teaching Language Models to Improve Language Models Autonomously. *arXiv preprint arXiv:2606.31551* **2026**, [arXiv:cs.CL/2606.31551].
327. Li, Q.; Zhang, Y.; Yang, X.; Yang, X.; Wang, Z.; Liu, W.; Bian, J. FT-Dojo: Towards Autonomous LLM Fine-Tuning with Language Agents. *arXiv preprint arXiv:2603.01712* **2026**, [arXiv:cs.AI/2603.01712].
328. Ning, J.; Li, X.; Zeng, J.; Kang, H.; Xiong, C. Auto Research with Specialist Agents Develops Effective and Non-Trivial Training Recipes, 2026, [arXiv:cs.MA/2605.05724].
329. Mroueh, Y.; Fonseca, C.; Belgodere, B.; Cox, D. CliffSearch: Structured Agentic Co-Evolution over Theory and Code for Scientific Algorithm Discovery, 2026, [arXiv:cs.LG/2604.01210].
330. Chen, S.; Tang, Z.; Zhang, W.; Yang, F.; Wang, Y.; Li, T.; Liu, Y. OptiCo: Adaptive Distributed Training Optimization via Collaborative Agent Reasoning. In Proceedings of the Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics, 2026.
331. Gao, S.; Fang, A.; Zitnik, M. AutoScientists: Self-Organizing Agent Teams for Long-Running Scientific Experimentation. *arXiv preprint arXiv:2605.28655* **2026**, [arXiv:cs.AI/2605.28655].
332. Ma, Z.; Wang, G.; Xie, X.; Chen, Y.; Du, H.; Li, B.; Sun, Y.; Liu, W.; Chen, K.; Li, Y. TREX: Automating LLM Fine-tuning via Agent-Driven Tree-based Exploration, 2026, [arXiv:cs.AI/2604.14116].
333. Xia, S.; Zhang, Y.; Chen, A.; Wu, S.; Yuan, S.; Xiao, Y. From AI Assistant to AI Scientist: Autonomous Discovery of LLM-RL Algorithms with LLM Agents, 2026, [arXiv:cs.LG/2603.23951].
334. Ma, Y.J.; Liang, W.; Wang, G.; Huang, D.A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.J.; Anandkumar, A. Eureka: Human-Level Reward Design via Coding Large Language Models, 2023, [arXiv:cs.RO/2310.12931].
335. Sun, S.; Liu, R.; Lyu, J.; Yang, J.W.; Zhang, L.; Li, X. A Large Language Model-Driven Reward Design Framework via Dynamic Feedback for Reinforcement Learning. *arXiv preprint arXiv:2410.14660* **2024**, [arXiv:cs.LG/2410.14660].
336. Wu, X.; Zheng, Z.; Xiong, H. LLM-ALSO: LLM-Driven Adaptive Learning-Signal Optimization for Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2605.29293* **2026**, [arXiv:cs.MA/2605.29293].
337. Hazra, R.; Sygkounas, A.; Persson, A.; Loutfi, A.; Martires, P.Z.D. REvolve: Reward Evolution with Large Language Models using Human Feedback. *arXiv preprint arXiv:2406.01309* **2024**, [arXiv:cs.NE/2406.01309].
338. Du, S.; Yan, X.; Shi, J.; Cao, Z.; Feng, S.; Liang, Z.; Sun, B.; Peng, T.; Zhou, Y.; Li, X.; et al. MLEvolve: A Self-Evolving Framework for Automated Machine Learning Algorithm Discovery. *arXiv preprint arXiv:2606.06473* **2026**, [arXiv:cs.AI/2606.06473].

 doi:10.20944/preprints202608.0051.v1

339. Wang, H.; Wu, Y.; Chang, D.; Wei, L.; Heldt, L. Self-Evolving Recommendation System: End-To-End Autonomous Model Optimization With LLM Agents, 2026, [arXiv:cs.LG/2602.10226].
340. Li, P.; Tang, H.; Qiao, J.; Zheng, Y.; Hao, J. LaRes: Evolutionary Reinforcement Learning with LLM-based Adaptive Reward Search. In Proceedings of the Advances in Neural Information Processing Systems, 2025.
341. Li, P.; Hao, J.; Tang, H.; Yuan, Y.; Qiao, J.; Dong, Z.; Zheng, Y. R*: Efficient Reward Design via Reward Structure Evolution and Parameter Alignment Optimization with Large Language Models. In Proceedings of the Proceedings of the 42nd International Conference on Machine Learning. PMLR, 2025, Vol. 267, *Proceedings of Machine Learning Research*, pp. 34509–34527.
342. Gao, N.; Zhang, X.; Jiang, X.; You, M.; Zhang, M.; Deng, Y. RF-Agent: Automated Reward Function Design via Language Agent Tree Search. *arXiv preprint arXiv:2602.23876* **2026**, [arXiv:cs.AI/2602.23876].
343. Huang, C.; Chang, Y.; Lin, J.; Liang, J.; Zeng, R.; Li, J. Efficient Language-instructed Skill Acquisition via Reward-Policy Co-Evolution. *arXiv preprint arXiv:2412.13492* **2024**, [arXiv:cs.RO/2412.13492].
344. Liu, Z.; Chai, J.; Zhu, X.; Tang, S.; Ye, R.; Zhang, B.; Bai, L.; Chen, S. ML-Agent: Reinforcing LLM Agents for Autonomous Machine Learning Engineering, 2025, [arXiv:cs.AI/2505.23723].
345. Zhang, Y.; Zhou, K.; Xu, Z.; Ramnath, K.; Zhou, Y.; Woo, S.; Ding, H.; Cheong, L.L. Learning to Ideate for Machine Learning Engineering Agents, 2026, [arXiv:cs.AI/2601.17596].
346. Wu, F.; Zheng, X.; Wang, Z.; ming Dai, Y.; Li, H. RHyVE: Competence-Aware Verification and Phase-Aware Deployment for LLM-Generated Reward Hypotheses, 2026, [arXiv:cs.LG/2604.28056].
347. Chen, M.; Xiao, B.; Liang, D.; Zeng, C.; Wen, Z. Efficient Hyperparameter Optimization for LLM Reinforcement Learning. *arXiv preprint arXiv:2606.03073* **2026**, [arXiv:cs.LG/2606.03073].
348. Lu, C.; Holt, S.; Fanconi, C.; Chan, A.J.; Foerster, J.; van der Schaar, M.; Lange, R.T. Discovering Preference Optimization Algorithms with and for Large Language Models. In Proceedings of the Advances in Neural Information Processing Systems, 2024, [2406.08414].
349. Cheng, S.; Li, T.; Huang, X.; Yin, X.; Zou, D. Differentiable Evolutionary Reinforcement Learning, 2026, [arXiv:cs.AI/2512.13399].
350. Ahmadi, A.; Sharif, S.; Banad, Y. Enhanced LLM Reasoning by Optimizing Reward Functions with Search-Driven Reinforcement Learning, 2026, [arXiv:cs.CL/2605.02073].
351. Han, X.; Yang, Q.; Chen, X.; Chu, X.; Zhu, M. Generating and Evolving Reward Functions for Highway Driving with Large Language Models. *arXiv preprint arXiv:2406.10540* **2024**, [arXiv:cs.AI/2406.10540].
352. Wei, Y.; Shan, X.; Li, J. LERO: LLM-driven Evolutionary framework with Hybrid Rewards and Enhanced Observation for Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2503.21807* **2025**, [arXiv:cs.LG/2503.21807].
353. Jiang, Z.; Schmidt, D.; Srikanth, D.; Xu, D.; Kaplan, I.; Jacenko, D.; Wu, Y. AIDE: AI-Driven Exploration in the Space of Code. *arXiv preprint arXiv:2502.13138* **2025**, [arXiv:cs.AI/2502.13138].
354. Pepe, A.; Lin, C.Y.; Magka, D.; Acun, B.; Wu, Y.N.; Protopopov, A.; Wu, C.J.; Bachrach, Y. Agentic Discovery of Neural Architectures: AIRA-Compose and AIRA-Design, 2026, [arXiv:cs.AI/2605.15871].
355. Jeddi, A.; Le, M.N.; Karaimer, H.C.; Derpanis, K.G.; Taati, B. GEAR: Genetic AutoResearch for Agentic Code Evolution, 2026, [arXiv:cs.AI/2605.13874].
356. Jiang, J.; Zhu, H.; Zhu, Z. SMCEvolve: Principled Scientific Discovery via Sequential Monte Carlo Evolution, 2026, [arXiv:cs.AI/2605.15308].
357. Si, C.; Yang, Z.; Choi, Y.; Candès, E.; Yang, D.; Hashimoto, T. Towards Execution-Grounded Automated AI Research, 2026, [arXiv:cs.AI/2601.14525].
358. Yuan, S.; Chen, Z.; Xi, Z.; Ye, J.; Du, Z.; Chen, J. Agent-R: Training Language Model Agents to Reflect via Iterative Self-Training, 2025, [arXiv:cs.AI/2501.11425].
359. Chen, M.; Lv, C.C.; Zhang, G.; Chang, H.; Zhou, S. HarnessForge: Joint Harness and Policy Evolution for Adaptive Agent Systems. 2026.
360. Iacob, A.; Jovanovic, A.; Shen, W.F.; Burkhardt, D.; Kurmanji, M.; Tastan, N.; Sani, L.; Venanzi, N.A.E.; Odonnat, A.; Cao, Z.; et al. The Red Queen Gödel Machine: Co-Evolving Agents and Their Evaluators, 2026, [arXiv:cs.LG/2606.26294].
361. Hu, J.; Shukla, P.; Huang, K. Verifying the Verifiers: Failure Attribution for Agentic Benchmark Diagnostics and Training Data Curation. In Proceedings of the ICLR 2026 Workshop on AI with Recursive Self-Improvement (RSI), 2026.
362. Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; Zhou, D. Self-Consistency Improves Chain of Thought Reasoning in Language Models, 2023, [arXiv:cs.CL/2203.11171].

 doi:10.20944/preprints202608.0051.v1

363. Chen, X.; Aksitov, R.; Alon, U.; Ren, J.; Xiao, K.; Yin, P.; Prakash, S.; Sutton, C.; Wang, X.; Zhou, D. Universal Self-Consistency for Large Language Model Generation, 2023, [arXiv:cs.CL/2311.17311].
364. Wang, Z.; Wang, K.; Wang, Q.; Zhang, P.; Li, L.; Yang, Z.; Jin, X.; Yu, K.; Nguyen, M.N.; Liu, L.; et al. RAGEN: Understanding Self-Evolution in LLM Agents via Multi-Turn Reinforcement Learning, 2025, [arXiv:cs.LG/2504.20073].
365. Yi, B.; Liu, Q.; Cheng, Y.; Xu, H. Escaping Model Collapse via Synthetic Data Verification: Near-term Improvements and Long-term Convergence, 2026, [arXiv:stat.ML/2510.16657].
366. Xia, P.; Chen, J.; Wang, H.; Liu, J.; Zeng, K.; Wang, Y.; Han, S.; Zhou, Y.; Zhao, X.; Chen, H.; et al. SkillRL: Evolving Agents via Recursive Skill-Augmented Reinforcement Learning. *CoRR* **2026**, *abs/2602.08234*, [2602.08234]. https://doi.org/10.48550/ARXIV.2602.08234.
367. Li, Y.; Miao, R.; Qi, Z.; Lan, T. ARISE: Agent Reasoning with Intrinsic Skill Evolution in Hierarchical Reinforcement Learning. *CoRR* **2026**, *abs/2603.16060*, [2603.16060]. https://doi.org/10.48550/ARXIV.2603.16060.
368. Hebbar, P.; Manawat, Y.; Verboomen, S.; Ivanova, A.; Palanimalai, S.; Bhatia, K.; Baskaran, V. SIA: Self Improving AI with Harness & Weight Updates. *CoRR* **2026**, *abs/2605.27276*, [2605.27276]. https://doi.org/10.48550/ARXIV.2605.27276.
369. Li, Y.; Zhang, Y.; Zhang, X.; Liu, X.; Liu, Y. CODESKILL: Learning Self-Evolving Skills for Coding Agents. *CoRR* **2026**, *abs/2605.25430*, [2605.25430]. https://doi.org/10.48550/ARXIV.2605.25430.
370. Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; Anandkumar, A. Voyager: An Open-Ended Embodied Agent with Large Language Models. *CoRR* **2023**, *abs/2305.16291*, [2305.16291]. https://doi.org/10.48550/ARXIV.2305.16291.
371. Xia, C.S.; Wang, Z.; Yang, Y.; Wei, Y.; Zhang, L. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? *CoRR* **2025**, *abs/2511.13646*, [2511.13646]. https://doi.org/10.48550/ARXIV.2511.13646.
372. Lou, X.; Lázaro-Gredilla, M.; Dedieu, A.; Wendelken, C.; Lehrach, W.; Murphy, K.P. AutoHarness: improving LLM agents by automatically synthesizing a code harness. *CoRR* **2026**, *abs/2603.03329*, [2603.03329]. https://doi.org/10.48550/ARXIV.2603.03329.
373. Fang, T.; Zhang, H.; Zhang, Z.; Ma, K.; Yu, W.; Mi, H.; Yu, D. WebEvolver: Enhancing Web Agent Self-Improvement with Coevolving World Model, 2025, [arXiv:cs.CL/2504.21024].
374. Guo, Y.; Lee, T.; Shi, L.X.; Chen, J.; Liang, P.; Finn, C. VLAW: Iterative Co-Improvement of Vision-Language-Action Policy and World Model, 2026, [arXiv:cs.RO/2602.12063].