Position: Agentic Evolution is the Path to Evolving LLMs

Minhua Lin
mfl5681@psu.edu
The Pennsylvania State University
University Park, USA

Hanqing Lu
luhanqin@amazon.com
Amazon
Palo Alto, USA

Zhan Shi
Amazon
Palo Alto, USA

Bing He
Amazon
Palo Alto, USA

Rui Mao
Amazon
Palo Alto, USA

Zhiwei Zhang
The Pennsylvania State University
University Park, USA

Zongyu Wu
The Pennsylvania State University
University Park, USA

Xianfeng Tang
Amazon
Palo Alto, USA

Hui Liu
Amazon
Palo Alto, USA

Zhenwei Dai
Amazon
Palo Alto, USA

Rui Zhang
Independent Researcher

Xiang Zhang
The Pennsylvania State University
University Park, USA

Suhang Wang
szw494@psu.edu
The Pennsylvania State University
University Park, USA

Benoit Dumoulin
Amazon
Palo Alto, USA

Jian Pei
Duke University
Durham, USA

Abstract

As Large Language Models (LLMs) move from curated training sets into open-ended real-world environments, a fundamental limitation emerges: static training cannot keep pace with continual deployment environment change. Scaling training-time and inference-time compute improves static capability but does not close this train–deploy gap. We argue that addressing this limitation requires a new scaling axis—*evolution*. Existing deployment-time adaptation methods, whether parametric fine-tuning or heuristic memory accumulation, lack the strategic agency needed to diagnose failures and produce durable improvements. Our position is that **agentic evolution represents the inevitable future of LLM adaptation**, elevating evolution itself from a fixed pipeline to an autonomous evolver agent. We instantiate this vision in a **general framework**, A-Evolve, which treats deployment-time improvement as a deliberate, goal-directed optimization process over persistent system state. We further propose the **evolution-scaling hypothesis**: the capacity for adaptation scales with the compute allocated to evolution, positioning agentic evolution as a scalable path toward sustained, open-ended adaptation in the real world.

Keywords

Self Evolution, Agentic Evolution, Self-improving LLM Agents, Continual Learning, Large Language Models

1 Introduction

Large Language Models (LLMs) [6, 29, 36] have achieved remarkable progress by scaling along two primary axes: increasing training-time compute (spanning pre-training and post-training) [21, 22] and scaling inference-time compute via reasoning chains [34, 41]. However, real-world applications of LLMs in open-ended environments still face a fundamental challenge: the **train-deploy environment gap** [12, 16]. Once deployed, models trained with finite training data cannot exhaustively anticipate the infinite variety of real-world cases, shifting APIs, and evolving constraints. Therefore, purely static models inevitably degrade or fail under prolonged deployment.

This motivates the need for a new scaling axis: *evolution*—the deployment-time ability of a model to autonomously improve its capabilities during interaction, without manual intervention. We define **evolution** as *continual learning for LLM systems during deployment*. An LLM system's behavior is governed by a composite policy $\pi = (\pi_\theta, \pi_S)$, where π_θ denotes the parametric backbone (e.g., LLM weights) and π_S is a non-parametric *persistent artifact state*, such as tools [45], code [20], memories [10], and structured knowledge [15]. Evolution corresponds to cross-episode policy improvement driven by accumulated experience:

$$(\pi_\theta^{t+1}, \pi_S^{t+1}) \leftarrow F_{\text{Evolve}}(\pi_\theta^t, \pi_S^t, \text{Obs}[1:t]), \tag{1}$$

where $\text{Obs}[1:t]$ is deployment observations (e.g., interaction traces, environment feedback, rewards), and F_{Evolve} is the update mechanism. In this view, evolution is not a one-shot training procedure, but an *ongoing process* that converts interaction evidence into lasting behavioral improvement.

Evolution is essential for LLM systems for at least three important reasons. First, the train-deploy gap implies that static optimization over fixed distributions is fundamentally insufficient; adaptation must occur *in situ*. Second, many deployment settings impose privacy and governance constraints: user feedback, proprietary data, or sensitive interactions cannot be centrally logged for global retraining. Evolution enables *local*, governed improvement

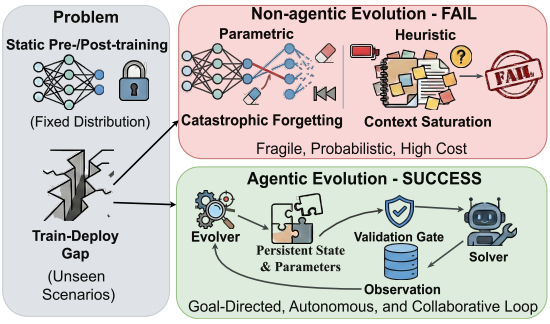


Figure 1: From train-deploy gap to agentic evolution: An overview.

through persistent artifacts without leaking private data. Third, test-time compute alone does not scale. While additional reasoning can solve novel instances, it is wasteful for recurrent failures. When a deployed LLM system repeatedly lacks a capability, such as parsing a new file format or handling a brittle API, thinking longer each time is inferior to learning once and reusing the solution. Instead, evolution amortizes expensive reasoning into cheap, persistent capability.

While several pioneering approaches to LLM evolution have emerged, they remain fundamentally limited in achieving this goal. **Parametric evolution** methods [17, 18], such as test-time training or online fine-tuning, update π_θ using recent observations. While expressive in principle, these updates are opaque and difficult to govern, and they risk catastrophic forgetting under distribution shift [24]. **Non-parametric heuristic evolution** methods [13] instead modify π_S by appending textual memories [40] or optimizing prompts *using fixed rules* [51]. Although lightweight, these approaches treat evolution as a storage or search problem: as experience accumulates, memory saturates with noisy, unverified text, and improvements exhibit diminishing returns. In both cases, the update rule F_{Evolve} is static and heuristic, rather than adaptive and goal-directed.

These limitations are not incidental, but stem from a common root cause: the *absence of agentic capability in the evolution process itself*. In existing approaches, F_{Evolve} is either a static optimization rule (e.g., gradient updates [45]) or a fixed heuristic (e.g., append-and-retrieve [32]). Such mechanisms lack the ability to *reason about* failures, *decide* which aspects of the system should change, and *adapt* their own update strategy as deployment conditions evolve. As a result, parametric methods blindly modify weights without semantic accountability, while heuristic non-parametric methods indiscriminately accumulate experience without understanding relevance, causality, or long-term utility. Both fail precisely because evolution is treated as a mechanical procedure rather than an agentic decision-making problem.

We therefore propose **our position** in this paper: **LLM evolution has to be agentic**. The novel paradigm of **agentic evolution** is shown in Fig. 1. The central idea is to elevate F_{Evolve} from a fixed, heuristic-driven workflow to an *explicit evolver agent*—a goal-directed optimizer that diagnoses what to change, autonomously governs when to change, and collaboratively synthesizes composable updates to maintain and improve both the parametric backbone π_θ and the persistent artifact state π_S.

Consider a cloud platform agent deployed to compute routine metrics, such as p95 latency per endpoint, from raw production logs. At first, everything works smoothly—until the environment changes. A logging update renames a field, introduces a nested JSON structure, or slightly alters an interface that the agent quietly depended on. The next day, the agent starts failing. Existing evolution strategies respond in blunt ways. *Parametric approaches* attempt retraining, incurring high costs and risking *catastrophic forgetting* [24]. *Non-parametric heuristic methods* instead record the failure as text, hoping the agent will re-discover the correct logic, but often face *context saturation* and inconsistent retrieval. In contrast, **agentic evolution** takes a different view. The evolver treats failures as diagnostic signals, identifies *what* needs to change, and uses a *validation gate* to verify the fix (e.g., a regression-tested parser), ensuring the update is persistent and safe. This shift from reactive patching to deliberate, goal-directed evolution captures the essence of agentic evolution and motivates the three principles we introduce next.

Agentic evolution is characterized by three core principles. First, the **goal-oriented principle** specifies *what* to change: the evolver actively diagnoses deployment failures, attributes them to underlying causes, and targets persistent artifacts whose modification is expected to yield durable performance gains. Second, the **autonomy principle** specifies *when* to change: rather than following a fixed update schedule, the evolver governs the adaptation process by selecting relevant evidence, triggering evolution only when warranted, and explicitly deciding whether to commit or reject candidate updates. Finally, the **compositional principle** specifies *how* to change: improvements are realized through structured, modular artifacts—such as tools, workflows, and validation tests—produced by decomposed decision processes and integrated only after verification, while allowing periodic updates to non-parametric components when appropriate. These principles elevate evolution from a static heuristic pipeline to a deliberate, governed, and scalable decision-making process.

More broadly, agentic evolution reframes deployment-time adaptation as the joint optimization of persistent state and model parameters, rather than relying solely on the direct tuning of weights or the passive accumulation of memory. Under this view, the update rule itself becomes a *budgeted optimizer*: an explicit evolver agent that decides *what* to change, *when* to change, and *how* to change under a finite evolution-time compute budget. This leads to our central vision: **Agentic evolution represents a scalable path for the sustained, open-ended evolution of LLM systems over indefinite deployment horizons**. Unlike static heuristics that inevitably plateau, a sufficiently capable evolver can continue to extract increasingly complex improvements from accumulated experience. Continuous, open-ended evolution demands both sustainability and scalability. This motivates the **evolution-scaling hypothesis**: *the capacity for adaptation—the achievable performance frontier of evolution—scales with the compute allocated to the evolution process*. We identify this as a third scaling axis, complementary

to training-time and inference-time computation. Ultimately, adaptive intelligence in LLM systems depends not on chance, but on systematically scaling the ability to *learn how to improve*.

2 Agentic Evolution and Principles

We consider a deployed *LLM system* that repeatedly interacts with an environment over a sequence of episodes. At episode t, the system observes an input or state x_t, produces actions a_t, and receives feedback o_t from the environment. We model the system as a *composite policy*

$$\pi_t = (\pi_{\theta,t}, \pi_{S,t}), \tag{2}$$

where $\pi_{\theta,t}$ denotes the parametric backbone (e.g., an LLM model with parameters θ), and $\pi_{S,t}$ denotes a persistent, editable artifact state that conditions behavior across episodes, such as tools, skills, workflows, structured knowledge, and validation assets. Crucially, π_t persists beyond a single interaction and constitutes the system's deployment-time interface to capability.

Evolution refers to cross-episode improvement that occurs *during deployment*. Rather than treating deployment as a static inference phase, evolution formalizes the process by which a system converts accumulated interaction evidence into lasting behavioral change. Let $\mathrm{Obs}_{1:t}$ denote the deployment evidence collected up to episode t, including trajectories, tool traces, errors, and feedback. Evolution is defined as a cross-episode update process:

$$(\pi_{\theta,t+1}, \pi_{S,t+1}) \leftarrow F_{\mathrm{Evolve}}(\pi_{\theta,t}, \pi_{S,t}, \mathrm{Obs}_{1:t}), \tag{3}$$

where F_{Evolve} is an update mechanism that may modify the parametric model π_θ, the persistent state π_S, or both. This definition abstracts over a broad spectrum of approaches, ranging from offline training and online fine-tuning to heuristic memory updates. Detailed taxonomy is in Appendix A.

Agentic evolution instantiates F_{Evolve} as an explicit *evolver agent*: a goal-directed, autonomous optimizer that reasons over accumulated deployment evidence and produces *persistent, governed* updates. Rather than executing a fixed script, the evolver treats evolution itself as a decision-making problem. At each episode t, it proposes a structured candidate update Δ_t, such as add, patch, refactor, or prune operations over π_S and/or π_θ, and makes an explicit commit decision:

$$\Delta_t \leftarrow F_{\mathrm{Evolve}}(\pi_{\theta,t}, \pi_{S,t}, \mathrm{Obs}_{1:t}),\ c_t \leftarrow C(\pi_t, \Delta_t, \mathrm{Obs}_{1:t}), \tag{4}$$

followed by

$$\pi_{t+1} \leftarrow \begin{cases} \mathrm{Apply}(\pi_t, \Delta_t) & \text{if } c_t = 1, \\ \pi_t & \text{if } c_t = 0, \end{cases} \tag{5}$$

where $c_t \in \{0, 1\}$ is the commit decision from the governance gate C. This gate is typically instantiated via automated verification (e.g., unit tests, regression checks) or human-in-the-loop review. The defining characteristic of agentic evolution is that updates are treated as *conditional decisions*—explicitly proposed, evaluated, and either committed or rejected—rather than as unconditional steps in a fixed workflow.

Agentic evolution is governed by three core principles that specify *what* to evolve, *when* to evolve, and *how* evolution is carried out.

The **goal-oriented principle** specifies *what* to change. The evolver does not blindly accumulate experience or optimize generic proxies; instead, it explicitly diagnoses deployment failures, attributes them to actionable causes, and targets specific components of the persistent state whose modification is expected to improve *future* performance. This shifts adaptation from correlation-based updates to causal, capability-level repair. In practice, goal orientation is realized by localizing failures to missing tools, brittle logic, interface mismatches, or incomplete workflows, and then applying targeted edits to the corresponding artifacts in π_S (and, when appropriate, to π_θ). By making the adaptation objective explicit and forward-looking, the system converts raw experience into durable improvements rather than transient fixes.

The **autonomy principle** specifies *when* to change. Instead of following a pre-defined update schedule (e.g., evolve after every failure), the evolver controls the update decision process itself. It selects which evidence is relevant, determines whether a failure is actionable or merely transient noise, and explicitly decides whether to commit an update or perform a no-op. This decision-theoretic control allows evolution compute to be allocated only when improvement is feasible and worthwhile. Autonomy distinguishes agentic evolution from non-agentic pipelines in which updates are implicitly triggered or hard-coded, and it is essential for stable long-horizon deployment in open-ended environments.

The **compositional principle** specifies *how* to evolve. Internally, the evolver is naturally decomposed into cooperating decision functions—such as diagnosis, planning, updating, and verification—that operate over shared evidence and state. This decomposition improves reliability and expressivity, whether realized within a single system or as a multi-agent workflow. Externally, evolution produces *modular, structured artifacts*—for example, executable tools, reusable workflows, and validation tests—rather than unstructured text. This compositionality enables *amortization*: recurring reasoning and fragile deliberation are compiled into reusable capabilities, allowing the system to bypass context saturation and avoid the diminishing returns typical of heuristic memory accumulation.

These principles elevate evolution from a static, heuristic-driven procedure to a governed and scalable decision-making process. By treating deployment-time improvement as an agentic optimization problem over persistent state, agentic evolution provides a foundation for LLM systems that can reliably adapt to open-ended environments while maintaining stability, interpretability, and scalability.

To illustrate, let us revisit the cloud-log agent scenario from Sec. 1. Let $\pi_t = (\pi_{\theta,t}, \pi_{S,t})$ denote the composite policy. Under the schema drift scenario (e.g., nested JSON changes), parametric evolution attempts to update the backbone π_θ via fine-tuning, entangling a localized interface mismatch with global model behavior and risking catastrophic forgetting. Heuristic methods instead append raw traces to π_S, forcing the solver to repeatedly regenerate brittle logic at inference time with limited reuse or guarantees.

Agentic evolution reframes such failures as a cross-episode optimization problem. Deployment trajectories are aggregated into structured evidence, from which an explicit evolver diagnoses the underlying cause, such as an outdated schema assumption, and formulates a targeted update objective. Rather than modifying weights

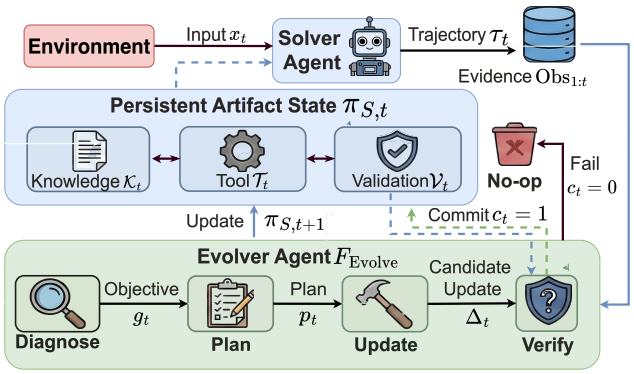


Figure 2: Framework of A-Evolve.

or appending raw memories, the evolver proposes precise, structured edits to π_S, such as synthesizing a versioned adapter function to permanently handle the new nested format and rectifying the API schema definition. Candidate updates are evaluated and committed conditionally through explicit verification, ensuring that only validated improvements persist. In this way, agentic evolution amortizes repeated inference-time reasoning into durable, governed capability, transforming recurrent failures into stable system assets.

3 A-Evolve: A General Framework for Agentic Evolution of LLM Systems

A-Evolve is a general, implementation-agnostic framework that instantiates the principles of agentic evolution introduced in Sec. 2. As illustrated in Fig. 2, A-Evolve makes deployment-time evolution explicit, governed, and amortizable, enabling scalable capability improvement beyond static inference.

A-Evolve is built around three commitments. First, evolution is *goal-oriented*: from deployment evidence, the system derives an explicit update objective g and proposes targeted edits Δ over a structured space. Second, evolution is *autonomous*: updates are conditional, driven by explicit evidence selection and a commit/no-op decision c, rather than a fixed schedule. Third, evolution is *compositional*: the update rule F_{Evolve} is realized as a modular evolver that produces typed artifacts and commits them only through explicit acceptance mechanisms such as validation or review. We instantiate these commitments through three structural components: a persistent artifact state, a solve–evolve control loop, and an explicit evolver.

Persistent Artifact State π_S. A-Evolve elevates the non-parametric component π_S to a first-class, editable interface to system capability. Instead of accumulating transient context, deployment experience is compiled into persistent artifacts that can be created, revised, validated, and reused across episodes. This exposes a concrete edit space for evolution, allowing targeted updates to specific components rather than diffuse prompt modification. At episode t, the artifact state is

$$\pi_{S,t} = \{\mathcal{K}_t, \mathcal{T}_t, \mathcal{V}_t\}. \tag{6}$$

The **knowledge registry** $\mathcal{K}_t$ stores structured or textual artifacts such as schemas, workflows, interface contracts, and exemplars. Artifacts are addressable and versioned, enabling retrieval, patching, and replacement. This supports goal-oriented evolution by allowing failures to be localized to specific knowledge components.

The **tool registry** $\mathcal{T}_t$ contains executable functions, including scripts and API wrappers, with explicit input–output signatures and associated tests. During solve-time, tools provide deterministic action primitives that reduce inference variance. During evolve-time, they serve as diagnostic instruments for replaying failures, probing edge cases, and extracting structured error signals.

The **validation registry** $\mathcal{V}_t$ contains governance assets such as unit tests, regression suites, and optional human review hooks. Validation artifacts are themselves editable. Crucially, $\mathcal{V}_t$ grounds the commit decision c_t: updates are committed only if they pass verification, preventing regressions and uncontrolled drift over long deployment horizons.

The Solve–Evolve Loop. A-Evolve explicitly separates instance-level task execution from cross-episode capability improvement, ensuring that evolution is a governed process with its own control flow and compute budget. For episode t with input x_t, the solve phase produces a trajectory

$$\tau_t = \mathrm{Solve}(\pi_t, x_t), \tag{7}$$

which is appended to the cumulative evidence buffer $\mathrm{Obs}_{1:t} = \mathrm{Obs}_{1:t-1} \cup \{\tau_t\}$. The evolve phase then produces a conditional update

$$\pi_{t+1} = \pi_t \oplus (c_t \cdot \Delta_t), \tag{8}$$

where Δ_t and c_t are defined below.

In the **solve phase**, the system executes under the composite policy $\pi_t = (\pi_{\theta,t}, \pi_{S,t})$, retrieving artifacts from $\mathcal{K}_t$ and invoking tools from $\mathcal{T}_t$ as needed. The artifact state $\pi_{S,t}$ is treated as read-only to ensure reproducibility.

In the **evolve phase**, triggered asynchronously or when sufficient evidence accumulates, the evolver analyzes $\mathrm{Obs}_{1:t}$ and proposes a structured update Δ_t together with a commit decision c_t. The evolver may invoke tools from $\mathcal{T}_t$ to reproduce failures or analyze traces. Candidate updates are evaluated against $\mathcal{V}_t$; only validated updates are committed. This separation ensures that solve-time compute targets task completion, while evolve-time compute targets amortized improvement across episodes.

The Evolver F_{Evolve}. A-Evolve implements F_{Evolve} as an explicit evolver composed of four cooperating functions:

$$\begin{aligned} g_t &\leftarrow \mathrm{Diagnose}(\mathrm{Obs}_{1:t}, \pi_{S,t}), \quad p_t \leftarrow \mathrm{Plan}(g_t, \pi_{S,t}), \\ \Delta_t &\leftarrow \mathrm{Update}(p_t, \pi_{S,t}), \quad c_t \leftarrow \mathrm{Verify}(\Delta_t, \mathcal{V}_t). \end{aligned} \tag{9}$$

Diagnose identifies actionable failure modes and their causes (e.g., missing schemas or brittle tools), optionally invoking $\mathcal{T}_t$ to extract structured error signatures, and produces an update objective g_t. **Plan** translates g_t into an explicit edit plan specifying target artifacts, edit operators (add, patch, refactor, prune), and ordering constraints. **Update** executes the plan by synthesizing concrete artifact changes, assembling a candidate update Δ_t with provenance and tests. **Verify** evaluates Δ_t against $\mathcal{V}_t$ and returns a commit decision $c_t \in \{0, 1\}$, allowing the evolver to reject harmful or brittle updates.

Edit Operators and Auditing. A-Evolve supports a small set of canonical edit operators over π_S, including adding or patching tools, schemas, and tests, and pruning obsolete artifacts. For example, under schema drift, the evolver may add a new schema, patch a parser tool, and introduce regression tests, committing the update only if all checks pass. All proposed updates are logged with full provenance, enabling auditing, rollback, and risk-based human oversight.

Summary. A-Evolve provides a concrete framework for agentic evolution. By exposing a typed persistent artifact state, separating solve-time execution from evolve-time optimization, and enforcing modular, verifiable updates, A-Evolve enables LLM systems to convert deployment experience into durable, governed capability improvements without sacrificing stability or interpretability.

4 The Evolution-Scaling Hypothesis

A central motivation for agentic evolution is to enable *autonomy*: the ability of an LLM system to adapt reliably in open-ended, non-stationary environments where increased training-time compute [21, 22] and increased test-time compute [30] inevitably fall short. While existing evolution-style methods [7, 26] demonstrate that deployment-time adaptation is possible, they are typically driven by ad-hoc heuristics and fixed update rules. As a result, their effectiveness is often fragile, difficult to predict, and prone to early saturation.

This leads to a fundamental question for long-horizon deployment and, more broadly, for the path toward AGI: *Can evolution itself be made scalable and sustainable?* Specifically, if we allocate more resources to the evolution process—such as stronger evolvers, more analysis steps, or larger evolution-time compute budgets—does the effectiveness and speed of adaptation increase in a predictable and systematic way, or is improvement largely bounded by chance and problem-specific heuristics?

The Evolution-Scaling Hypothesis. We propose the **Evolution-Scaling Hypothesis**, which frames deployment-time adaptation not as a stochastic or opportunistic phenomenon, but as a scalable optimization process governed by resources. Analogous to established scaling laws for pre-training and inference, the hypothesis posits that allocating additional compute to the *evolution loop*—rather than only to solving individual tasks—systematically improves the attainable level of adaptation.

Concretely, we argue that evolution-time compute enables an LLM system to (i) diagnose failures more accurately, (ii) consider more candidate updates, (iii) synthesize more robust artifacts, and (iv) apply stronger verification before committing changes. These capabilities compound over episodes because validated updates persist. As a result, evolution is not merely a collection of local fixes, but a convergent process whose effectiveness scales with the resources devoted to it. This perspective shifts the paradigm from hoping that heuristics generalize to *engineering adaptation through compute.*

Evolution-Scaling Formalism. To make this notion precise, we define a compute-optimal view of evolution. Let $P(\pi)$ denote the performance of a deployed policy π under a fixed environment and a fixed solve-time compute budget. Given an initial policy π_0, we define the *compute-optimal evolution frontier* as

$$P^*(C_{\text{evolve}}, \pi_0) \triangleq \max_{F_{\text{evolve}}} \mathbb{E}_{\pi \sim F_{\text{evolve}}(\pi_0)} \big[P(\pi)\big], \tag{10}$$

where F_{evolve} ranges over all evolution strategies whose total evolution-time compute cost does not exceed C_{evolve}. Here, C_{evolve} captures the resources allocated to the evolution process, including analysis steps, tool invocations, candidate synthesis, and verification, while implicitly reflecting the capability of the evolver itself.

The Evolution-Scaling Hypothesis states that, holding the environment and solve-time compute fixed, the frontier P^* is strictly increasing with respect to evolution-time compute. Formally, for any $C^{(1)}_{\text{evolve}} < C^{(2)}_{\text{evolve}}$, we hypothesize:

$$P^*(C^{(1)}_{\text{evolve}}, \pi_0) \;<\; P^*(C^{(2)}_{\text{evolve}}, \pi_0). \tag{11}$$

Intuitively, allocating more evolution-time compute enlarges the space of feasible update strategies, enabling deeper diagnosis, more reliable edits, and stronger governance, and thereby raising the achievable performance ceiling.

Interpretation. This hypothesis reframes deployment-time learning as a *predictable scaling regime.* Rather than viewing adaptation as a series of isolated, heuristic-driven repairs, evolution is treated as an optimization process whose convergence rate and asymptotic performance depend on the strength and compute budget of the evolver. In this view, insufficient evolution resources lead to noisy, brittle improvements, while sufficient resources guarantee systematic progress toward the compute-optimal frontier.

Strategic Implications. The evolution-scaling perspective yields two complementary research directions:

Approaching the frontier. For a fixed evolution-time budget, the goal is to design evolution algorithms F_{evolve} that efficiently approach P^*. Agentic evolution advances this direction by replacing fixed heuristics with autonomous, structured decision-making, as empirically validated in Sec. 5.2.

Raising the frontier. Beyond algorithmic efficiency, the hypothesis predicts that aggressively scaling evolution-time resources, such as evolver capability and analysis depth, systematically raises the performance ceiling itself. In long-horizon deployment, reallocating compute from repeatedly "thinking harder" at solve time to "evolving better" across episodes converts raw compute into durable capability, enabling mastery of increasingly complex and shifting environments. We will empirically observe the evolution scaling effect in Sec. 5.4.

5 Empirical Studies

In this section, we empirically evaluate the feasibility and advantages of agentic evolution by addressing three questions: (i) **Effectiveness:** does agentic evolution improve task performance under *compute-matched* conditions? (ii) **Feasibility:** how do the core components of agentic evolution contribute to performance gains and loop reliability? (iii) **Evolution-scaling:** does evolution capacity increase with evolution-time compute and evolver capability?

5.1 Experimental Setup

Datasets. We evaluate on AppWorld [37], a widely used benchmark for tool-using agents with executable environments and unit tests

for goal verification. We sample 50 tasks from the training split for evolution and 50 tasks from the test-normal split for evaluation. Dataset details and the sampling protocol are in Appendix B.1.

Implementation Details. We implement A-Evolve as a two-level system consisting of a *solver*, which executes tasks, and an *evolver*, which accumulates persistent improvements across episodes. The evolver decomposes the evolve step into four roles: diagnoser, planner, updater, and verifier, which cooperate through a shared evolution state. Unless otherwise specified, we use Claude Sonnet 4.5 [2] as the evolver backbone, and evaluate solvers ranging from Claude Haiku 4.5 [1] and Claude Sonnet 4/4.5 to GPT-5 [33] and Gemini 3 Flash [14]. Additional details are in Appendix B.2.

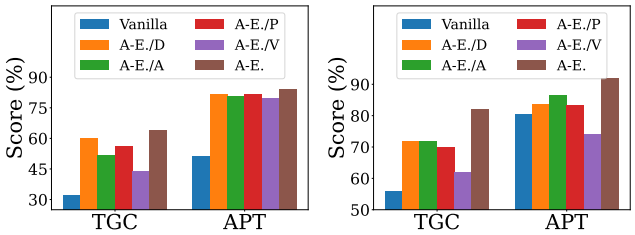
Evaluation Protocol. Following prior work [8, 37], we report two metrics: *Task Goal Completion* (TGC), the fraction of tasks successfully completed, and *Average Passed Tests* (APT), the average fraction of unit tests passed per task. To ensure fair comparison, we fix a solve-time compute budget C_{solve} (maximum tool calls, steps, or tokens per task) and an evolve-time compute budget C_{evolve} (maximum tokens and tool invocations per episode) across all methods. Full evaluation details are given in Appendix B.3.

5.2 Effectiveness: Agentic vs. Non-agentic Evolution

Setting. We compare A-Evolve against a *non-evolving* reference and two representative *non-agentic evolution* baselines and: (i) *APE* [51], a search-based prompt evolution method that proposes candidate instructions and selects among them via task-level scoring; (ii) *AWM* [40], an experience-based workflow memory baseline that induces reusable workflows from past trajectories; and (iii) *Vanilla*, which directly queries the solver without persistent updates. All evolution methods learn updates on the training set and are evaluated on the test set.

Results. Tab. 1 reports TGC and APT across methods. We observe two consistent trends. *(i) Agentic evolution acts as a capability multiplier.* A-Evolve yields substantial gains across all solver backbones. For example, it achieves 64% and 82% TGC on Claude Haiku 4.5 and Gemini 3 Flash, respectively, compared to 46%/52% for AWM and 32%/56% for Vanilla. This demonstrates that agentic evolution reliably converts deployment evidence into reusable improvements that static prompt or workflow updates fail to capture. *(ii) Narrowing the capacity gap.* Smaller solvers augmented with A-Evolve can match or exceed larger vanilla models (e.g., Haiku 4.5 with A-Evolve reaches 64% TGC versus 42% for vanilla Sonnet 4), indicating that persistent procedural refinement across episodes can rival gains from stronger backbones alone.

Case Studies. Qualitative analysis of evolution traces reveals why these gains arise. Non-agentic baselines such as AWM tend to capture only *surface-level patterns*, retrieving raw trajectories that often include irrelevant context or hallucinated reasoning. In contrast, A-Evolve performs *active diagnosis*: the evolver invokes analysis tools to identify underlying failure causes (e.g., hidden API dependencies). These insights are synthesized into *persistent artifacts* in π_S, such as verified tools ($\mathcal{T}$) or refined knowledge schemas ($\mathcal{K}$), rather than stored as raw text. All updates are gated by the validation registry ($\mathcal{V}$), ensuring that only improvements passing regression checks are committed. At solve-time, this replaces fragile, probabilistic regeneration with robust, governed capability invocation. Detailed examples are provided in Appendix C.1.



(a) Claude Haiku 4.5 **(b) Gemini 3 Flash**

Figure 3: Ablation studies of A-Evolve **using Claude Haiku 4.5 and Gemini 3 Flash as solvers.**

5.3 Feasibility: Reliability of Agentic Evolution Loop

Setting. To isolate the contribution of each component in agentic evolution, we compare A-Evolve against four ablated variants: (i) *A-Evolve/D*, which removes **diagnosis** and proposes updates directly from raw trajectories; (ii) *A-Evolve/A*, which retains diagnosis but disables **analysis tools** for aggregating evidence across episodes; (iii) *A-Evolve/P*, which removes **planning** and generates updates in a single step without explicit edit structure; and (iv) *A-Evolve/V*, which removes **verification** and commits updates without validation gating. Claude Sonnet 4.5 is fixed as the evolver, with Claude Haiku 4.5 and Gemini 3 Flash as solvers. Other settings follow Sec. 5.2.

Results Analysis. Fig. 3 shows two consistent trends. *(i) The full loop is strongest.* The complete A-Evolve achieves the best performance across solvers. Removing any single component degrades results, although all ablations still outperform the vanilla solver, highlighting the complementary roles of the modules and the importance of end-to-end coupling. *(ii) Verification is a critical stabilizer.* Among all variants, *A-Evolve/V* degrades the most. Committing updates without validation allows brittle or overfit patches to accumulate, leading to regressions and unstable long-horizon behavior.

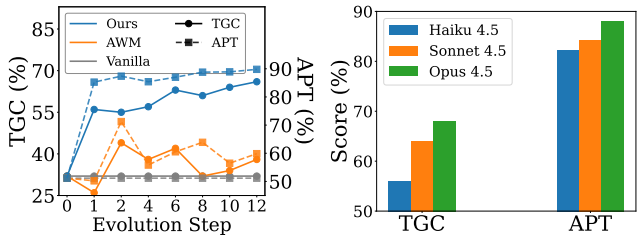
Case Studies. Qualitative analysis reveals distinct failure modes for each ablation. Without *diagnosis* (*A-Evolve/D*), the evolver acts blindly, producing superficial patches that mask errors rather than fixing root causes. Without *analysis tools* (*A-Evolve/A*), updates rely on single-trajectory inference and capture only local symptoms, missing systematic patterns across episodes. Without *planning* (*A-Evolve/P*), the evolver fails to coordinate interdependent edits, leading to incoherent updates such as modifying tools without updating corresponding schemas. Finally, removing *verification* (*A-Evolve/V*) results in defective artifacts (e.g., tools with syntax errors) being committed, polluting context and degrading established capabilities. Detailed examples are provided in Appendix C.2.

5.4 Analysis of Evolution Scaling

Scaling Evolution Compute. We first evaluate whether increasing evolution-time compute reliably improves capability. We vary the *evolution step* from 1 to 12 steps as a proxy for C_{evolve}, where each step corresponds to the evolver processing a batch of 10 episodes.

Table 1: Comparison of agentic evolution against baselines on AppWorld. We report TGC and APT (%) across solver backbones.

	Claude Haiku 4.5		Claude Sonnet 4.5		Claude Sonnet 4		GPT 5		Gemini 3 Flash	
Method	TGC	APT	TGC	APT	TGC	APT	TGC	APT	TGC	APT
Vanilla	32	51.16	86	94.94	42	79.42	80	94.15	56	80.45
APE	30	56.00	80	85.00	44	82.00	82	95.00	52	84.00
AWM	46	65.76	88	**97.32**	50	86.29	84	94.08	52	87.75
A-Evolve	**64**	**84.31**	**90**	96.47	**56**	**88.21**	**88**	**97.02**	**82**	**92.05**



(a) Evolution step **(b) Evolver size**

Figure 4: Analysis of evolution-scaling hypotheses: we vary evolution step and evolver model size.

We use Claude Haiku 4.5 as the solver and Claude Sonnet 4.5 as the evolver. Other settings follow Sec. 5.2. Fig. 4(a) compares A-Evolve with vanilla and AWM baselines. Two trends emerge. *(i) Sustained scaling.* A-Evolve improves monotonically as compute increases, achieving the best performance across all budgets, while AWM quickly plateaus. This supports the evolution-scaling hypothesis that additional evolution compute translates into higher attainable performance, keeping P^* increasing with C_{evolve}. *(ii) High efficiency.* Even minimal evolution step (e.g., $0 \rightarrow 1$ step) yield substantial gains, indicating that structured evolution converts compute into capability more efficiently than heuristic baselines.

Impact of Evolver Size. We next examine scaling compute *per update* by increasing *evolver model size*. Fixing the solver as Claude Haiku 4.5, we vary the evolver backbone across Claude Haiku 4.5, Sonnet 4.5, and Opus 4.5; other settings follow Sec. 5.2. Results are shown in Fig. 4(b). *(i) Larger evolvers perform better.* Both TGC and APT increase monotonically with evolver size, mirroring the trends observed under sample scaling and confirming that evolver capacity is a key contributor to C_{evolve}. *(ii) Reduced optimization noise.* Qualitative analysis shows that smaller evolvers more often hallucinate root causes and propose brittle updates that fail verification. Larger evolvers more reliably diagnose failures and synthesize robust, generalizable artifacts, converting compute into capability more effectively. Additional analysis is provided in Appendix C.3.

6 Alternative Views

We now address three common alternative perspectives on deployment-time adaptation.

"Inference-time reasoning is sufficient." A common counter-argument suggests that explicit evolution is unnecessary if inference-time compute is scaled, allowing models to bridge environmental gaps by "thinking longer" [25]. While extended reasoning is effective for novel instances, it is inefficient for recurrent failures. Without evolution, the system repeatedly rediscovers the same solutions. Agentic evolution amortizes this cost by converting transient reasoning into persistent artifacts, such as verified tools, yielding stable and reusable capability.

"Why not rely on parametric plasticity?" Some advocate direct adaptation through online parameter updates, arguing that continual fine-tuning should replace artifact management [24]. We contend that unconstrained weight updates are opaque to auditing and prone to catastrophic forgetting. In contrast, agentic evolution operates over explicit, verifiable artifacts (e.g., code and tests), enabling modular updates, structured governance, and stronger safety guarantees than just gradient descent.

"Explicit evolution is too costly." Critics argue that maintaining an evolver agent is computationally wasteful compared to heuristics or passive memory. We view this as a trade-off between short-term efficiency and long-term scalability. While heuristics plateau, our results in Sec. 5.4 show that agentic evolution continues to improve with evolution-time compute, approaching the compute-optimal frontier. Allocating compute to evolution converts raw compute into durable capability for unbounded environments.

7 Conclusion and Future Directions

This paper argues that the primary bottleneck to deploying intelligent systems in the real world is not static model capacity, but the ability to *evolve*. As agents move beyond curated training distributions into open-ended environments, static optimization and non-agentic update rules inevitably saturate. We contend that agentic evolution provides a scalable alternative by elevating evolution from a fixed pipeline to an autonomous, governed decision process that converts deployment experience into persistent capability. We further propose the evolution-scaling hypothesis, which posits that adaptation capacity scales with evolution-time compute, introducing a new axis of scaling beyond training-time and inference-time computation.

Realizing the full potential of agentic evolution requires progress along three complementary directions.

Benchmarks. Future benchmarks should explicitly capture the train–deploy gap by placing agents in high-entropy, non-stationary environments where failures cannot be resolved through inference alone. Beyond task success, benchmarks should measure the durability and reuse of evolved artifacts to assess whether evolution-time compute produces lasting capability gains.

Frameworks. Our analysis suggests that the structure of the evolution framework determines how efficiently a system approaches the compute-optimal frontier P^*. While A-Evolve provides a concrete starting point, future frameworks should further improve diagnosis,

planning, and verification to better convert evolution-time compute into stable improvements.

Theory. A foundational theory of agentic evolution remains open. Promising directions include formalizing evolution as optimization over a combinatorial program space and establishing separation results showing that agentic evolution admits a higher attainable frontier than non-agentic heuristics. Bounding regret relative to idealized oracle fine-tuning would provide a principled basis for understanding long-horizon adaptation.

References

[1] Anthropic. 2025. Claude Haiku 4.5 System Card. https://assets.anthropic.com/m/99128ddd009bdcb/Claude-Haiku-4-5-System-Card.pdf

[2] Anthropic. 2025. Claude Sonnet 4.5 System Card. https://assets.anthropic.com/m/12f214efcc2f457a/original/Claude-Sonnet-4-5-System-Card.pdf

[3] Ali Behrouz, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni. 2025. It's All Connected: A Journey Through Test-Time Memorization, Attentional Bias, Retention, and Online Optimization. *arXiv preprint arXiv:2504.13173* (2025).

[4] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. 2024. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663* (2024).

[5] Sarfraz Brohi, Qurat-ul-ain Mastoi, N. Z. Jhanjhi, and Thulasyammal Ramiah Pillai. 2025. A Research Landscape of Agentic AI and Large Language Models: Applications, Challenges and Future Directions. *Algorithms* 18 (2025). doi:10.3390/a18080499

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, et al. 2025. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191* (2025).

[8] Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. 2025. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. *arXiv preprint arXiv:2512.10696* (2025).

[9] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335* (2024).

[10] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413* (2025).

[11] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. In *Proceedings of the 41st International Conference on Machine Learning*. 13481–13544.

[12] João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.* 46, 4 (2014). doi:10.1145/2523813

[13] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046* (2025).

[14] Google. 2025. Gemini 3 Flash Model Card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf

[15] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309* (2024).

[16] Jinwu Hu, Zitian Zhang, Guohao Chen, Xutao Wen, Chao Shuai, Wei Luo, Bin Xiao, Yuanqing Li, and Mingkui Tan. 2025. Test-Time Learning for Large Language Models. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=iCYbIaGKSR

[17] Jinwu Hu, Zhitian Zhang, Guohao Chen, Xutao Wen, Chao Shuai, Wei Luo, Bin Xiao, Yuanqing Li, and Mingkui Tan. 2025. Test-Time Learning for Large Language Models. *arXiv preprint arXiv:2505.20633* (2025).

[18] Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. 2025. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004* (2025).

[19] Jonas Hübotter, Sascha Bongni, Ido Hakimi, and Andreas Krause. 2025. Efficiently Learning at Test-Time: Active Fine-Tuning of LLMs. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=NS1G1Uhny3

[20] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907* (2023).

[21] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[22] Hanyu Lai, Xiao Liu, Junjie Gao, Jiale Cheng, Zehan Qi, Yifan Xu, Shuntian Yao, Dan Zhang, Jinhua Du, Zhenyu Hou, Xin Lv, Minlie Huang, Yuxiao Dong, and Jie Tang. 2025. A Survey of Post-Training Scaling in Large Language Models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

[23] Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, et al. 2025. SPIRAL: Self-Play on Zero-Sum Games Incentivizes Reasoning via Multi-Agent Multi-Turn Reinforcement Learning. *arXiv preprint arXiv:2506.24119* (2025).

[24] Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. 2025. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing* (2025).

[25] Aaron Jaech OpenAI, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[26] Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140* (2025).

[27] Zehua Pei, Hui-Ling Zhen, Shixiong Kai, Sinno Jialin Pan, Yunhe Wang, Mingxuan Yuan, and Bei Yu. 2025. SCOPE: Prompt Evolution for Enhancing Agent Effectiveness. *arXiv preprint arXiv:2512.15374* (2025).

[28] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. 2023. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. 3845–3864.

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[30] Rylan Schaeffer, Joshua Kazdan, John Hughes, Jordan Juravsky, Sara Price, Aengus Lynch, Erik Jones, Robert Kirk, Azalia Mirhoseini, and Sanmi Koyejo. 2025. How Do Large Language Monkeys Get Their Power (Laws)?. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=QqVZ28qems

[31] Yuchen Shi, Yuzheng Cai, Siqi Cai, Zihan Xu, Lichao Chen, Yulei Qin, Zhijian Zhou, Xiang Fei, Chaofan Qiu, Xiaoyu Tan, et al. 2025. Youtu-Agent: Scaling Agent Productivity with Automated Generation and Hybrid Policy Optimization. *arXiv preprint arXiv:2512.24615* (2025).

[32] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[33] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. 2025. OpenAI GPT-5 System Card. *arXiv preprint arXiv:2601.03267* (2025).

[34] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM Test-Time Compute Optimally Can be More Effective than Scaling Parameters for Reasoning. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=4FWAwZtd2n

[35] Liangcai Su, Zhen Zhang, Guangyu Li, Zhuo Chen, Chenxi Wang, Maojia Song, Xinyu Wang, Kuan Li, Jialong Wu, Xuanzhong Chen, et al. 2025. Scaling agents via continual pre-training. *arXiv preprint arXiv:2509.13310* (2025).

[36] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[37] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901* (2024).

[38] Yibo Wang, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. 2025. SPACE: Noise contrastive estimation stabilizes self-play fine-tuning for large language models. *arXiv preprint arXiv:2512.07175* (2025).

[39] Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. 2025. Evoagentx: An automated framework for evolving agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 643–655.

[40] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent workflow memory. *arXiv preprint arXiv:2409.07429* (2024).

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[42] Tianxin Wei, Ting-Wei Li, Zhining Liu, Xuying Ning, Ze Yang, Jiaru Zou, Zhichen Zeng, Ruizhong Qiu, Xiao Lin, Dongqi Fu, et al. 2026. Agentic Reasoning for Large Language Models. *arXiv preprint arXiv:2601.12538* (2026).

[43] Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, et al. 2025. Evo-Memory: Benchmarking LLM Agent Test-time Learning with Self-Evolving Memory. *arXiv preprint arXiv:2511.20857* (2025).
[44] Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. 2025. Toward Training Superintelligent Software Agents through Self-Play SWE-RL. *arXiv preprint arXiv:2512.18552* (2025).
[45] Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. 2025. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043* (2025).
[46] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110* (2025).
[47] Fangcong Yin, Xi Ye, and Greg Durrett. 2024. Lofit: Localized fine-tuning on llm representations. *Advances in Neural Information Processing Systems* 37 (2024), 9474–9506.
[48] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496* (2024).
[49] Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, et al. 2025. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395* (2025).
[50] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618* (2025).
[51] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations.*

A Related Work: Evolution Paradigms of LLMs

In this section, we review prior work through the lens of our evolution definition in Eq. 3 and the three orthogonal axes: *what* is updated (parametric weights π_θ vs. persistent artifact state π_S), *when* updates occur (offline training vs. deployment-time), and *how* the update rule F_{Evolve} is realized (fixed heuristics vs. an explicit agentic optimizer). The evolution paradigm is outlined in Tab. 2.

A.1 Offline Parametric Evolution

Self-play fine-tuning from synthetic interactions. A predominant paradigm for enhancing agent capability is to improve the parametric backbone π_θ *offline* by bootstrapping training signals from the model itself. Traditional approaches, such as SPIN [9], iteratively refine the policy by contrasting self-generated outputs against a target distribution. Recent advances extend this to multi-turn and multi-agent settings: SPIRAL [23] induces reasoning skills via self-play in structured interaction games, while SPACE [38] stabilizes the self-play objective to mitigate training instability. While these methods enlarge pre-deployment capability, they commit improvements solely to π_θ during training, leaving the agent static and unable to adapt to distribution shifts once deployed.

Zero-data curriculum generation. To circumvent the reliance on human-curated datasets, recent frameworks construct self-generated curricula within the offline loop. R-Zero [18] and Agent0 [45] instantiate co-evolving roles (e.g., a challenger and a solver) to synthesize frontier tasks and tool-use trajectories without external data. Similarly, in software engineering domains, SSR [44] trains agents by injecting and repairing bugs in sandboxed repositories. Although these methods effectively address the data bottleneck, they remain instances of *offline parametric evolution*: adaptation ceases the moment the model is deployed.

Agentic Continual Pre-training. Beyond fine-tuning, recent works propose injecting agentic capabilities [5, 42] even earlier in the pipeline. AgentFounder [35] introduces *Agentic Continual Pre-training (CPT)* as an intermediate stage, scaling agentic capabilities via massive offline data synthesis (e.g., First-order and Higher-order Action Synthesis) derived from knowledge bases and trajectories. While this approach significantly raises the baseline performance of the foundation model, it remains fundamentally an instance of *offline parametric evolution*: the model's adaptability is frozen once the pre-training concludes, leaving it vulnerable to novel environmental shifts during deployment.

A.2 Online Parametric Evolution

Test-time fine-tuning. Online parametric evolution attempts to update π_θ (or a subspace thereof) during deployment. Approaches like SIFT [19] explore active fine-tuning by selecting informative test-time examples for weight updates. To reduce computational overhead, methods such as LoFiT [47] localize updates to specific internal representations or heads. While enabling continuous adaptation, direct parametric modification introduces significant governance challenges, such as catastrophic forgetting [24], irreversibility, and instability, making it risky for long-horizon deployment.

Architectures for test-time memorization. A complementary direction blurs the boundary between weights and state by updating neural memory modules during streaming inference. Titans [4] introduces a neural long-term memory module for context memorization, while MIRAS [3] provides a framework for optimizing associative memory online. In our taxonomy, these methods still rely on *parametric* updates to a learnable module rather than explicit updates to interpretable artifacts, retaining the opacity of weight-based evolution.

A.3 Heuristic Non-Parametric Evolution

Experience memory with fixed retrieval. To avoid the risks of weight updates, many systems [26, 43, 46] modify a persistent artifact state π_S, typically a memory bank of traces or summaries, using fixed heuristics. ReasoningBank [26] distills strategies into a reusable memory bank via self-judgment. However, because the update rule F_{Evolve} is heuristic (e.g., "append and retrieve"), these systems often suffer from context saturation and retrieval noise as experience accumulates.

Prompt and context optimization. Another class of non-parametric methods [11, 28, 51] treats π_S as a set of optimizable prompts. GrIPS [28] and Promptbreeder [11] apply discrete search operators (e.g., mutation, crossover) to evolve prompts. More recently, Training-Free GRPO [7] performs policy optimization in the context space via token priors, and TextGrad [48] applies textual differentiation to optimize components. While effective, these methods typically employ a static, pre-specified search algorithm as F_{Evolve}, limiting the system's ability to strategically diagnose failures or plan complex structural updates.

A.4 Toward Agentic Evolution

Agent-driven context and prompt engineering. Several works operationalize prompt engineering as an agentic reasoning task. ACE [50] and SCOPE [27] treat context management as an online optimization problem, employing a modular loop (generate–reflect–curate) to evolve playbooks or guidelines. While these methods

Table 2: Evolution paradigms organized by *what* is updated, *when* updates occur, and *how* the update rule F_{Evolve} is instantiated.

Paradigm	What is Updated	When is Updated	How F_{Evolve} is Realized
Offline Parametric Evolution	π_θ	Offline (pre-deployment)	Training loop (e.g., self-play / distillation)
Online Parametric Evolution	π_θ	Online (deployment-time)	Online optimization (e.g., gradient descent)
Non-parametric Heuristic Evolution	$\pi_S\ (\mathcal{K})$	Online (deployment-time)	Fixed heuristic rules
Agentic Evolution (Ours)	$\pi_S\ (\mathcal{K}, \mathcal{T}, \mathcal{V})$	Online (deployment-time)	**Explicit evolver agent**

move beyond heuristic search by using an LLM to reason about updates, they are primarily restricted to the *textual context* modality, optimizing instructions rather than executable code or structural system logic.

Automated synthesis of tools and memory. Other approaches [31, 39, 49] focus on synthesizing specific functional components. Youtu-Agent [31] and AgentEvolver [49] introduce meta-agents capable of generating new tools or training curricula to expand agent capabilities. Similarly, A-Mem [46] targets the memory modality, employing an agentic controller to dynamically restructure and cross-reference memory artifacts rather than passively appending logs. These systems demonstrate the feasibility of agent-driven updates for specific artifacts. However, they typically operate as specialized pipelines for a single type of state (e.g., only tools or only memory) and often lack a unified mechanism for *governed diagnosis* and *verification* across diverse failure modes.

Positioning of our work. Existing works can be viewed as specific instances of using LLMs to optimize state. Our work generalizes these developments into the formal framework of **Agentic Evolution**. Unlike prior approaches that focus on optimizing a single modality (e.g., prompts *or* tools), we propose a holistic, budget-aware evolver capable of diagnosing diverse execution failures and proposing structural edits to a heterogeneous state π_S. Crucially, we introduce the principles of *goal orientation*, *autonomy*, and *compositionality* to strictly govern this process, ensuring that agent-driven updates are not just flexible, but reliable and scalable enough for deployment.

B Additional Details of Experimental Setup

B.1 Dataset Details

In this paper, we focus on AppWorld, a controllable environment and benchmark designed for interactive coding agents. It simulates a world of 9 day-to-day applications (e.g., Amazon, Spotify, Venmo, Gmail) and 2 helper apps (ApiDocs, Supervisor) through 457 APIs. The environment is populated with the digital activities of approximately 100 fictitious users, simulating realistic lives and relationships (e.g., family, roommates) to support complex interaction scenarios.

The AppWorld framework consists of two primary components:

- **AppWorld engine:** This is a high-quality execution environment. It provides a fully controllable and reproducible simulator where agents can operate apps via APIs without real-world consequences, such as spending actual money or spamming emails. The engine allows agents to execute rich Python code, including loops and conditionals, to interact with the environment iteratively.
- **AppWorld benchmark:** A suite of 750 diverse and challenging tasks derived from 250 task scenarios. These tasks require agents to navigate multiple apps (average 1.8 apps per task) and utilize up to 26 APIs (avg. 9.5) with dependencies between calls (i.e., outputs of calls used as inputs to subsequent calls). The dataset is split into "normal" (`test-normal`) and "challenge" (`test-challenge`) sets, where the challenge set requires agents to utilize APIs from apps seen only during test time.

B.2 Implementation Details of Agentic Evolution

In this subsection, we provide granular details on the architecture, artifact representation, and inference configuration of A-Evolve.

Architecture of the evolver agent. The Evolver is implemented as a sequential pipeline with four specialized components: Observer, Proposer, Updater, and Verifier. Unlike role-based prompting, each component is instantiated as a distinct module with dedicated logic and persistence.

- **Diagnoser:** Receives the full interaction trace $\text{Obs}_{1:t}$ from the Solver, including the user instruction, tool calls, stdout/stderr, and the final error. Its objective is *failure attribution*: distinguishing between stochastic environment noise and epistemic gaps in the agent's policy. We employ a cross-episode aggregation strategy over batched observations (window size $W = 10$) to perform root cause analysis, filtering out transient errors to focus on persistent capability deficits.
- **Planner:** Conditioned on the diagnosis, the planner is responsible for synthesizing a structural update δ. We constrain the planner's action space to a strict schema of operations $\mathcal{A}$ = {CreateTool, EvolveTool, AddKnowledge, EvolveKnowled
This structured output forces the model to map abstract failure modes into concrete, semantically meaningful artifact definitions rather than unstructured text, significantly reducing the search space for improvements.
- **Updater:** The updater acts as the deterministic execution engine. It holds write access to the *Artifact Registry* (a version-controlled file system) and translates the Planner's high-level

intent into atomic file operations. It enforces strict separation of concerns, ensuring that tools (Python modules), skills (procedural knowledge), and facts (JSON knowledge base) are stored in their respective canonical formats.
- **Verifier:** Responsible for governance. In our case, it validates syntactic correctness of generated code (Python AST parsing for tools), schema compliance (YAML frontmatter for skills), and runtime safety (isolated execution in `ToolWorkspace`). Only proposals passing validation are committed to the workspace and reflected in the agent's system prompt via on-policy state injection.

Hyperparameters and Model Configuration. Unless otherwise specified, we use Claude Sonnet 4.5 [2] as the evolver backbone, and evaluate solvers ranging from Claude Haiku 4.5 [1] and Claude Sonnet 4/4.5 to GPT-5 [33] and Gemini 3 Flash [14]. For generation hyperparameters, we use a temperature of 0.7 for both the evolver and solver. The maximum output token limits are set to 8,192 for the evolver and 4,096 for the solver.

B.3 Evaluation Metrics

To ensure the robustness of our evaluation, following prior work [8, 37], we report two complementary metrics based on the concept of a **task score**.

For each task i, we define its score $s_i \in [0, 1]$ as the fraction of passed unit tests:

$$s_i = \frac{\text{passed_tests}_i}{\text{total_tests}_i}. \tag{12}$$

Based on this score, we define:

- **Average Passed Tests (APT):** This metric captures the granular, incremental progress of the agent. It is simply the average score across all evaluation tasks:

$$\text{APT} = \frac{1}{N} \sum_{i=1}^{N} s_i. \tag{13}$$

  APT is particularly valuable for measuring evolution, as it reflects improvements in capability (e.g., s_i increasing from 0.2 to 0.8) even when the full task is not yet perfectly solved.
- **Task Goal Completion (TGC):** This metric represents the strict success rate. A task is considered successfully completed if and only if it achieves a perfect score (i.e., all verification conditions are met). TGC is defined as:

$$\text{TGC} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(s_i = 1), \tag{14}$$

  where N is the total number of tasks and $\mathbb{I}(\cdot)$ is the indicator function.

C More details of Experimental Results

C.1 Case Studies of Effectiveness of Agentic Evolution in Sec. 5.2

To provide qualitative evidence for the effectiveness trends in Tab. 1, we analyze representative failure and success patterns from A-Evolve and AWM. Both methods use the same training and test sets (50 tasks each) with Claude Haiku 4.5 as the solver.

Observation 1: failure recovery mechanisms. A key distinction is how each method handles errors mid-trajectory:

- **AWM (Passive):** When the solver encounters an API error (e.g., `401 Unauthorized`), it has no mechanism to update its workflow in-place. The error propagates, and the solver must re-explore from scratch—often repeating the same mistake. In task *"What is the title of the most-liked song in my Spotify playlists"*, the agent cycles through 30 identical failed attempts without adaptation.
- **A-Evolve (Active):** After observing repeated `401` errors across a batch, the evolver creates a `manage_auth_token` tool that automatically caches and injects access tokens. The diagnosis explicitly identifies: "*The trajectory shows repeated 401 errors because the agent forgets to pass 'access_token' after login.*" This structural insight is encoded as a verified tool, enabling one-shot authentication in subsequent tasks.

Observation 2: context saturation. We compare the learned artifacts after 50 training tasks:

- AWM produces a single monolithic workflow file that grows linearly with training. This file contains concatenated trajectory fragments with task-specific details (e.g., hardcoded credentials, specific note IDs) that do not generalize. When retrieved, these fragments often mislead the solver with irrelevant context.
- A-Evolve produces 16 modular skills (e.g., `systematic_api_exploration`, `detect_execution_stall`), 4 verified tools (e.g., `authenticate_app`), and 4 knowledge entries (e.g., `appworld_authentication`). Each artifact is *semantically indexed* by trigger conditions and *verified* before commitment.

Observation 3: trajectory comparison. For task *"Reply to Christopher with movie recommendations from SimpleNote"*, we compare solve-time behavior:

AWM Agent (Score: 0.88, 29 steps):

- Steps 1-6: Explores API documentation (redundant with workflow)
- Steps 7-11: Fails 401 on SimpleNote, re-explores
- Steps 12-23: Successful login and note retrieval after trial-and-error
- Steps 24-28: Finds contact, sends message
- Step 29: Completes task

A-Evolved Agent (Score: 1.0, 8 steps):

- Step 1: Invokes `analyze_task_requirements()` to parse goal
- Step 2: Retrieves `authentication_tool_usage_pattern` skill
- Steps 3-4: Authenticates SimpleNote and Phone with cached token procedure
- Steps 5-6: Retrieves note, extracts movies
- Step 7: Sends message to Christopher
- Step 8: Completes task

The A-Evolved agent is 3.6× more efficient (8 vs. 29 steps) because it *amortizes* authentication and goal-parsing into reusable procedures that execute deterministically, rather than re-exploring on each task.

Takeaway. The case studies reveal a fundamental asymmetry: AWM captures *what* the agent did (raw trajectories), while A-Evolve captures *why* failures occurred and *how* to prevent them (verified tools, diagnostic knowledge). This explains the consistent TGC and APT gaps in Tab. 1: agentic evolution converts deployment feedback

into persistent, governed capabilities that static workflow retrieval cannot match.

C.2 Analysis of Feasibility of Agentic Evolution in Sec. 5.3

Qualitative analysis. We analyze evolution traces and observe distinct failure modes for each ablation.

- *Diagnosis* is critical for depth: without it (*A-Evolve/D*), the agent acts **blindly**, forced to infer updates directly from **raw trajectories** without root-cause reasoning. Consequently, it resorts to **superficial patching** (e.g., masking a crash with 'try-except' rather than fixing the incorrect parser logic).
- *Analysis Tools* enable **pattern discovery**: without them (*A-Evolve/A*), the agent relies on pure LLM inference over single trajectories, capturing only **superficial symptoms** (e.g., fixing a one-off error message). In contrast, analysis tools allow the evolver to aggregate **underlying statistical patterns** across episodes (e.g., clustering recurring failure modes), thereby distinguishing systematic deficits from transient noise.
- *Planning* ensures coherence: without it (*A-Evolve/P*), the agent lacks the mechanism to manage inter-dependencies between artifacts, leading to disjointed updates (e.g., modifying a tool's signature in $\mathcal{T}$ without updating its usage schema in $\mathcal{K}$, causing subsequent execution failures).
- Finally, *Verification* acts as the stabilizer: without it (*A-Evolve/V*), the system blindly commits **defective artifacts** (e.g., tools with **syntax errors**). This **pollutes the context**, causing the solver to waste inference budget on broken calls, and introduces regressions that degrade established capabilities.

Case Studies. We provide several representative case studies to better support our analysis:

Case 1: verification prevents defective artifacts (A-Evolve/V). When considering task *"What is the title of the most-played song in my Spotify album library"*, the evolver created a `discover_apis` tool without verification. On the solver's first invocation:

> *discover_apis: 'success': False, 'error': "module 'appworld' has no attribute 'list_apis'"*

The solver wasted a step before falling back to manual exploration. This pattern repeated: **2 broken tools committed** across evolution, causing runtime errors that degraded solver efficiency by ∼15% compared to full A-Evolve.

Case 2: planning enables implementable fixes (A-Evolve/P). The planning stage translates high-level diagnostic insights into *coordinated action sequences*. Consider a task in full A-Evolve: the diagnosis identifies "agent wastes steps on 401 authentication errors." The **Plan** generates 4 interdependent actions:

- `CREATE_TOOL`: `discover_api_spec` (API discovery)
- `CREATE_TOOL`: `manage_auth_token` (token storage)
- `ADD_SKILL`: `systematic_api_exploration` (usage pattern)
- `ADD_SKILL`: `authentication_workflow` (login procedure)

Crucially, item (3) relies on item (1): the skill `systematic_api_exploration` explicitly instructs the solver to invoke the newly created `discover_api_spec` tool. **Without planning, this dependency graph is lost.** In the A-Evolve/P ablation, although the diagnosis produced similar insights, the immediate generation step failed to sequence the creation process. It attempted to define the skill before the tool existed or hallucinated a tool signature, resulting in validation failures. Consequently, while A-Evolve/P accumulated some loose textual knowledge, it successfully synthesized 0 executable tools and 0 structured skills, failing to implement coherent, multi-artifact solutions.

Case 3: diagnosis enables root-cause analysis (A-Evolve/D). Without diagnosis, the evolver observed the same errors but lacked structured analysis. With full A-Evolve, the diagnosis identified:

> *"Agent uses wrong parameter names (e.g., 'email' instead of 'username') causing 422 errors. The agent doesn't consistently check API documentation before calling APIs."*

This insight led to the `systematic_api_exploration` skill, reducing 422 errors from 8 per trajectory to <1.

Case 4: analysis tools enable pattern discovery (A-Evolve/A). Without analysis tools, the evolver relied on LLM inference over single trajectories. With analysis tools, `grep_observations` revealed that 401 errors occurred in **18/20 training tasks**, enabling the evolver to prioritize `authentication_workflow` over superficial one-off fixes.

Takeaway. Reliable agentic evolution is an *irreducible loop*: diagnosis and analysis enable targeted updates, planning enables implementable multi-step fixes, and verification contributes most for stable accumulation over long horizons.

C.3 Case Studies of Impact of Evolver Size

To provide qualitative evidence for the scaling trends in Fig. 4(b), we analyze how evolvers of different capacities respond to identical failure patterns. We run A-Evolve on the same 50-task training set with identical configurations, varying only the evolver backbone: Claude Sonnet 4.5 vs. Claude Haiku 4.5. The solver remains fixed as Claude Haiku 4.5 in both conditions.

Observation 1: diagnosis depth and artifact quality. We observe stark qualitative differences in how evolvers diagnose failures and synthesize artifacts. Consider a recurring failure pattern: the solver successfully authenticates and retrieves playlists but fails to find the "most-liked song" due to incorrect interpretation of the `like_count` field versus `popularity` score.

- **Sonnet 4.5 Evolver:** The diagnosis identifies the *structural* root cause: "*The agent's technical execution is flawless (40% score suggests partial credit for correct workflow), but it's missing the actual task requirements. The trajectory shows it reads 'What is the title of...' but does not extract the semantic meaning.*" The evolver proposes a multi-artifact update:
  - `EVOLVE_TOOL`: Updates `analyze_task_requirements` to parse task descriptions and extract expected output formats
  - `ADD_SKILL`: Creates `pre_submission_verification` to validate answers against task requirements before calling `task_complete()`
  - `ADD_SKILL`: Creates `detect_execution_stall` to recognize when the agent is not progressing toward the goal
- **Haiku 4.5 Evolver:** The diagnosis is shallower: "*The agent failed because it did not call the correct API. Fix by adding a tactical patch.*" The evolver proposes:
  (1) `EVOLVE_TOOL`: Attempts to create `api_interaction_engine` to wrap all API calls

In addition, the tool fails verification after 3 repair attempts due to missing parameter validation. The structural tool is rejected, leaving only shallow behavioral textual patches.

Observation 2: verification failure rates. We quantify the difference in artifact quality by examining verification outcomes:

Table 3: Verification outcomes across 50-task training runs. Larger evolvers produce artifacts that pass verification at significantly higher rates.

Evolver	Proposals	Tools Committed	Verification Failures
Sonnet 4.5	21	4	2 (repaired successfully)
Haiku 4.5	25	1	8 (3+ repair attempts each)

The Haiku evolver emits 4× more verification failures despite generating fewer total proposals. Qualitative inspection reveals that Haiku-generated tools often have subtle bugs: missing required parameters, incorrect function signatures, or hallucinated API endpoints.

Observation 3: cascading effects on solver behavior. The quality gap compounds over training. With Sonnet-evolved artifacts, the solver learns to:

- Call `analyze_task_requirements()` before execution to parse the goal
- Invoke evolved authentication tools that correctly manage access tokens
- Apply `pre_submission_verification` to catch answer-format mismatches

However, with Haiku-evolved artifacts, the solver exhibits *regression*: it attempts to use the partially-broken `api_interaction_engine` tool, which fails silently or returns errors. The solver then falls back, losing any efficiency gains from tool use.

Observation 4: trajectory comparison. For task "venmo payment request based on SimpleNote records", we compare solver behavior:

- **Sonnet-evolved solver:** Invokes `analyze_task_requirements()` on first turn, retrieves relevant skills, and completes the task in 8 steps with no errors.
- **Haiku-evolved solver:** Fail to call tools, it then attempts 14 exploratory API calls before arriving at the correct workflow. Despite executing the business logic correctly (e.g., creating 4 payment requests), it fails to call `task_complete()`, a failure mode that triggered repeated but unsuccessful evolution attempts.

Takeaway. Larger evolvers produce higher-quality diagnoses that identify *structural* capability gaps rather than surface-level symptoms. This translates into more robust artifact synthesis: tools that pass verification, skills that encode generalizable procedures, and knowledge that captures system invariants. The verification stage acts as a quality filter, but cannot rescue fundamentally flawed proposals—hence the importance of evolver capacity in converting C_{evolve} into durable capability.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009