EVALUATION-FREE RECURSIVE SELF-IMPROVEMENT FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent self-improving coding agents have achieved strong benchmark performance by iteratively modifying their own executable implementations. A common pattern uses downstream task-derived evidence to decide which candidates or lineages continue evolving, which we term *evaluate-and-select*. Obtaining this signal requires repeatedly executing evolved agents on downstream tasks, so broader exploration incurs increasing evaluation cost, while repeated optimization against a fixed evaluation distribution may encourage benchmark-specific adaptation. We ask a more fundamental question: *can coding agents self-improve without downstream task evaluation inside the evolution loop?* We introduce **E**xp**l**ore-then-**I**n**t**egrat**e** (ELITE), which integrates implementation mechanisms rather than selecting a single exploratory child to continue the lineage. At each generation, ELITE proposes diverse focused improvements, implements them independently from the same parent, and inspects the resulting programs to identify complementary, redundant, and conflicting mechanisms before reimplementing a compatible subset as one coherent descendant. No downstream benchmark task, task-solving trajectory, or candidate performance score is used to guide evolution; only deterministic checks of structural and operational validity are used inside the loop. On *SWE-bench Pro*, ELITE improves the initial agent from 60.0% to 85.6%, compared with 86.7% for the Darwin Gödel Machine (DGM) and Huxley-Gödel Machine (HGM), while using zero candidate task evaluations and reducing observed wall-clock evolution time by 8.8x and 3.2x, respectively, under the evaluated configurations. Without further adaptation, ELITE averages 73.3% on *SWE-bench Verified* and 69.3% on *Polyglot*. Together, these results demonstrate the feasibility of coding-agent self-improvement without downstream candidate task evaluation in our setting.

1 INTRODUCTION

The idea of AI systems improving their capabilities has a long history, from early discussions of machines designing more capable successors (Good, 1966) to self-referential learning and the Gödel machine (Schmidhuber, 1987; 2003). Recent work continues to pursue this goal through modern agentic systems (Fang et al., 2025; Gao et al., 2025; Ren et al., 2026). These systems usually improve either the underlying large language model (LLM) or its scaffold, where the latter comprises prompts (Fernando et al., 2023; Yuksekgonul et al., 2024), memory (Xu et al., 2026b; Zhang et al., 2026c), tools (Cai et al., 2024; Qiu et al., 2025), and control logic (Zelikman et al., 2023; Yin et al., 2025) that govern how the model is used. Coding agents provide a practical setting for studying self-improvement because they can access and modify their own source code to recursively generate better variants of themselves. Recent systems have improved coding performance by repeatedly modifying this software while keeping the LLM fixed (Hu et al., 2025; Zhang et al., 2025; Robeyns et al., 2025; Zhang et al., 2026a; Wang et al., 2026a). Because each generation becomes the starting point for subsequent self-improvement, a central design question is not only how to generate self-modifications, but also which modifications should be carried forward across generations.

Existing approaches answer this question in different ways, but most rely on a common principle that we call *evaluate-and-select* to decide which candidates or lineages continue evolving. Although their search and selection mechanisms differ, downstream task-derived evidence is used to determine which candidates, groups, or lineages receive subsequent evolutionary effort. DGM uses benchmark performance to guide archive expansion or parent selection (Zhang et al., 2026a); HGM estimates

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

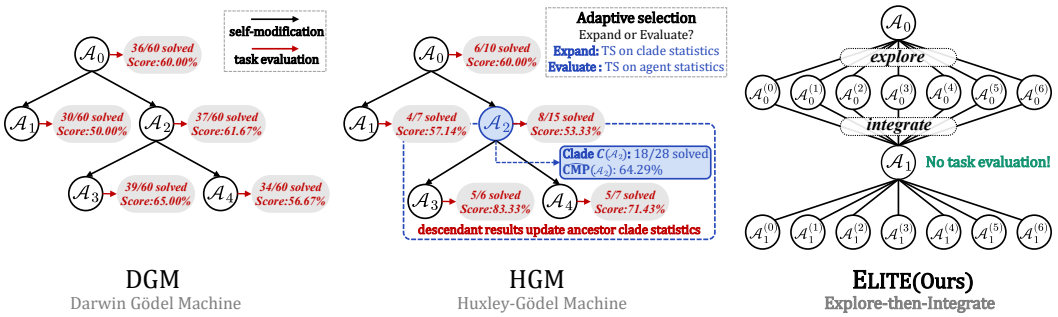


Figure 1: DGM and HGM exemplify *evaluate-and-select* strategies, using task-derived evaluation to guide evolution. ELITE instead explores multiple same-parent implementations and integrates compatible mechanisms into a descendant, without using downstream task outcomes to guide evolution.

the long-term productivity of an entire descendant clade from task outcomes (Wang et al., 2026a); other methods extend this paradigm through group-level search and task-based novelty (Weng et al., 2026), cross-lineage information sharing (Liu et al., 2026), learned ranking signals (Fu et al., 2026), or evolving evaluation criteria (Iacob et al., 2026). This evaluate-and-select paradigm introduces two limitations. First, obtaining reliable downstream feedback requires repeatedly executing evolved agents on target tasks, making broader exploration increasingly expensive (Xu et al., 2026a; Fu et al., 2026; Wang et al., 2026b; Zeng et al., 2026). Second, candidate-level selection compresses each exploratory implementation into a selection decision, potentially discarding useful mechanisms contained in otherwise unselected candidates (Weng et al., 2026). Repeated optimization against a fixed evaluation distribution may further encourage benchmark-specific adaptation rather than transferable improvements (Kapoor et al., 2024; Xia et al., 2026). We therefore ask: can coding agents decide which changes to carry forward without downstream candidate task evaluation?

Coding agents offer an alternative source of evidence that is usually overlooked by evaluate-and-select methods; their self-modifications are executable programs whose implementation mechanisms can be directly inspected. Rather than treating each exploratory candidate as an indivisible unit and reducing it to a task-derived selection signal, we can inspect what the candidate actually changed and reason about which mechanisms are complementary, redundant, or conflicting. This shifts the inheritance decision from the candidate level to the mechanism level. Under this view, exploration need not end by selecting a single winner. Multiple candidates can instead serve as proposals of distinct implementation mechanisms, from which compatible mechanisms are integrated into the next descendant. This suggests a different route to recursive self-improvement: explore diverse modifications, inspect their implementations, and integrate compatible mechanisms without using downstream task performance to decide what is inherited.

We introduce ELITE, an Explore-then-Integrate approach to self-improvement. ELITE first generates a portfolio of focused and diverse improvement hypotheses grounded in concrete limitations of the current agent's runtime source. Each hypothesis is implemented independently from the same parent, producing multiple executable alternatives. ELITE then inspects all valid alternatives, identifies mechanisms that are complementary, redundant, conflicting, incomplete, or unnecessarily complex, and integrates compatible mechanisms into one coherent descendant. The integrated descendant is accepted only if deterministic structural and operational checks pass. No downstream benchmark task, task-solving trajectory, or candidate performance score is used to guide this process. Figure 1 illustrates the difference between ELITE and existing evaluate-and-select approaches. On the 60-task subset of SWE-bench Pro, ELITE improves the initial agent from 60.0% to 85.6%, compared with 86.7% for DGM and HGM, while using zero candidate task evaluations and reducing observed evolution time by 8.8× and 3.2×, respectively, under our evaluated configurations. Without further adaptation, the evolved agent achieves 73.3% on *SWE-bench Verified* and 69.3% on *Polyglot*. Ablations further show that successful task-fitness-free evolution still requires constraints: removing deterministic validity filtering causes performance to collapse. Post-hoc analyses further examine how integration exploits mechanisms across exploratory children and characterize the resulting evolution dynamics.

Our contributions are threefold. **(1)** We observe that coding agents' self-modifications are inspectable programs, so which changes to carry forward can be decided from implementation-level evidence rather than downstream task outcomes; we formulate this as the problem of self-improvement without candidate task evaluation. **(2)** We introduce ELITE, which replaces candidate-level selection with same-parent executable exploration and compatibility-aware mechanism-level integration, while using deterministic checks only for operational validity. **(3)** We provide empirical evidence that coding-agent self-improvement without downstream candidate task evaluation is feasible in our setting, and analyze its evolution dynamics, transfer behavior, and key design components.

2 RELATED WORK

Self-improvement in coding agents. Self-improving agents can modify either the underlying model or the agent scaffold, including prompts, memory, tools, workflows, and control logic (Gao et al., 2025; Fang et al., 2025; Ren et al., 2026). Software-engineering agents provide infrastructure for interacting with repositories, development tools, and execution environments (Qian et al., 2024; Yang et al., 2024; Wang et al., 2025; Hong et al., 2024; Xia et al., 2025). Automated design methods such as GPTSwarm, ADAS, and AFlow optimize prompts, graph structures, or code-represented workflows (Zhuge et al., 2024; Hu et al., 2025; Zhang et al., 2025), while self-referential approaches such as STOP, Gödel Agent, and SICA modify their own scaffolds or implementations (Zelikman et al., 2023; Yin et al., 2025; Robeyns et al., 2025). Coding agents thus provide a concrete setting for recursive self-improvement because their executable implementations can be directly modified and used as the basis for subsequent improvements.

Research on self-improvement has a long history, including self-referential learning and self-modifying policies (Schmidhuber, 1987; Schmidhuber et al., 1997), as well as the Gödel Machine's formulation of self-rewriting under provable utility improvement (Schmidhuber, 2003). Within this broader area, evolutionary search offers one approach to program improvement. Relevant historical foundations include early evolutionary computation (Barricelli et al., 1954; Holland, 1975), genetic programming (Smith, 1980; Cramer, 1985; Dickmanns et al., 1987), and meta-evolution of the improvement procedure itself (Schmidhuber, 1987). Recent LLM-based systems revisit these ideas through executable self-modification. DGM uses benchmark performance to guide archive expansion (Zhang et al., 2026a), while HGM estimates lineage potential from descendant outcomes (Wang et al., 2026a). Hyperagents combines task solving and meta-level improvement in one editable program (Zhang et al., 2026b). GEA and MGM share experience or reuse trajectories and outcomes across groups and lineages (Weng et al., 2026; Liu et al., 2026); SIFT reduces evaluation cost through LLM-based ranking (Fu et al., 2026), and RQGM co-evolves evaluation criteria (Iacob et al., 2026). Despite their different search and evaluation strategies, these systems retain downstream task-derived evidence to guide subsequent evolution. ELITE instead studies whether implementation-level inspection and mechanism integration can construct the next descendant without downstream candidate task evaluation.

Implementation-level reasoning for code changes. Recent work in software engineering explores implementation-level reasoning without executing candidate code. ICE-Score and CodeJudge assess generated code, while RePaCA reasons over buggy and patched implementations to assess patch correctness (Zhuo, 2024; Tong & Zhang, 2024; Fuster-Pena et al., 2026). LLM critics assess repository-level changes, sometimes using reference information such as gold test patches (Yadavally et al., 2025). Agentic Rubrics scores patches against repository-specific criteria (Raghavendra et al., 2026), while SWE-RM learns reward models for fine-grained feedback without unit-test execution (Shum et al., 2026). ASPIRE uses multiple agents to simulate runtime behavior and reason about correctness without code execution (Ramanan et al., 2024). PatchFusion combines edit-level evidence across repair candidates without test outcomes at decision time (Yang et al., 2026). These works address feedback, verification, or patch combination for individual tasks, rather than how an agent's own implementation should evolve across generations. ELITE draws on the inspectability of code, using implementation-level reasoning and mechanism integration to construct the next agent without running candidate agents on downstream tasks or using downstream task outcomes to guide evolution.

3 PRELIMINARIES

3.1 FORMALIZING SELF-IMPROVING AGENTS

Following prior work (Ren et al., 2026), we formalize an agent configuration as $\mathcal{A} = (\theta, \Sigma)$, where θ denotes the parameters of the underlying LLM and Σ denotes the agent scaffold, including prompts, memory, tools, and control logic. Self-improvement proceeds over generations $t = 0, 1, \ldots, T-1$, with updates abstracted as $\mathcal{A}_{t+1} = \texttt{IMPROVE}(\mathcal{A}_{0:t}; S_t)$, where S_t denotes update-relevant information such as task outcomes, execution trajectories, critiques, verification signals, or proposed modifications. Each updated agent persists as a basis for subsequent self-modification. We focus on coding agents with a fixed underlying LLM, so $\mathcal{A}_t = (\theta, \Sigma_t)$ throughout evolution. In our setting, Σ_t is implemented as a source-code repository (Appendix A.2), allowing the scaffold and its modifications to be inspected before downstream evaluation. Such inspection does not establish downstream capability, but can provide useful implementation-level information (Yadavally et al., 2025; Raghavendra et al., 2026; Shum et al., 2026).

3.2 EVALUATE-AND-SELECT SELF-IMPROVEMENT

Self-improving coding agents organize search differently, but many use downstream task outcomes to guide evolution. We call this pattern *evaluate-and-select*. Let $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$ be a downstream evaluation set, and let $R(\mathcal{A}, \tau_i)$ denote the outcome of agent $\mathcal{A}$ on task τ_i. Under evaluate-and-select, these outcomes, or estimates obtained from partial evaluation, enter S_t and influence which agents, groups, or lineages are retained, expanded, or allocated further evaluation. The selection rule may be deterministic or stochastic, may operate over retained historical agents, and may combine task outcomes with lineage statistics, novelty, or resource costs; it need not select a single highest-scoring winner. This formulation encompasses a range of existing evaluation-driven strategies for selecting or allocating further effort across agents, groups, and lineages (Zhang et al., 2026a; Wang et al., 2026a; Weng et al., 2026; Liu et al., 2026; Fu et al., 2026; Iacob et al., 2026).

Obtaining this evidence requires task execution: evaluating M candidates on N tasks costs MN rollouts, while adaptive allocation changes the number of evaluations assigned to each candidate but does not eliminate this cost (Fu et al., 2026). Repeated optimization against a fixed evaluation distribution may further encourage benchmark-specific adaptation (Kapoor et al., 2024). Separately, agent-level performance scores do not by themselves specify how complementary implementation mechanisms should be combined. These considerations motivate our question: *can coding agents decide which changes to carry forward without downstream candidate task evaluation?*

4 ELITE: EXPLORE-THEN-INTEGRATE SELF-IMPROVEMENT

Under the formalization in Section 3.1, the two paradigms differ in the information used to update the agent. Evaluate-and-select uses downstream task-derived evidence in S_t to guide which agents or lineages receive further evolutionary effort. ELITE instead uses exploratory implementations and their deterministic validity results; no downstream task outcome enters S_t. Rather than selecting an entire child based on estimated performance, ELITE treats implementations as evidence and carries forward compatible mechanisms, shifting the unit of evolutionary decision from whole candidates to implemented mechanisms.

At each generation, ELITE follows two stages. **Explore** generates a portfolio of focused improvement hypotheses grounded in limitations of the current agent, each of which is independently implemented from the same parent to produce an executable alternative. **Integrate** then reasons over the mechanisms realized in these alternatives and reimplements a compatible subset into a coherent descendant. Deterministic gates are used only to ensure structural and operational validity; no downstream benchmark task, task-solving trajectory, or candidate performance score is used to guide the next-agent decision.

4.1 LIMITATION-DRIVEN SAME-PARENT EXPLORATION

Limitation-driven hypothesis generation. At generation t, ELITE examines the implementation of the current agent $\mathcal{A}_t$ together with a compact evolution history m_t, which records prior self-

modification instructions and their accompanying code-based rationales. Rather than applying a fixed set of mutation categories, a proposal operator $\mathcal{P}$ jointly generates K focused improvement hypotheses,

$$\mathcal{H}_t = \mathcal{P}(\mathcal{A}_t, m_t; K) = \{h_t^{(1)}, \ldots, h_t^{(K)}\}. \tag{1}$$

Each hypothesis specifies the capability to improve, a concrete limitation in the current implementation, a focused mechanism for addressing it, and the conditions under which the implementation is complete (Appendix A.3.1).

Same-parent executable exploration. Each hypothesis is implemented independently against the same parent $\mathcal{A}_t$. An implementation operator $\mathcal{E}$ produces a patch $\Delta_t^{(k)}$, which is applied to $\mathcal{A}_t$ to obtain an executable exploratory child:

$$\Delta_t^{(k)} = \mathcal{E}(\mathcal{A}_t, h_t^{(k)}), \qquad \mathcal{A}_t^{(k)} = \mathrm{Apply}(\mathcal{A}_t, \Delta_t^{(k)}), \qquad k = 1, \ldots, K. \tag{2}$$

Because all exploratory children are independently instantiated from the same parent, their modifications provide directly comparable implementation-level alternatives rather than sequentially accumulated changes. This common reference point also makes it possible to reason about mechanisms across children before constructing the next descendant.

Deterministic validity filtering. Before integration, ELITE applies a lightweight deterministic gate $G_{\mathrm{exp}}(\mathcal{A}) \in \{0, 1\}$ to exclude invalid program-level modifications. The gate checks structural and operational validity, such as executability, deployability, required agent interfaces, and permitted modification scope. The valid alternatives form

$$\mathcal{V}_t = \left\{ \mathcal{A}_t^{(k)} \mid G_{\mathrm{exp}}(\mathcal{A}_t^{(k)}) = 1 \right\}. \tag{3}$$

Importantly, G_{exp} is an admissibility check rather than a measure of downstream capability: it uses no benchmark task outcomes or candidate performance scores.

From exploration to integration. Exploration therefore produces a set of independently implemented, operationally valid alternatives rather than a ranked population. Since no downstream task score identifies a winning child in $\mathcal{V}_t$, the remaining decision is not which child should continue as a whole, but which mechanisms implemented across these children can be carried forward together. Section 4.2 addresses this decision through compatibility-aware mechanism integration.

4.2 COMPATIBILITY-AWARE MECHANISM INTEGRATION

From children to mechanisms. Given the valid alternatives $\mathcal{V}_t$, ELITE shifts the decision unit from whole children to their implemented mechanisms. Rather than assigning each child a scalar score or predicting which child would perform best, the integrator treats the implementations themselves as evidence. Mechanisms are identified from the intended improvement $h_t^{(k)}$ together with its realized implementation, allowing a mechanism to be considered independently of the child that contains it and jointly with compatible mechanisms realized by other children. For each valid child in $\mathcal{V}_t$, the integrator receives its hypothesis $h_t^{(k)}$, a summary of the realized patch $\Delta_t^{(k)}$, and the gate results; the full patch remains available for inspection. The integrator first reviews this evidence across all valid children and then inspects the full implementations of a small set of relevant alternatives. The underlying LLM performs integration through implementation-level reasoning guided by the structured prompt in Appendix A.3.3, without a separate compatibility model or task-performance predictor.

Compatibility-aware mechanism reasoning. Using this implementation-level evidence, the integrator reasons about candidate mechanisms along five dimensions: *complementary*, when they can operate together; *redundant*, when they substantially overlap; *conflicting*, when they make incompatible assumptions about interfaces, state, or control flow; *overly complex*, when they introduce unnecessary machinery; and *incomplete*, when the intended mechanism is only partially realized. These judgments determine which mechanisms can be carried forward together, without reducing each child to a single predicted-quality score. Based on these judgments, an integration operator $\mathcal{I}$ constructs an integrated patch against the original parent:

$$\Delta_t^{\mathrm{int}} = \mathcal{I}(\mathcal{A}_t, \mathcal{V}_t), \qquad \mathcal{A}_t^{\mathrm{int}} = \mathrm{Apply}(\mathcal{A}_t, \Delta_t^{\mathrm{int}}). \tag{4}$$

Importantly, integration operates at the mechanism level rather than by mechanically concatenating or sequentially applying exploratory patches. The selected mechanisms are reimplemented as a coherent modification of the common parent $\mathcal{A}_t$, allowing shared interfaces, dependencies, and control flow to be reconciled when mechanisms interact.

Deterministic descendant acceptance. The integrated descendant is accepted only if it passes a stricter deterministic gate $G_{\mathrm{acc}}(\mathcal{A}) \in \{0, 1\}$, which verifies structural and operational validity. The next generation is therefore

$$\mathcal{A}_{t+1} = \begin{cases} \mathcal{A}_t^{\mathrm{int}}, & G_{\mathrm{acc}}(\mathcal{A}_t^{\mathrm{int}}) = 1, \\ \mathcal{A}_t, & \text{otherwise.} \end{cases} \tag{5}$$

As with G_{exp}, this gate verifies structural and operational validity rather than downstream capability. No downstream benchmark task, task-solving trajectory, or candidate performance score is used to determine which mechanisms are integrated or whether the resulting descendant is accepted as the next agent.

Downstream evaluation cost. Unlike the direct evaluate-and-select setting in Section 3.2, ELITE requires no downstream candidate-task rollouts for next-agent decisions. Its cost instead comes from generating executable alternatives, inspecting their implementations, and constructing the integrated descendant; this decision cost is independent of the size of a downstream evaluation set. The complete procedure is summarized in Algorithm 1 in Appendix A.6.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on *SWE-bench Pro*, *SWE-bench Verified* (Deng et al., 2025; Jimenez et al., 2024), and *Polyglot* (Paul Gauthier, 2024). We use a fixed 60-task subset of SWE-bench Pro for the evolution comparison, while SWE-bench Verified and Polyglot serve as transfer benchmarks. After evolution, the resulting agents are frozen and evaluated on the same Pro subset and the two transfer subsets without further adaptation. Detailed benchmark and task-subset information is provided in Appendix A.4.

Compared methods and evolution protocol. We compare ELITE with DGM (Zhang et al., 2026a) and HGM (Wang et al., 2026a) using `DeepSeek-V4-Flash` and `Qwen-3.7-Plus`. DGM and HGM use outcomes from the Pro subset as evolutionary feedback, whereas ELITE uses no downstream task outcomes during evolution. For each backbone, all methods start from the same scaffold and use the same LLM for both self-modification and task solving, holding the starting agent and model fixed across methods. ELITE is run three times per backbone and reported using arithmetic means, while DGM and HGM are each run once per backbone because repeating these evaluation-driven baselines requires rerunning the full evolutionary search with substantial downstream task-evaluation cost; repetition costs and detailed evolution configurations are provided in Appendix A.5.

Metrics. We report final task success rate as the primary performance metric. To characterize the resources consumed during self-improvement, we additionally report downstream task-evaluation count, self-modification throughput (Self-Mod./h), estimated evolution-stage token usage, end-to-end wall-clock time, and estimated API cost.

5.2 MAIN RESULTS

Performance. Table 1 summarizes performance across both LLM backbones. Under `DeepSeek-V4-Flash`, ELITE improves Pro performance from 60.0% to 85.6%, slightly below DGM and HGM at 86.7%, which use downstream task evaluation feedback to guide evolution. However, its advantage emerges when directly evaluated on *SWE-bench Verified* and *Polyglot* without further evolution, achieving 73.3% and 69.3%, respectively. These results indicate that the improvements produced by ELITE transfer across benchmarks despite the absence of task-specific evaluation feedback during evolution. In contrast, DGM and HGM, evolved on Pro, show limited gains or even

regression on these transfer benchmarks. The same pattern holds under `Qwen-3.7-Plus`, although the trade-off becomes more pronounced: evolving without downstream task feedback is associated with a larger performance gap behind DGM and HGM on Pro (45.6% vs. 50.0% and 48.3%), while ELITE still achieves better cross-benchmark transfer (56.7% on Verified and 61.3% on Polyglot).

Table 1: **Main performance results across LLM backbones.** Scores are success rates (%); superscripts denote changes from the corresponding initial agent in percentage points. Shaded Avg. columns report mean scores across the three benchmarks.

	DeepSeek-V4-Flash				Qwen-3.7-Plus			
Method	**Pro**	**Verified**	**Polyglot**	**Avg.**	**Pro**	**Verified**	**Polyglot**	**Avg.**
Initial Agent	60.0	64.0	64.0	62.7	33.3	48.0	50.0	43.8
DGM	**86.7**$^{+26.7}$	64.0$^{0.00}$	62.0$^{-2.0}$	70.9$^{+8.2}$	**50.0**$^{+16.7}$	54.0$^{+6.0}$	50.0$^{0.00}$	51.3$^{+7.6}$
HGM	**86.7**$^{+26.7}$	68.0$^{+4.0}$	64.0$^{0.00}$	72.9$^{+10.2}$	48.3$^{+15.0}$	54.0$^{+6.0}$	58.0$^{+8.0}$	53.4$^{+9.7}$
ELITE	85.6$^{+25.6}$	**73.3**$^{+9.3}$	**69.3**$^{+5.3}$	**76.1**$^{+13.4}$	45.6$^{+12.3}$	**56.7**$^{+8.7}$	**61.3**$^{+11.3}$	**54.5**$^{+10.8}$

Evolution efficiency. Table 2 reports evolution resource usage. Under `DeepSeek-V4-Flash`, ELITE completes evolution in a mean of 4.90 hours, compared with 43.15 hours for DGM and 15.66 hours for HGM, yielding observed wall-clock speedups of 8.81× and 3.20× under the reported execution settings. A key source of this efficiency is the elimination of candidate task evaluations, compared with 810 for DGM and 800 for HGM. These task rollouts consume additional time, tokens, and API expenditure during evolution. ELITE achieves 16.35 self-modification attempts per hour, compared with 0.58 for DGM and 3.77 for HGM. Under `Qwen-3.7-Plus`, ELITE similarly avoids candidate task evaluations, averaging 7.35 hours, approximately $20, and 10.89 self-modification attempts per hour, with lower estimated token usage than both baselines.

Table 2: **Evolution resource usage across LLM backbones.** Times are observed wall-clock durations under the reported execution settings.

Method	**Task Eval. ↓**	**Self-Mod. /h ↑**	**Est. Tokens (M) ↓**	**Time (h) ↓**	**Cost ($) ↓**
DeepSeek-V4-Flash					
DGM	810	0.58	2900	43.15	45
HGM	800	3.77	2100	15.66	30
ELITE	**0**	**16.35**	**80**	**4.90**	**8**
Qwen-3.7-Plus					
DGM	820	0.58	2500	48.33	125
HGM	800	2.99	1800	19.70	90
ELITE	**0**	**10.89**	**70**	**7.35**	**20**

5.3 TRANSFER AND EVOLUTION ANALYSIS

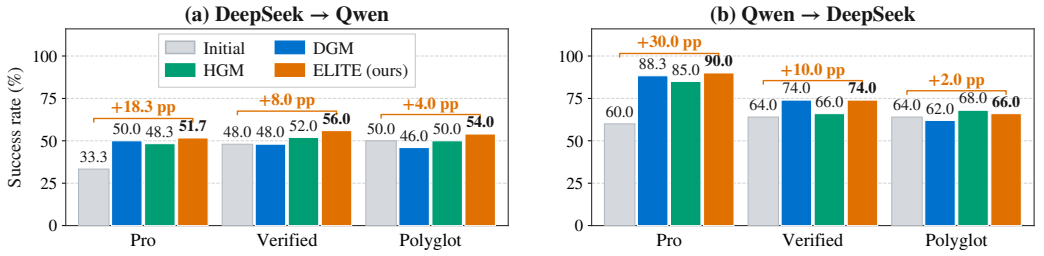


Figure 2: **Cross-backbone scaffold portability.** Scaffolds evolved in the SWE-bench Pro setting by DGM, HGM, and ELITE with one LLM backbone are transferred to the other backbone without further evolution and evaluated on Pro, Verified, and Polyglot.

Cross-backbone scaffold portability. We further evaluate cross-backbone scaffold portability by pairing scaffolds evolved with one backbone with the other backbone, without additional evolution.

All scaffolds are evolved only in the SWE-bench Pro setting; SWE-bench Verified and Polyglot are used solely for transfer evaluation and never participate in evolution. As shown in Figure 2, the DeepSeek-evolved ELITE scaffold achieves the best transferred performance on all three benchmarks under `Qwen-3.7-Plus`. In the reverse direction, the Qwen-evolved ELITE scaffold likewise achieves the best or second-best transferred performance across the three benchmarks. Overall, these results provide evidence that some scaffold improvements produced by ELITE remain useful when transferred across the two evaluated backbones.

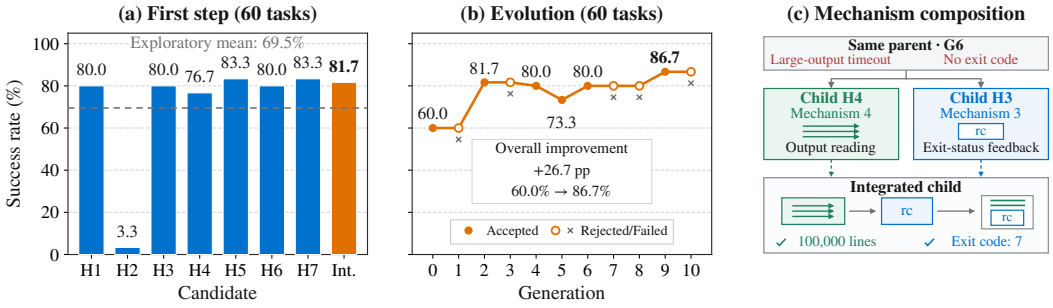


Figure 3: **Post-hoc evolution dynamics and mechanism composition in a `DeepSeek-V4-Flash` run.** (a) First-step scores on 60 Pro tasks for seven exploratory children and the integrated descendant; the dashed line marks the child mean. (b) Retained-agent performance over ten generations, starting from the initial agent at generation 0. Filled markers denote accepted updates. Crosses mark generations that did not update the parent: hypothesis composition failed before integration at generations 1, 3, and 7, whereas integrated descendants were rejected by the acceptance gate at generations 8 and 10. (c) A mechanism composition case: integration combines H4's output reading with H3's exit-status feedback, returning 100,000 lines and exit code 7 in a local probe.

Post-hoc evolution dynamics. We analyze the evolution process after it has concluded; all downstream task outcomes reported in this analysis are measured post hoc and never exposed to ELITE during evolution. Figure 3(a) compares the seven exploratory children and the integrated descendant from the first accepted step on the same 60 SWE-bench Pro tasks. The exploratory children exhibit substantial outcome variation: six achieve 76.7–83.3%, while one falls to 3.3%. The integrated descendant reaches 81.7%, outperforming five of the seven children while remaining below the highest exploratory score of 83.3%. This comparison illustrates that ELITE can construct a competitive descendant without observing candidate task performance, although integration does not necessarily recover the best-performing exploratory alternative.

Figure 3(b) shows the complete $T = 10$ evolution process. The retained agent is updated only when an integrated descendant passes the acceptance gate; otherwise, the current agent is carried forward to the next generation. Across five accepted updates, post-hoc task performance improves overall from 60.0% to 86.7%, despite two intermediate regressions. This non-monotonic trajectory is consistent with the gate's intended role: it enforces structural and operational validity without guaranteeing downstream task improvement. An additional sampling-based comparison between integrated descendants and exploratory populations is provided in Appendix A.1.3.

Mechanism-level integration. We further inspect the implementations to examine what integration carries forward. Figure 3(c) presents a representative composition case in which two same-parent exploratory children independently introduce complementary mechanisms: one enables complete output reading, while the other incorporates exit-status feedback. Rather than selecting either child as the next agent, ELITE integrates both mechanisms into a single coherent descendant. In the analyzed DeepSeek run, integration incorporates primary mechanisms from 25 of 41 eligible proposals. Together, the case study and incorporation analysis illustrate how mechanisms from different exploratory children can be carried forward into one descendant without ranking children by downstream task performance. Additional composition and refinement cases are provided in Appendix A.1.4; proposal diversity, gate outcomes, and repository-level analyses are reported in Appendix A.1.1.

5.4 Ablation and Sensitivity Analysis

Design ablations. We compare ELITE with five simplified variants. *Single exploration (direct)* sets $K = 1$ and directly uses the exploratory child as the next agent, removing both multi-branch exploration and integration. *Select-one* retains $K = 7$ same-parent exploration but replaces integration with an LLM decision selecting one child. *SIFT-style Select-one* instead selects a child using pairwise LLM judgments aggregated with a Bradley–Terry model (Fu et al., 2026). *w/o hypothesis generation* removes structured hypothesis generation and directly proposes self-modifications, retaining $K = 7$, integration, and both gates. *w/o deterministic gates* removes both validity gates, allowing all exploratory children into integration and accepting the resulting descendant without deterministic filtering. The *w/o deterministic gates* ablation uses three runs; all other ablations use a single run.

Table 3: **Design ablations of ELITE.**

Method	Pro	Verified	Polyglot	Avg.
ELITE ($K = 7$)	**85.6**	**73.3**	**69.3**	**76.1**
Single exploration	81.7	64.0	64.0	69.9
Select-one	83.3	68.0	64.0	71.8
SIFT-style Select-one	85.0	72.0	46.0	67.7
w/o hypothesis generation	85.0	68.0	60.0	71.0
w/o deterministic gates	59.4	51.3	45.3	52.0

Table 3 reports the results. Replacing integration with Select-one reduces the average score from 76.1% to 71.8%, while SIFT-style pairwise selection reaches 67.7%, indicating that the benefit of ELITE is not reproduced by either evaluated LLM-based single-child selection scheme. Single exploration further reduces the average score to 69.9%, supporting the role of multi-branch exploration together with integration. Removing structured hypothesis generation lowers the average score to 71.0%, while removing deterministic gates causes the largest degradation, to 52.0%. Together, these results support the complementary roles of diverse exploration, mechanism-level integration, structured hypothesis generation, and deterministic validity filtering.

Table 4: **Sensitivity to exploration width.** We use $K = 7$ by default.

Width K	3	5	7	9
Pro	82.8	85.0	85.6	88.9
Verified	67.3	70.7	73.3	70.0
Polyglot	61.3	68.7	69.3	68.0
Avg.	70.5	74.8	**76.1**	75.6

Sensitivity to exploration width. We vary the number of same-parent exploratory children $K \in \{3, 5, 7, 9\}$ while keeping the remaining settings fixed. Table 4 reports three-run means, with $K = 7$ as the default configuration. Among the tested widths, $K = 9$ achieves the highest Pro score of 88.9%, while $K = 7$ performs best on both transfer benchmarks. Overall, the average score increases from 70.5% at $K = 3$ to 76.1% at $K = 7$ and remains comparable at 75.6% for $K = 9$, suggesting diminishing returns from further increasing exploration width rather than a monotonic benefit from generating more exploratory children.

6 Conclusion

We introduced ELITE, an Explore-then-Integrate framework for studying whether coding-agent self-improvement can proceed without downstream candidate task evaluation. Rather than using task outcomes to rank evolved candidates, ELITE explores multiple executable modifications from the same parent and constructs the next agent by reasoning over their implementation mechanisms. Our results provide evidence for the feasibility of substantial coding-agent self-improvement without downstream candidate task feedback in the setting studied here.

Limitations. Due to the cost of evaluation-driven evolution, baseline methods are each run once per backbone, whereas ELITE is evaluated over three runs. Additional baseline repetitions would strengthen the assessment of comparative variability. Furthermore, ELITE relies on the underlying LLM's code reasoning and implementation abilities to propose and integrate modifications. Although improvements are observed with both evaluated backbones, two models provide limited evidence about how reliably this process generalizes across different capability levels. Broader backbone evaluation would help characterize this dependence.

AI USE STATEMENT

Large language models (LLMs) were used to assist with the writing and presentation of this manuscript, including improving academic English, correcting grammatical and typographical errors, refining sentence structure and clarity, and addressing formatting and typesetting issues. The scientific content of this work, including its motivation, methodology, experimental design, analysis, and conclusions, was developed and determined by the authors. No LLM-assisted output was included without human review. All such output was carefully checked, revised, and verified by the authors before inclusion in the manuscript.

ETHICS STATEMENT

This work studies self-improving coding agents in benchmark environments. Automated code modification and execution can introduce software defects or unintended changes. Our experiments use containerized task environments and restrict the permitted scope of self-modification. The deterministic validity checks used by ELITE do not establish general safety or correctness; deployment beyond the evaluated settings would require additional validation, access controls, and human oversight.

REPRODUCIBILITY STATEMENT

We describe the initial agent scaffold in Appendix A.2, the prompts used by ELITE in Appendix A.3, and the complete procedure in Appendix A.6. Benchmark subsets and task identifiers are provided in Appendix A.4, while model settings, search configurations, execution conditions, and repetition budgets are documented in Appendix A.5. Individual main-run results and sample standard deviations are reported in Appendix A.1.2.

REFERENCES

Nils Aall Barricelli et al. Esempi numerici di processi di evoluzione. *Methodos*, 6(21-22):45–68, 1954.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, volume 2024, pp. 54067–54089, 2024.

Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pp. 183–187, USA, 1985. L. Erlbaum Associates Inc. ISBN 0805804269.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Dirk Dickmanns, Jürgen Schmidhuber, and Andreas Winklhofer. Der genetische algorithmus: Eine implementierung in prolog. *Fortgeschrittenenpraktikum, Institut f ur Informatik, Lehrstuhl Prof. Radig, Technische Universit at M unchen*, 1987.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems, 2025. URL https://arxiv.org/abs/2508.07407.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Xinghong Fu, Aravinth Kulanthaivelu, and Yutaro Yamada. Self improvement via fast tree-search, 2026. URL https://arxiv.org/abs/2609.19526.

Marcos Fuster-Pena, David de Fitero-Dominguez, Antonio Garcia-Cabot, and Eva Garcia-Lopez. Repaca: Leveraging reasoning large language models for static automated patch correctness assessment. *Neurocomputing*, pp. 134583, 2026.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pp. 31–88. Elsevier, 1966.

JohnH Holland. Adaptation in natural and artificial systems. univ. *Michigan Ann Arbor, MIT*, 1975.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Alex Iacob, Andrej Jovanović, William F. Shen, Daniel Burkhardt, Meghdad Kurmanji, Nurbek Tastan, Lorenzo Sani, Niccolò Alberto Elia Venanzi, Ambroise Odonnat, Zeyu Cao, Bill Marino, Xinchi Qiu, and Nicholas D. Lane. The red queen gödel machine: Co-evolving agents and their evaluators, 2026. URL https://arxiv.org/abs/2606.26294.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Changzhi Liu, Yilun Liu, Sikuan Yan, Volker Tresp, and Yunpu Ma. Mendel gödel machine: Recursive self-improving coding agents via comparative evolution, 2026. URL https://arxiv.org/abs/2608.07645.

Paul Gauthier. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, December 2024. Accessed: 2025-04-16.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186, 2024.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for swe agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15265–15290, 2026.

Buvaneswari A Ramanan, Manzoor A Khan, and Abhinav Rao. Aspire: a multi-agent framework for execution-free code analysis and repair. In *2024 IEEE International Conference on Big Data (BigData)*, pp. 8811–8813. IEEE, 2024.

Zhe Ren, Yimeng Chen, Dandan Guo, Guowei Rong, Tonghui Li, RB Xiong, Qingfeng Lan, Wenyi Wang, Li Nanbo, Yibo Yang, et al. Self-improvements in modern agentic systems: A survey. *arXiv preprint arXiv:2607.13104*, 2026.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

Jürgen Schmidhuber. Godel machines: Self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs.LO/0309048*, 2003.

Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1): 105–130, 1997.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, et al. Swe-rm: Execution-free feedback for software engineering agents. In *International Conference on Learning Representations*, volume 2026, pp. 143426–143446, 2026.

Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms*. University of Pittsburgh, 1980.

Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 20032–20051, 2024.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-g\" odel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations*, volume 2026, pp. 80356–80386, 2026a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026b. URL https://arxiv.org/abs/2607.12227.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-evolving agents: Open-ended self-improvement via experience sharing. *arXiv preprint arXiv:2602.04837*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, Huaxiu Yao, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. Rrsi: Regularized recursive self-improvement of agent harnesses, 2026. URL https://arxiv.org/abs/2609.24972.

Jinghan Xu, Yikai Zhang, Aili Chen, Weiyuan Li, Jiaqing Liang, and Deqing Yang. Verify smarter, evolve further: Efficient harness evolution through behavior-aware verification, 2026a. URL https://arxiv.org/abs/2608.27311.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026b.

Aashish Yadavally, Hoan Nguyen, Laurent Callot, and Gauthier Guinet. Large language model critics for execution-free evaluation of code changes. *arXiv preprint arXiv:2501.16655*, 2025.

Boyang Yang, Xiangliang Hu, Luyao Ren, Yanjun Chen, Bach Le, Tegawendé F Bissyandé, and Haoye Tian. A single patch is not enough: Deterministic fusion of repair candidates. *arXiv preprint arXiv:2607.01597*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents, 2026. URL https://arxiv.org/abs/2606.28436.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026a.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026b.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026c.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. In *Findings of the association for computational linguistics: EACL 2024*, pp. 2232–2242, 2024.

APPENDIX CONTENTS

A.1 Additional Results 15
A.1.1 Detailed Evolution Traces 15
A.1.2 Results Across Three Runs 16
A.1.3 Sampled Exploration–Integration Comparison 16
A.1.4 Code and Outcome Case Studies 18
A.2 Initial Agent 22
A.2.1 Repository overview 22
A.2.2 Tools 22
A.2.3 Task Prompt 23
A.3 Prompts Used by ELITE 23
A.3.1 Improvement Hypothesis Generation 23
A.3.2 Exploratory Self-Modification 25
A.3.3 Integration 25
A.4 Benchmarks and Task Subsets 26
A.4.1 SWE-bench Pro 27
A.4.2 SWE-bench Verified 27
A.4.3 Polyglot 27
A.5 Experimental Settings 27
A.5.1 Evolution Configuration 27
A.6 ELITE Algorithm 31

A Appendix

A.1 Additional Results

A.1.1 Detailed Evolution Traces

Repository-level performance. Figure 4 shows that ELITE's aggregate gains are distributed across multiple software repositories. Relative to the corresponding initial agent, the final DeepSeek scaffold improves on 8 of 11 repositories, ties on 2, and declines on 1; the Qwen scaffold improves on 7, ties on 2, and declines on 2. These results demonstrate the breadth of the observed improvements, despite non-monotonic intermediate DeepSeek scores. To limit evaluation cost, Qwen post-hoc evaluation covers only the initial and final agents; intermediate checkpoints are omitted.

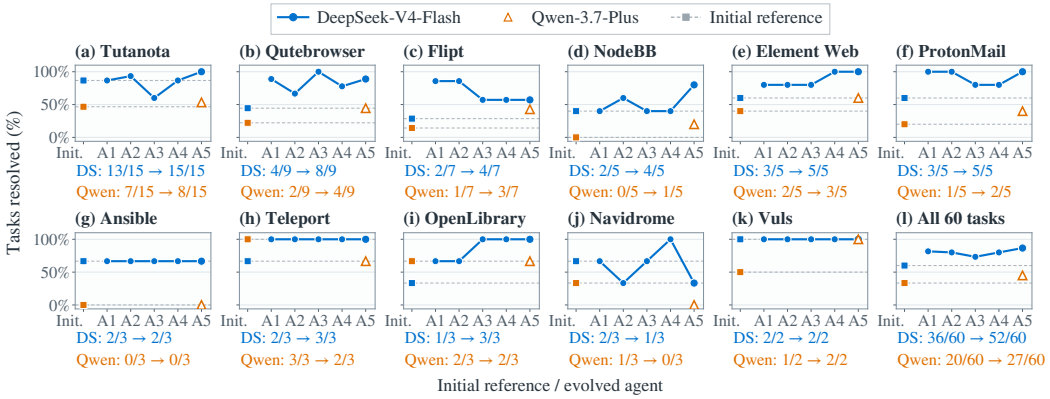
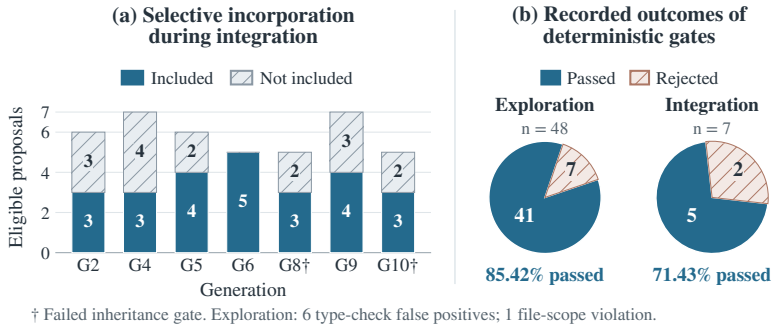


Figure 4: **Repository-level performance across evolution.** Post-hoc SWE-bench Pro solve rates across 11 repositories and the aggregate 60-task set. Blue circles denote DeepSeek-V4-Flash checkpoints; orange triangles denote the final Qwen-3.7-Plus checkpoint. Colored squares and gray dashed lines mark initial references. Annotations report initial-to-final solved counts.

Integration selectivity and gate outcomes. Figure 5(a) summarizes primary-mechanism incorporation based on manual inspection of the recorded patches and integration logs. Filled bars denote eligible proposals whose primary mechanism is incorporated into the integrated descendant, while hatched bars denote proposals not incorporated. Figure 5(b) reports deterministic gate outcomes: blue denotes passed candidates and hatched orange denotes rejected candidates. The exploration gate accepts 41/48 candidates and the acceptance gate accepts 5/7 integrated descendants. The symbol † marks integration attempts whose resulting descendant is rejected by the acceptance gate. The recorded exploration rejections consist of six conservative type-check false positives and one file-scope violation; the two acceptance rejections arise from a tool-order violation and a call-limit violation.

Exploration diversity. Figure 6 visualizes all seven integration attempts in the `DeepSeek-V4-Flash` run. Each row corresponds to one shared parent; H1–H7 are independently implemented exploratory children, and the rightmost card is the resulting integrated descendant. The card title summarizes the primary mechanism of each patch. Colored squares indicate which scaffold components are modified, following the legend for Agent, Protocol, Bash, Editor, Prompt, and Utilities. A red border with $\times$ marks an exploratory child rejected by the deterministic gate. The annotation L denotes the total number of added and deleted lines, while h denotes the number of diff hunks. The black bar visualizes $\log(1 + L)$ so that patch sizes with different scales remain visually comparable. For each generation, the reported mean diff distance summarizes textual dissimilarity among exploratory patches, shown for all returned children and for gate-passing children separately. The blue strip beneath each integrated descendant shows its textual similarity to H1–H7, ranging from 0 to 1. These quantities describe implementation structure only and should not be interpreted as semantic quality or causal contribution.



† Failed inheritance gate. Exploration: 6 type-check false positives; 1 file-scope violation.

Figure 5: **Integration choices and deterministic gate outcomes in the `DeepSeek-V4-Flash` run.** Left: incorporation of primary mechanisms from eligible exploratory proposals. Right: pass and rejection outcomes of the exploration and integration gates.

A.1.2 Results Across Three Runs

Table 5 reports the individual performance results underlying the three-run ELITE means in the main text. For each backbone, Run 1–3 denote the three reported runs. We report arithmetic means $\pm$ sample standard deviations (SD, $n = 3$, with an $n - 1$ denominator).

Table 5: **Per-run performance of ELITE.** Scores are success rates (%); SDs are in percentage points. Verified and Polyglot are evaluated directly after evolution without further adaptation.

	`DeepSeek-V4-Flash`			`Qwen-3.7-Plus`		
Run	**Pro**	**Verified**	**Polyglot**	**Pro**	**Verified**	**Polyglot**
Run 1	86.7	72.0	68.0	45.0	56.0	60.0
Run 2	85.0	74.0	68.0	45.0	58.0	62.0
Run 3	85.0	74.0	72.0	46.7	56.0	62.0
Mean $\pm$ SD	85.6 ± 1.0	73.3 ± 1.2	69.3 ± 2.3	45.6 ± 1.0	56.7 ± 1.2	61.3 ± 1.2

A.1.3 Sampled Exploration–Integration Comparison

We use a post-hoc diagnostic on one `DeepSeek-V4-Flash` evolution run to compare integrated candidates with their valid exploratory populations. Evaluations occur after acceptance decisions and never inform evolution. Figure 7 shows the five accepted updates.

Tasks and sampling. We randomly draw 10 SWE-bench Pro tasks independently of agent outcomes and keep this subset fixed across all comparisons. Each integrated candidate is evaluated once per task. Let $m_t \geq 2$ denote the number of exploratory children passing the exploration gate at step t. Using a fixed random seed, we randomly order tasks and children, then assign two distinct children to each task using a balanced cyclic schedule. The resulting 20 distinct child–task pairs cover all valid children in this study, with evaluation counts differing by at most one between children. Assignments use neither downstream outcomes nor integration decisions.

Reported quantities and evaluation budget. The exploration estimate is the mean binary success rate over the 20 sampled child–task outcomes; the integration score is the success rate on the 10 integrated-agent rollouts. Their difference, in percentage points, is the integration gap. Each comparison therefore uses $20 + 10 = 30$ rollouts. The complete diagnostic covers seven integration attempts, requiring $7 \times 30 = 210$ fresh rollouts; the five accepted comparisons shown in Figure 7 account for $5 \times 30 = 150$ of these.

Results and interpretation. The integrated descendant exceeds the sampled exploratory mean at each of the five accepted updates, with step-averaged scores of 68.0% and 48.0%, respectively



Figure 6: **Same-parent exploration across integration attempts.** Each row shows H1–H7 independently implemented from a shared parent and the resulting integrated descendant. Card annotations summarize mechanisms and modified components; red crosses mark gate rejections. L denotes added/deleted lines, h diff hunks, black bars show $\log(1 + L)$, blue strips show textual similarity to H1–H7, and row labels report mean diff distances.

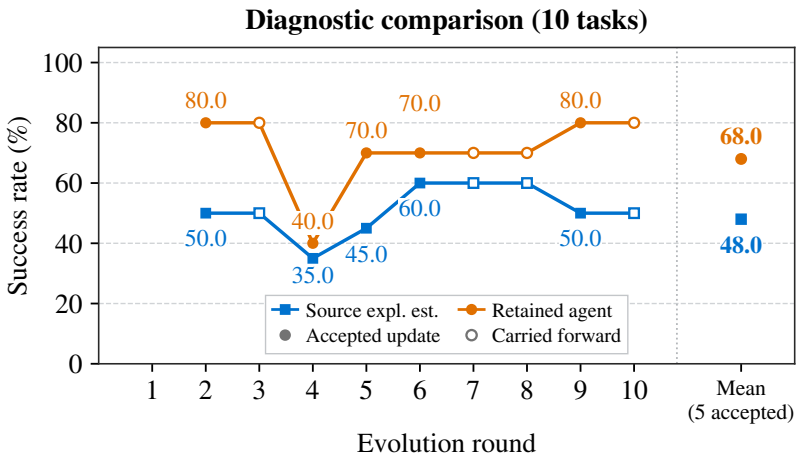


Figure 7: **Sampled exploration–integration comparison.** Filled markers show recorded comparisons at accepted rounds 2, 4, 5, 6, and 9. Hollow markers carry forward the preceding retained agent and its source exploratory estimate after rejection or failure; they are not outcomes of the unaccepted children or newly generated exploratory populations. The Mean column equally averages the five accepted comparisons, counting each once.

(a 20.0 pp gap). We average these five comparisons equally. Averaging all ten displayed rounds would instead weight checkpoints by their retention duration. These are descriptive comparisons on one fixed subset and one accepted lineage; they do not establish a general advantage across tasks or runs. Because each child solves only a subset of the diagnostic tasks, the samples estimate population-average performance on this subset and cannot identify an oracle-best child.

A.1.4 Code and Outcome Case Studies

The first three cases analyze the `DeepSeek-V4-Flash` run; the final three analyze the `Qwen-3.7-Plus` run. G denotes the original generation within each run.

Combining complementary code changes. Figure 8 links two G6 exploratory modifications to the integrated implementation and their observed behavior. H3 adds exit-status feedback, while H4 introduces continuous output reading. Under identical local probes, H3 reports silent failures but times out on large output; H4 returns all output but omits the exit status. The integrated tool preserves both properties, returning all 100,000 lines together with the correct failure code. These checks replay the archived `tools/bash.py` implementations with `(exit 7)` and `seq 1 100000; (exit 7)`, each under a 3-second timeout, without LLM calls or benchmark task evaluation.

Repairing an exploratory implementation. Figure 9 compares the G2-H6 manual tool-action parser with its integrated counterpart. H6's brace matching skips only double-quoted strings, so braces inside single-quoted commands can disrupt extraction. Integration adds a helper that handles both quote styles. For the displayed input containing `printf {`, H6 extracts no valid action, whereas the integrated parser recovers the command. The table shows the resulting correction on three constructed edge cases while preserving the two formats already handled by H6. These checks establish local parser behavior, without attributing downstream score changes to this particular parsing path.

A review mechanism used during task solving. Figure 10 connects the review code introduced by G9-H1 and retained during integration to an actual NodeBB trajectory. The added call receives the exported patch and preceding solve history. During review, the agent removes three Redis artifacts containing 15,710 added text lines while preserving all three source-code diff blocks unchanged, reducing the patch from six files to three. The review code is in `agents/universal_coding_agent.py`. The final task is resolved; because the pre-review

H3 · Exit-status feedback

```
+ output, exit_code =
↪   _extract_exit_code(output)
   ... (omitted)
+ return output, error, exit_code
   ... (omitted)
+ if exit_code is not None:
+     result += "\nExit code: " +
↪   str(exit_code)
```

H4 · Continuous output reading

```
+ chunk = await stream.read(65536)
   ... (omitted)
+ self._stdout_task =
↪   asyncio.ensure_future(
+     self._read_stream(
↪   self._process.stdout,
↪   self._stdout_lines)
+ )
```

Integrated · H3 + H4 in the same tool

```
+ self._stdout_task = asyncio.ensure_future(
+     self._read_stream(self._process.stdout, self._stdout_lines)
+ )
   ... (omitted)
+ output, exit_code = _extract_exit_code(output)
   ... (omitted)
+ return output, error, exit_code
```

Version	Silent failure exit code	Large output returned lines	Large output exit code
Parent	—	Timeout	—
H3	7	Timeout	—
H4	—	100,000	—
Integrated	**7**	**100,000**	**7**

Figure 8: **Complementary code changes and their observed effects.** Matching blue and green highlights connect H3's exit-status feedback and H4's output reading to the integrated tool. The table compares all four versions on the same local probes; a dash denotes no returned exit status.

```
{'tool_name': 'bash', 'tool_input': {'command': 'printf {'}}
```

H6 · Only double quotes are skipped

```
- if ch == '"':
-     i += 1
-     while i < n:
   ... (omitted)
-         if text[i] == '"':
-             i += 1
-             break
```

Integrated · Both quote styles

```
+ if ch in ("'", '"'):
+     i = _skip_string(text, i)

_skip_string helper (excerpt)
+ quote = text[i]
   ... (omitted)
+     if text[i] == quote:
+         return i + 1
```

Input format	Parent	H6	Integrated
Tagged JSON (control)	Pass	Pass	Pass
Bare JSON	Fail	Pass	Pass
Bare dict: literal {	Fail	Fail	Pass
Bare dict: literal }	Fail	Fail	Pass
Function call: literal {	Fail	Fail	Pass

Figure 9: **An implementation refinement and five format checks.** Before/after excerpts reconstructed from archived H6 and integrated sources show the change in quote handling. The table compares the exact archived manual parsers on the same five inputs; parsed commands were not executed.

patch has no separate official evaluation, the demonstrated benefit is patch cleanup rather than a measured failure-to-success transition.

Archived scaffold diff · one review after solving

```
- chat_with_agent(instruction, model=self.code_model, msg_history=[],
↪   logging=safe_log)
+ history = chat_with_agent(instruction, model=self.code_model,
↪   msg_history=[], logging=safe_log) or []
  ... (omitted)
+ code_diff = self.get_current_edits()
+ review_message = (
+     "Review this patch; fix problems or state completion.\n\n"
+     "<attempted_patch>\n"
+     f"{code_diff}\n"
+     "</attempted_patch>"
+ )
+ chat_with_agent(review_message, model=self.code_model,
↪   msg_history=history, logging=safe_log)
```

NodeBB task · recorded cleanup command and its output

```
$ cd /app && rm -rf appendonlydir dump.rdb && git status --short
  M src/database/mongo/sorted.js
  M src/database/postgres/sorted.js
  M src/database/redis/sorted.js
```

Figure 10: **From an evolved code change to observed task behavior.** The archived review patch and the recorded NodeBB cleanup command show how the added review removes non-source artifacts from the exported patch.

Reducing model-mediated test discovery. Figure 11 shows how G5 integration retains H7's deterministic test discovery while omitting H3's additional model-based planning stage. The archived H7 and integrated discovery methods are identical: they scan repository files and construct test commands, replacing the parent's `chat_with_agent` call. Local fixtures confirm file discovery and command construction for Python, JavaScript, Go, and Rust without model calls. This eliminates the model-mediated discovery stage; subsequent test execution remains agent-mediated.

Parent · Model-based discovery

```
- history = chat_with_agent(
↪   instruction,
↪   model=self.code_model,
↪   msg_history=[],
↪   logging=safe_log)
  ... (omitted)
- if not history:
-     return "No regression test
↪   summary was produced."
  ... (omitted)
- return str(content)
```

H7 → Integrated · File scan

```
+ for root, dirs, files in os.walk(
↪   repo):
  ... (omitted)
+ test_files.append(relpath)
  ... (omitted)
+ commands.append(f"pytest -rA {'
↪   '.join(py_test_files)}")
  ... (omitted)
+ return "\n".join(lines)
```

Local fixture	Files	Generated command
Python	2	`pytest -rA tests/test_alpha.py tests/test_beta.py`
JavaScript	2	`npx jest src/alpha.test.js src/beta.spec.js`
Go	1	`go test ./...`
Rust	1	`cargo test`

Figure 11: **Deterministic test discovery retained during Qwen integration.** The code change replaces model-mediated discovery with file scanning. H7 and the integrated version return the same outputs on four local fixtures; the displayed commands are generated, not executed.

Composing and constraining edit recovery. Figure 12 connects G8-H6's replacement fallback and G8-H4's immediate Python/JSON checks to their integrated implementation. Integration restricts H6's similarity-based matching to trailing-whitespace normalization. On the constructed inputs, H6 replaces `timeout_ms` when asked to replace the absent `timeout_s`; integration rejects this mismatch while retaining recovery from trailing-space differences. It also retains H4's checks on newly created Python and JSON files, which H6 does not check. The five local checks illustrate selective refinement of a child mechanism alongside composition. Verification reports errors after writing and does not roll back invalid edits.

H6 · Similarity-based fallback

```
+ ratio = difflib.SequenceMatcher(
↪   None, old_text, window).ratio()
  ... (omitted)
+ if best_ratio >= threshold and
↪   best_match is not None:
+     return best_match
```

H4 · Python / JSON checks

```
+ ast.parse(content)
  ... (omitted)
+ json.loads(content)
  ... (omitted)
+ result += verify_file(path_obj)
```

Integrated · Restricted fallback + immediate checks

```
+ old_stripped = [line.rstrip() for line in old_lines]
  ... (omitted)
+ window_stripped = [line.rstrip() for line in window]
+ if old_stripped == window_stripped:
+     return ''.join(window)
  ... (omitted)
+ result += verify_file(path_obj)
```

Local check	Parent	H6	H4	Integrated
Exact replacement	Pass	Pass	Pass	**Pass**
Trailing-space replacement	Fail	Pass	Fail	**Pass**
Preserve different identifier	Pass	Fail	Pass	**Pass**
Report invalid Python on create	Fail	Fail	Pass	**Pass**
Report invalid JSON on create	Fail	Fail	Pass	**Pass**

Figure 12: **Qwen integration combines edit recovery with immediate checks.** Blue traces H6's matching mechanism and its restricted integrated form; green traces H4's verification. Pass/Fail indicates the specified behavior on each constructed local input, rather than benchmark task success.

Using change-aware regression checks. Figure 13 traces G9-H1's unchanged test-selection code through a recorded Qutebrowser task involving dark-mode image classification. The mechanism identifies changed modules, locates tests that reference them, and falls back to all discovered tests when necessary. In this trajectory, it selects two of 176 test files and executes 141 passing test cases under a headless Qt configuration; the official evaluator also resolves the task. This shows that the evolved guidance is used during task solving without attributing final success or runtime reduction to this mechanism alone.

G9-H1 → Integrated · Change-aware test selection

```
+ changed_modules = self._get_changed_modules()
+ impacted_tests = self._find_impacted_tests(test_files, changed_modules)
  ... (omitted)
+ use_focused = len(impacted_tests) > 0 and len(impacted_tests) < len(
↪   test_files)
+ selected_tests = impacted_tests if use_focused else test_files
  ... (omitted)
+ py_test_files = sorted([f for f in selected_tests if f.endswith('.py')])
```

Qutebrowser · Recorded discovery output

```
  Change-impact analysis: 2 of 176 tests selected
  Changed modules: qutebrowser.browser.webengine.darkmode, darkmode
```

Recorded test command and result

```
$ cd /app && QT_QPA_PLATFORM=offscreen python -m pytest -rA
↪   tests/unit/browser/webengine/test_darkmode.py
↪   tests/unit/config/test_qtargs.py 2>&1 | tail -20

  141 passed in 0.90s
```

Figure 13: **From Qwen's evolved test selection to recorded task execution.** The retained code, discovery output, and executed command establish the connection.

A.2 Initial Agent

A.2.1 Repository overview

The initial scaffold Σ_0 is an executable source-code repository containing the agent's prompts, tools, utilities, and control logic. Given a task description and a working repository, the agent uses an LLM to inspect the codebase, make changes, and perform local checks. It exposes two tools: `bash` executes shell commands, and `editor` supports file inspection, creation, and whole-file editing. All methods start from the same scaffold for a given backbone, and the same initial agent is evaluated across all three benchmarks. Figure 14 shows the repository structure.

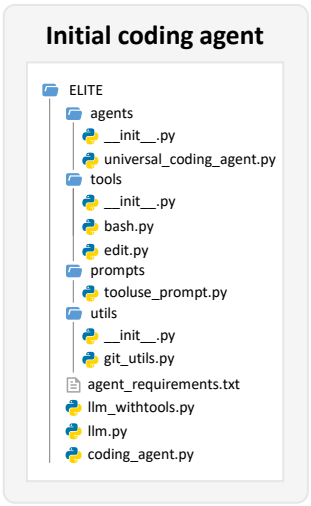


Figure 14: Initial coding agent repository.

A.2.2 Tools

The initial agent has two tools. Together, they support repository inspection, code modification, and local verification.

Bash. The `bash` tool accepts a shell command through its `command` argument and executes it in the task environment. The agent can use it to navigate and search the repository, inspect dependencies and build configurations, run tests, and review the resulting Git diff. The implementation collects standard output and filtered standard error and applies a 120-second command timeout. Each invocation starts a new Bash session, so commands should explicitly establish their working directory when necessary. Changes to files remain available to subsequent tool calls.

Edit. The file-editing tool is implemented in `tools/edit.py` and exposed to the model under the name `editor`. It takes an absolute `path` and supports three commands. The `view` command displays a file with line numbers or lists a directory to a depth of two levels; file contents are truncated after 10,000 characters before line numbers are added. The `create` command writes the supplied `file_text` to a new file and rejects an existing path. The `edit` command replaces the complete contents of an existing file with `file_text`. This initial interface therefore performs file creation and whole-file editing.

A.2.3 Task Prompt

The default execution path uses a brief system instruction to establish a careful, generalist coding-agent role and encourage tool use for inspection, editing, and testing. Listing 1 presents the main task template, shared across benchmarks, with the task description, repository location, and optional context supplied at runtime. Placeholder names are normalized for readability.

Listing 1: Main task prompt of the initial coding agent.

```
You are a generalist coding agent. Modify the repository at `{git_tempdir
    }` to solve the task below.

<task>
{problem_statement}
</task>

<repository_context>
{repository_context}
</repository_context>

<test_hint>
{test_description}
</test_hint>

Requirements:
1. Be dataset-agnostic: do not rely on benchmark names, task ids, hidden
    tests, or known leaderboard artifacts.
2. Be language-agnostic: inspect repository structure, dependency files,
    build files, and existing tests before deciding what to edit.
3. Prefer minimal, source-code-focused patches. Do not edit benchmark
    tests or create hard-coded task-id branches.
4. Use available tools to inspect files, run targeted tests, and edit
    code.
5. Before finishing, check `git diff` and ensure the patch is non-empty,
    relevant, and does not include accidental artifacts.
```

A.3 Prompts Used by ELITE

This section presents the templates for hypothesis generation, exploratory self-modification, and integration. Braced placeholders denote inputs supplied at runtime. Hypothesis generation uses separate system and user messages; exploration and integration supply self-modification tasks to the coding agent described in Appendix A.2.

A.3.1 Improvement Hypothesis Generation

System prompt. Listing 2 specifies the objectives and constraints for jointly generating focused, distinct, and complementary improvement hypotheses grounded in the current implementation.

Listing 2: System prompt for improvement hypothesis generation.

```
You design one generation of focused exploratory mutations for a self-
    evolving coding agent.
```

```
Given the exact current runtime source and a compact evolution report,
    generate the requested number of hypotheses jointly. The set should
    contain the most promising, distinct, and complementary behavioral
    improvements visible in the current implementation.

Each hypothesis must:
- address one concrete behavioral limitation;
- propose one focused mechanism that can be implemented independently;
- be complete enough to produce a meaningful child, but not bundle
    unrelated features;
- modify only the runtime components necessary for that mechanism;
- remain grounded in the supplied source rather than a fixed category
    list.

Across the set, avoid near-duplicates and unnecessary concentration in
    one subsystem. Do not treat tools, model wrappers, prompts, memory,
    planning, or new modules as inherently higher-impact than existing
    mechanisms. Do not generate extra candidates for a later selection
    stage.

Do not use benchmark IDs, hidden tests, reference solutions, leaderboard
    artifacts, or benchmark-specific parsers. Return one JSON object only.
```

User prompt. Listing 3 supplies the current runtime source, a compact evolution report, and the required JSON schema. Each hypothesis describes a capability objective, a limitation, a proposed mechanism, acceptance conditions, and its distinction from the other hypotheses. Structurally invalid responses trigger a request for corrected JSON.

Listing 3: User prompt for improvement hypothesis generation.

```
## Compact evolution report
{evolution_report}

## Exact current runtime source
{parent_source}

Generate exactly {hypothesis_count} focused, distinct, and complementary
    hypotheses. Return this schema:

{
  "hypotheses": [
    {
      "hypothesis_id": "H1",
      "capability": "Behavioral capability to improve",
      "limitation": "Concrete limitation visible in the current source",
      "mechanism": "One focused implementation mechanism",
      "acceptance": "Observable conditions showing the mechanism is
    complete",
      "distinctive_basis": "Why this hypothesis is meaningfully different
     from the others"
    }
  ]
}

Requirements:
- Include exactly {hypothesis_count} hypotheses, identified H1 through H{
    hypothesis_count}.
- Choose the final set directly; there is no later curation stage.
- Each hypothesis must represent one coherent intervention, not a broad
    redesign or a tiny cosmetic edit.
- Make the hypotheses complementary enough that a later synthesis step
    can combine compatible successes.
```

```
- Keep every field concise.
```

A.3.2 EXPLORATORY SELF-MODIFICATION

Exploration task prompt. Listing 4 converts each hypothesis into an executable self-modification task. The capability objective, limitation, mechanism, and acceptance conditions are inserted into the corresponding placeholders. Each exploratory child starts from the same parent and implements one focused intervention.

Listing 4: Task prompt for implementing an exploratory self-modification.

```
# Focused Self-Modification Task

## Capability Objective
{capability}

## Current Limitation
{limitation}

## Mechanism to Implement
{mechanism}

## Acceptance Conditions
{acceptance}

## Scope
Implement exactly one focused behavioral intervention. Touch the smallest
     coherent set of runtime components needed to make the mechanism
    complete and usable on the default solve path. Reuse existing
    interfaces where possible. Do not add adjacent features or broadly
    redesign the agent.
Do not modify tests, benchmark/evaluation infrastructure, proxy/evolution
     logic, run artifacts, or encode benchmark-specific information.
```

A.3.3 INTEGRATION

Integration task prompt. Listing 5 guides the construction of one integrated descendant from the unmodified common parent. The input manifest provides candidate descriptions, deterministic check results, and references to complete patches. The integrator considers all candidates at the manifest level, initially shortlists one to three compatible children, inspects their complete patches, and reimplements selected mechanisms as a coherent descendant. Additional patches are inspected only when needed to supply a missing compatible mechanism. Downstream task outcomes are not provided to the integrator.

Listing 5: Task prompt for integrating exploratory modifications.

```
# Integrated Self-Modification Task

This is the synthesis step of the current evolution generation.

The repository currently contains the unmodified parent agent. Before
    changing code, read:

    {manifest_path}

The manifest contains compact evidence for all {child_count} exploratory
    children and exact bounded patch paths.

Use this sequence:
1. Consider every child from the manifest-level evidence.
2. Ignore empty children and children that failed deterministic checks.
```

```
3. Shortlist one to three compatible children with the strongest
    behavioral mechanisms.
4. For each shortlisted child, read every file in its `patch_chunks` list,
     in order. This reconstructs its complete patch without a large
    clipped tool response.
5. Begin implementation after the shortlist has been fully inspected.
    Read another child's full patch only when its manifest evidence
    identifies a missing, compatible mechanism.

Create one coherent integrated child that captures the strongest
    compatible capability improvements discovered across the exploratory
    children.

Requirements:
- Consider every exploratory child at manifest level, but do not
    mechanically merge every patch.
- Keep useful mechanisms; reject redundant, conflicting, incomplete,
    speculative, or disconnected parts.
- Reimplement selected ideas as one consistent design starting from the
    current parent repository.
- Preserve a clear default solve path. Reuse existing mechanisms when
    they already provide the required behavior; any new component or
    control-flow change must be necessary, reachable, and fully
    integrated.
- Prefer a small number of mutually reinforcing capabilities over an
    incoherent bundle. Do not favor infrastructure changes merely because
     several children modify the same central subsystem.
- Keep the default solve path bounded: do not stack multiple full-agent
    passes or open-ended repair loops. Added stages must replace or
    simplify existing work rather than multiply end-to-end model calls.
- Do not spend the tool budget exhaustively reading low-value or invalid
    patches; move to implementation once the shortlist is understood.
- Before finishing, inspect `git diff` and ensure the resulting patch is
    non-empty and internally consistent.
- Patch size and file count are neither rewards nor penalties.
- Do not modify tests, benchmark/evaluation infrastructure, proxy/
    evolution logic, run artifacts, or encode benchmark-specific
    information.

The result of this task is the single integrated child proposed for this
    generation.
```

A.4 Benchmarks and Task Subsets

We evaluate the agents on fixed subsets of SWE-bench Pro, SWE-bench Verified, and Aider Polyglot, containing 60, 50, and 50 tasks, respectively. SWE-bench Pro is used to provide evolutionary feedback for the baselines, whereas SWE-bench Verified and Polyglot are used only for post-evolution transfer evaluation and never provide feedback during ELITE evolution. The benchmark referred to as Verified in the main results is specifically SWE-bench Verified. The task identifiers are listed in Tables 6–8.

Task execution. Each task is run in a separate container with a fresh agent process and the benchmark-specified repository state. An agent run may include multiple model responses, tool calls, and local tests. The resulting code changes are assessed by the corresponding benchmark evaluator.

Evolution feedback. DGM and HGM use SWE-bench Pro task outcomes during evolution. ELITE does not use downstream task outcomes from evolved candidates to decide integration or the next-agent update. Running available tests while solving an individual task is distinct from returning benchmark evaluation results to the outer self-improvement procedure.

A.4.1 SWE-BENCH PRO

SWE-bench Pro consists of repository-level software engineering tasks that require modifying an existing codebase.[1] The agent receives the problem statement and, when available, additional requirements, interface specifications, and test hints. Its generated patch is assessed using the SWE-bench Pro evaluation harness. We use a subset of 60 tasks spanning 11 repositories for the main evolution comparison. Table 6 lists these tasks by repository. Each complete identifier is obtained by concatenating the group prefix, the hash component, and the suffix.

A.4.2 SWE-BENCH VERIFIED

SWE-bench Verified is a human-validated subset of SWE-bench containing 500 repository-level tasks derived from real GitHub issues.[2] The agent receives an issue description and modifies the corresponding repository. The evaluator checks whether the issue is resolved and whether the required existing tests continue to pass. We follow DGM Zhang et al. (2026a) and use the same 50-task subset, comprising 25 Django tasks and 25 Sphinx tasks. Table 7 lists the task numbers under their respective repository prefixes.

A.4.3 POLYGLOT

The Aider Polyglot benchmark contains 225 programming exercises drawn from Exercism and covers C++, Go, Java, JavaScript, Python, and Rust.[3] The agent edits the provided solution files to implement the requested behavior. We follow DGM Zhang et al. (2026a) and use the same 50-task subset for transfer evaluation, including 4 C++ tasks, 8 Go tasks, 11 Java tasks, 8 JavaScript tasks, 10 Python tasks, and 9 Rust tasks. Table 8 lists the exercises under their language prefixes. Each task is attempted in one agent run, which may include tool use and local testing. After the run, the harness retains the modified solution files, restores the benchmark test revision, and executes the language-specific evaluation command. A task is marked as resolved when this command exits successfully. Final evaluation feedback is not followed by an additional agent repair attempt.

A.5 EXPERIMENTAL SETTINGS

A.5.1 EVOLUTION CONFIGURATION

Shared model settings. We use DeepSeek-V4-Flash and Qwen-3.7-Plus as LLM backbones. All methods share the same initial scaffold for a given backbone. The maximum output budget per model response is 16,384 tokens (`MAX_OUTPUT_TOKENS=16384`); hypothesis generation uses the same budget (`PORTFOLIO_MAX_TOKENS=16384`). Thinking is enabled through `DEEPSEEK_THINKING=enabled` for DeepSeek and `QWEN_ENABLE_THINKING=1` for Qwen.

Controlled execution environment. Agent execution and deterministic validity checks run inside isolated `Docker` sandbox containers built from the experiment runtime image. Each exploratory or integrated child is executed in a separate containerized working copy with the timeout shown in Table 9; containers are removed after the attempt; the host process handles orchestration and records status and artifacts. This containment isolates child mutations and tool execution from the host environment.

Evolution settings. Table 9 summarizes the main search parameters. Parallel workers denotes the configured concurrency limit. ELITE generates $K = 7$ exploratory children per valid hypothesis portfolio for both backbones. It uses eight workers for both backbones; the maximum exploration concurrency is therefore $\min(8, 7) = 7$. Integration follows exploration. HGM uses the same search configuration for both backbones, with 20 parallel workers shared by expansion and evaluation.

Search budgets. Following HGM's evaluation-budget convention (Wang et al., 2026a), HGM is configured for 800 task evaluations, and DGM is compared over a search prefix with approximately

[1] `https://huggingface.co/datasets/ScaleAI/SWE-bench_Pro`
[2] `https://openai.com/index/introducing-swe-bench-verified/`
[3] `https://aider.chat/2024/12/21/polyglot.html`

Table 6: The 60 SWE-bench Pro task IDs, grouped by repository. Each ID is the concatenation of the group prefix, the hash component, and the suffix. An em dash denotes an empty suffix.

Hash component	Suffix
`instance_ansible__ansible-` (3 tasks)	
de5858f48dc9e1ce9117034e0d7e76806f420ca8	-v1055803c3a812189a1133297f7f5468579283f86
0fd88717c953b92ed8a50495d55e630eb5d59166	-vba6da65a0f3baefda7a058ebbd0a8dcafb8512f5
cd473dfb2fdbc97acf3293c134b21cbbcfa89ec3	-vba6da65a0f3baefda7a058ebbd0a8dcafb8512f5
`instance_element-hq__element-web-` (5 tasks)	
5e8488c2838ff4268f39db4a8cca7d74eecf5a7e	-vnan
66d0b318bc6fee0d17b54c1781d6ab5d5d323135	-vnan
18c03daa865d3c5b10e52b669cd50be34c67b2e5	-vnan
e15ef9f3de36df7f318c083e485f44e1de8aad17	—
ca58617cee8aa91c93553449bfdf9b3465a5119b	-vnan
`instance_flipt-io__flipt-` (7 tasks)	
abaa5953795afb9c621605bb18cb32ac48b4508c	—
3d5a345f94c2adc8a0eaa102c189c08ad4c0f8e8	—
c6a7b1fd933e763b1675281b30077e161fa115a1	—
2ca5dfb3513e4e786d2b037075617cccc286d5c3	—
507170da0f7f4da330f6732bffdf11c4df7fc192	—
292fdaca9be39e6a921aaa8874c011d0fdd3e874	—
0b119520afca1cf25c470ff4288c464d4510b944	—
`instance_future-architect__vuls-` (2 tasks)	
f0b3a8b1db98eb1bd32685f1c36c41a99c3452ed	—
e52fa8d6ed1d23e36f2a86e5d3efe9aa057a1b0d	—
`instance_gravitational__teleport-` (3 tasks)	
37c3724d0d6637e959e39408ee351565d73afe71	-vee9b09fb20c43af7e520f57e9239bbcf46b7113d
65438e6e44b6ce51458d09b7bb028a2797cfb0ea	-vce94f93ad1030e3136852817f2423c1b3ac37bc4
0ac7334939981cf85b9591ac295c3816954e287e	—
`instance_internetarchive__openlibrary-` (3 tasks)	
8a5a63af6e0be406aa6c8c9b6d5f28b2f1b6af5a	-v0f5aece3601a5b4419f7ccec1dbda2071be28ee4
3f7db6bbbcc7c418b3db72d157c6aed1d45b2ccf	-v430f20c722405e462d9ef44dee7d34c41e76fe7a
798a582540019363d14b2090755cc7b89a350788	-v430f20c722405e462d9ef44dee7d34c41e76fe7a
`instance_navidrome__navidrome-` (3 tasks)	
874b17b8f614056df0ef021b5d4f977341084185	—
669c8f4c49a7ef51ac9a53c725097943f67219eb	—
89b12b34bea5687c70e4de2109fd1e7330bb2ba2	—
`instance_NodeBB__NodeBB-` (5 tasks)	
0f788b8eaa4bba3c142d171fd941d015c53b65fc	-v0ec6d6c2baf3cb4797482ce4829bc25cd5716649
b1f9ad5534bb3a44dab5364f659876a4b7fe34c1	-vnan
cfc237c2b79d8c731bbfc6cadf977ed530bfd57a	-v0495b863a912fbff5749c67e860612b91825407c
087e6020e490b4a1759f38c1ad03869511928263	-vf2cf3cbd463b7ad942381f1c6d077626485a1e9e
da0211b1a001d45d73b4c84c6417a4f1b0312575	-vf2cf3cbd463b7ad942381f1c6d077626485a1e9e
`instance_protonmail__webclients-` (5 tasks)	
b9387af4cdf79c2cb2a221dea33d665ef789512e	—
6f8916fbadf1d1f4a26640f53b5cf7f55e8bedb7	—
b387b24147e4b5ec3b482b8719ea72bee001462a	—
f080ffc38e2ad7bddf2e93e5193e82c20c7a11e7	—
cb8cc309c6968b0a2a5fe4288d0ae0a969ff31e1	—
`instance_qutebrowser__qutebrowser-` (9 tasks)	
de4a1c1a2839b5b49c3d4ce21d39de48d24e2091	-v2ef375ac784985212b1805e1d0431dc8f1b3c171
8cd06741bb56cdca49f5cdc0542da97681154315	-v5149fcda2a9a6fe1d35dfed1bade1444a11ef271
c09e1439f145c66ee3af574386e277dd2388d094	-v2ef375ac784985212b1805e1d0431dc8f1b3c171
fcfa069a06ade76d91bac38127f3235c13d78eb1	-v5fc38aaf22415ab0b70567368332beee7955b367
9ed748effa8f3bcd804612d9291da017b514e12f	-v363c8a7e5ccdf6968fc7ab84a2053ac78036691d
70248f256f93ed9b1984494d0a1a919ddd774892	-v2ef375ac784985212b1805e1d0431dc8f1b3c171
ec2dcfce9eee9f808efc17a1b99e227fc4421dea	-v5149fcda2a9a6fe1d35dfed1bade1444a11ef271
cc360cd4a34a126274c7b51f3b63afbaf3e05a02	-v5fc38aaf22415ab0b70567368332beee7955b367
fd6790fe8c02b144ab2464f1fc8ab3d02ce3c476	-v2ef375ac784985212b1805e1d0431dc8f1b3c171
`instance_tutao__tutanota-` (15 tasks)	
befce4b146002b9abc86aa95f4d57581771815ce	-vee878bb72091875e912c52fc32bc60ec3760227b
40e94dee2bcec2b63f362da283123e9df1874cc1	-vc4e41fd0029957297843cb9dec4a25c7c756f029
219bc8f05d7b980e038bc1524cb021bf56397a1b	-vee878bb72091875e912c52fc32bc60ec3760227b
fe240cbf7f0fdd6744ef7bef8cb61676bcdbb621	-vc4e41fd0029957297843cb9dec4a25c7c756f029
8513a9e8114a8b42e64f4348335e0f23efa054c4	-vee878bb72091875e912c52fc32bc60ec3760227b
db90ac26ab78addf72a8efaff3c7acc0fbd6d000	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf
b4934a0f3c34d9d7649e944b183137e8fad3e859	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf
d1aa0ecec288bfc800cfb9133b087c4f81ad8b38	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf
de49d486feef842101506adf040a0f00ded59519	-v10a26bfb45a064b93f4fc044a0254925037b88f1
1e516e989b3c0221f4af6b297d9c0e4c43e4adc3	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf
1ff82aa365763cee2d609c9d19360ad87fdf2ec7	-vc4e41fd0029957297843cb9dec4a25c7c756f029
09c2776c0fce3db5c6e18da92b5a45dce9f013aa	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf
51818218c6ae33de00cbea3a4d30daac8c34142e	-vc4e41fd0029957297843cb9dec4a25c7c756f029
f373ac3808deefce8183dad8d16729839cc330c1	-v2939aa9f4356f0dc9f523ee5ce19d09e08ab979b
fbdb72a2bd39b05131ff905780d9d4a2a074de26	-vbc0d9ba8f0071fbe982809910959a6ff8884dbbf

Table 7: The 50 SWE-bench Verified task IDs. Each ID consists of the prefix in the first column followed by one number from the second column.

ID prefix	Task numbers
`django__django-`	`9296`, `11790`, `11815`, `11848`, `11880`, `11885`, `11951`, `11964`, `11999`, `12039`, `12050`, `12143`, `12155`, `12193`, `12209`, `12262`, `12273`, `12276`, `12304`, `12308`, `12325`, `12406`, `12708`, `12713`, `12774`
`sphinx-doc__sphinx-`	`7454`, `7590`, `7748`, `7757`, `7985`, `8035`, `8056`, `8265`, `8269`, `8475`, `8548`, `8551`, `8638`, `8721`, `9229`, `9230`, `9281`, `9320`, `9367`, `9461`, `9698`, `10449`, `10466`, `10673`, `11510`

Table 8: The 50 Polyglot task IDs, grouped by language. Each ID consists of the prefix in the first column followed by one exercise name from the second column.

ID prefix	Exercise names
`cpp__`	`bank-account`, `diamond`, `gigasecond`, `queen-attack`
`go__`	`connect`, `counter`, `crypto-square`, `error-handling`, `food-chain`, `markdown`, `transpose`, `word-search`
`java__`	`bowling`, `custom-set`, `forth`, `go-counting`, `ledger`, `mazy-mice`, `react`, `rest-api`, `satellite`, `wordy`, `zipper`
`javascript__`	`forth`, `grade-school`, `ocr-numbers`, `queen-attack`, `say`, `simple-linked-list`, `triangle`, `wordy`
`python__`	`dominoes`, `dot-dsl`, `food-chain`, `grade-school`, `poker`, `scale-generator`, `sgf-parsing`, `tree-building`, `wordy`, `zipper`
`rust__`	`accumulate`, `doubly-linked-list`, `gigasecond`, `macros`, `pig-latin`, `poker`, `react`, `say`, `wordy`

Table 9: **Main evolution parameters.**

Method	**Parameter**	**Value**
ELITE	Max generations, T	10
	Exploration width, K	7
	Parallel workers	8
	Exploration timeout	1,800 s
	Integration timeout	1,800 s
DGM	Candidates per generation	2
	Parallel workers	1
	Evaluation parallel workers	8
	Self-modification timeout	1,800 s
HGM	Task-evaluation budget	800
	Parallel workers	20
	Initial expansions	5
	α, β	0.6, 1.0
	Cool-down	Enabled
	Final percentile	1.0
	Self-modification timeout	3,600 s
	Evaluation timeout	3,600 s

the same evaluation count. ELITE instead uses a fixed ten-generation horizon, since it performs no online benchmark evaluation of new descendants. These are method-specific search budgets, not matched compute budgets.

Runtime comparison. All methods run on the same server using the execution settings in Table 9. Search concurrency and task-evaluation concurrency operate at different levels. Under the settings in Table 9, DGM uses one self-modification worker, but evaluates each candidate's benchmark tasks with up to eight parallel workers. HGM follows the official 20-worker configuration, sharing its pool between expansion and evaluation. ELITE uses an eight-worker pool for $K = 7$ same-parent exploratory children, followed by integration; its exploration concurrency is therefore at most seven, even if the pool limit is raised to 20. Increasing candidate counts to occupy additional workers would change search breadth, while identical worker limits alone would not ensure equal active concurrency or resource use. We therefore report practical turnaround for the evaluated configurations and stopping budgets, including search-time task evaluation where applicable. The time ratios remain configuration-dependent; task-evaluation counts and billed API expenditure provide complementary measures of evolution cost.

Repetitions and evaluation budget. We report three evolution runs per backbone for ELITE and one for each baseline. Repeating an evaluation-driven baseline requires rerunning the entire search, including approximately 800 task evaluations, rather than simply reevaluating its final agent. Matching three runs for both baselines across both backbones would therefore require eight additional searches, or roughly 6,400 additional search-time task evaluations, followed by final assessments. Each task evaluation can involve multiple LLM calls, repository interactions, and test execution, incurring both runtime and API expenditure. Post-hoc analysis also requires substantial evaluation beyond the main results: The accepted-lineage analysis in Figure 3(b) likewise requires full 60-task evaluations of intermediate checkpoints. Evaluating every integrated candidate over $T = 10$ rounds would require at most $60 \times 10 = 600$ rollouts for one backbone on this benchmark. The first-step comparison covers eight agents on 60 tasks (480 agent–task outcomes), while the five-step diagnostic covers 150 outcomes. These analyses examine intermediate and exploratory agents whose task performance is not observed during ELITE evolution. Within the available time and API budget, we therefore prioritize repetitions of the main ELITE setting while retaining cross-backbone comparisons, ablations, and detailed outcome and code analyses. Table 5 reports the individual main-run results and sample standard deviations.

Algorithm 1 ELITE self-improvement

Require: initial agent $\mathcal{A}_0$, generations T, exploration width K

for $t = 0, \ldots, T-1$ **do**
  collect compact evolution history m_t
  $\mathcal{H}_t \leftarrow \mathcal{P}(\mathcal{A}_t, m_t; K)$
  $\mathcal{V}_t \leftarrow \emptyset$
  for $k = 1, \ldots, K$ **in parallel do**
    $\Delta_t^{(k)} \leftarrow \mathcal{E}(\mathcal{A}_t, h_t^{(k)})$
    $\mathcal{A}_t^{(k)} \leftarrow \mathrm{Apply}(\mathcal{A}_t, \Delta_t^{(k)})$
    if $G_{\mathrm{exp}}(\mathcal{A}_t^{(k)}) = 1$ **then**
      $\mathcal{V}_t \leftarrow \mathcal{V}_t \cup \{\mathcal{A}_t^{(k)}\}$
  if $\mathcal{V}_t = \emptyset$ **then**
    $\mathcal{A}_{t+1} \leftarrow \mathcal{A}_t$
    continue
  construct compact evidence records for children in $\mathcal{V}_t$
  $\Delta_t^{\mathrm{int}} \leftarrow \mathcal{I}(\mathcal{A}_t, \mathcal{V}_t)$
  $\mathcal{A}_t^{\mathrm{int}} \leftarrow \mathrm{Apply}(\mathcal{A}_t, \Delta_t^{\mathrm{int}})$
  if $G_{\mathrm{acc}}(\mathcal{A}_t^{\mathrm{int}}) = 1$ **then**
    $\mathcal{A}_{t+1} \leftarrow \mathcal{A}_t^{\mathrm{int}}$
  else
    $\mathcal{A}_{t+1} \leftarrow \mathcal{A}_t$
return $\mathcal{A}_T$ ▷ *No downstream task evaluation enters the self-improvement loop*

A.6 ELITE ALGORITHM

Algorithm 1 summarizes the complete ELITE self-improvement procedure. At each generation, valid exploratory children are integrated into a candidate descendant; if no child passes the exploration gate or the integrated descendant fails the acceptance gate, the current agent is retained for the next generation.