MENDEL GÖDEL MACHINE: RECURSIVE SELF-IMPROVING CODING AGENTS VIA COMPARATIVE EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-improving coding agents that iteratively rewrite their own source code have demonstrated impressive performance on coding tasks. However, existing solutions generally derive self-modification from a single failure trajectory at a time, overlooking rich comparative signals available in the agent's expanding archive of past attempts. According to Mendelian principles of controlled inheritance, we introduce Mendel Gödel Machine (MGM). In addition to the general single-trajectory *clonal mutation*, MGM includes two new types of self-modification that better utilizes evidences accumulated: the *reaction-norm mutation* edits an agent based on its trajectories on multiple tasks simultaneously, and the *cross-lineage hybridization* edits an agent using the trajectory of a reference agent from another lineage on the same task. Under an additive fitness landscape model, we prove theoretically and demonstrate via controlled surrogate simulation that the new strategies facilitate a faster and better convergence over single-trajectory baselines. Experiments on SWE-bench and Polyglot confirm MGM's consistent improvement in performance, efficiency, and generalizability. Our code is publicly available at `https://anonymous.4open.science/r/MendelGM/`

1 INTRODUCTION

The vision of artificial intelligence recursively rewriting itself to become better traces back several decades (Schmidhuber, 1987). Gödel Machine (Schmidhuber, 2003; 2007) conceives a mathematically rigorous, self-referential modification process when a provable (measurable) benefit can be derived. Recent advances in large language models (LLMs) and coding agents have started to empirically realize such an idea (Yang et al., 2024; Hu et al., 2025; Gao et al., 2026). Robeyns et al. (2025) show that an agent equipped with basic file-editing tools can autonomously refactor its own codebase and lift its performance. Zhang et al. (2026a) reframe this loop as open-ended evolution, maintaining an expanding archive of agent variants, from which at each iteration one is sampled to seed the next self-modification. Wang et al. (2026) improve the sampling policy by using the aggregated performance of agents' descendants as signals guiding evaluation and expansion operations.

However, progress so far has focused on optimizing the archive that stores generated agents and evaluated traces, and on the process that samples which agent to evaluate or edit next. The *self-modification* process, during which the agent edits its own source code, has remained essentially underexplored, with each self-modification step conditioned only on one agent's single trajectory (typically a recent failure) on a single task. The expanding archive, which records all agent variants ever generated together with their behavior on all evaluated tasks, has been only used as a leaderboard for sampling, overlooking rich comparative evidence that can facilitate better-informed edits.

We identify two types of such comparative signals. First, when an agent is evaluated across multiple tasks, the pattern of its successes and failures forms a reaction norm (Woltereck, 1909; Pigliucci, 2001): a stable, genotype-specific profile of how performance varies across environments. Recurring failure modes can more plausibly distinct genotype-level defects from task-specific accidents. Second,

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.

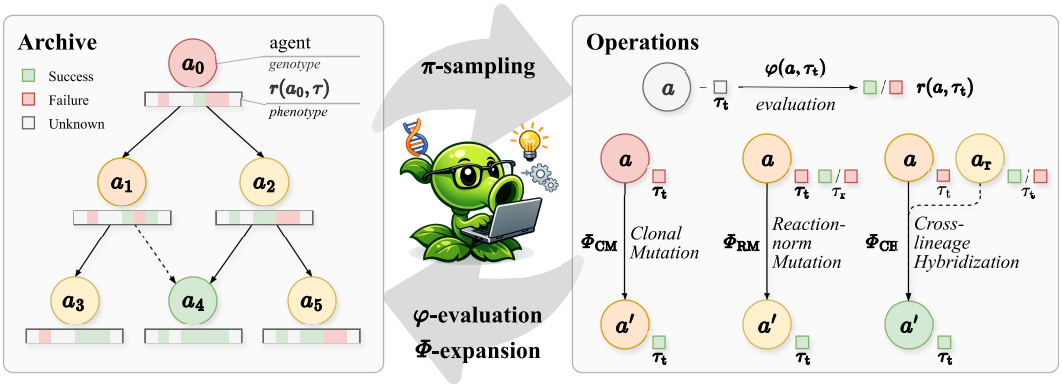


Figure 1: **Mendel Gödel Machine**. MGM organizes self-modification via controlled inheritance based on evidence across tasks and lineages. The archive maintains a lineage tree of agent variants; each stores its source code as genotype and evaluation outcomes as phenotype. Each iteration applies π-sampling that selects an operation to perform with the necessary resources from the archive. φ-evaluation executes the agent on untested tasks and records its trajectory and results. Clonal mutation Φ_{CM} edits the agent based on a single failure trajectory on target task τ_{t}. Reaction-norm mutation Φ_{RM} uses the agent's trajectories across reference tasks τ_{r}. Cross-lineage hybridization Φ_{CH} uses a reference agent a_{r} from a different lineage that attempted the same task.

when multiple agents have attempted the same task, their trajectories reveal transferable behavioral traits. Conditioning edits on these contrastive evidence enables targeted ability transfer across lineages and reduce redundant exploration. Building on these observations, we introduce **Mendel Gödel Machine (MGM)**, which isolates heritable effects through controlled comparisons as in Mendelian genetics. As shown in Figure 1, we structure self-modification into three operators: *clonal mutation* denotes standard single-agent, single-trajectory self-modification; *reaction-norm mutation* edits an agent conditioned on its trajectories across multiple tasks; and *cross-lineage hybridization* edits an agent using a reference agent's trajectory on the same task. All strategies directly operate on trajectories accumulated during routine archive evaluation and therefore incur no extra task evaluations.

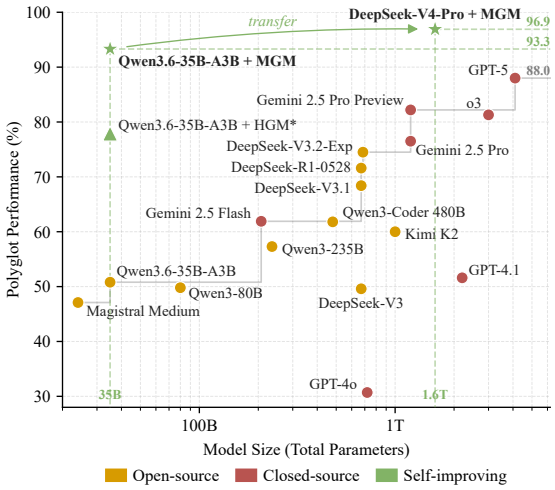


Figure 2: **Polyglot performance.** Result marked with asterisk is from Polyglot-60; all other scores are from the complete Polyglot-225[1]. Closed-source model sizes follows estimates reported by Li (2026).

To isolate the contribution of each dimension rigorously, we develop a formal additive fitness landscape model in which each agent is represented as a binary vector in genotype space, and self-improvement progress is measured as the reduction in Hamming distance to an oracle genotype. We demonstrate that MGM yields a faster expected convergence than single-trajectory baselines, and we validate this in controlled Monte Carlo surrogate simulations.

Experiments on Polyglot (Gauthier, 2024) and SWE-bench (Jimenez et al., 2024) confirm MGM's consistent gains over baselines in both performance and efficiency. Purely through evolving the agent scaffold, MGM advances Qwen3.6-35B-A3B on Polyglot from 50.8% to 93.3%, surpassing closed-source GPT-5 (Singh et al., 2026) with $\sim$117$\times$ fewer parameters, as Figure 2 shows. MGM also exhibits stronger generalizability across both unseen benchmarks and backbone LLMs. Notably, transferring the Qwen-evolved scaffold to DeepSeek-V4-Pro yields 96.9% on Polyglot. These findings demonstrate that MGM's richer comparative conditioning discovers reusable, workflow-level improvements that support strong coding-agent performance, indicating its potential as a scalable approach for future self-improving agent development.

[1]Sourced from Aider-Polyglot Leaderboard `https://llm-stats.com/benchmarks/aider-polyglot`.

2 PRELIMINARIES

We denote an executable coding agent $a \in \mathcal{A}$, together with its auxiliary scaffolding, as the genotype that self-improvement seeks to evolve. Running a on a task $\tau \sim \mathcal{D}$ yields an evaluation trajectory $\varphi(a, \tau)$ and a binary outcome $r(a, \tau) \in \{0, 1\}$, which is viewed as its phenotype. The expected utility of a is

$$U(a) = \mathbb{E}_{\tau \sim \mathcal{D}}[r(a, \tau)]. \tag{1}$$

A self-modification process Φ lets the parent agent edit its own code using diagnostic evidence E, a finite set of stored trajectory–outcome pairs:

$$a' \leftarrow \Phi(a, E), \qquad E \subseteq \{(\varphi(a, \tau), r(a, \tau))\}. \tag{2}$$

DGM (Zhang et al., 2026a) introduces an archive-based self-improvement framework by maintaining an expanding tree of generated agents and their evaluation trajectories. HGM (Wang et al., 2026) further formulates it as a fixed-budget tree-search problem. Let $\mathcal{G}_t$ be the archive tree at step t and $\mathcal{V}_t$ its node set. For node $a_i \in \mathcal{V}_t$, let S_i be its evaluated tasks, $F_i \subseteq S_i$ its failed tasks, and

$$n_{\mathrm{s}}(a_i) = |S_i| - |F_i|, \qquad n_{\mathrm{f}}(a_i) = |F_i|. \tag{3}$$

At each step t, HGM chooses to allocate either an evaluation or an expansion to its archive. An expansion happens when

$$N_t^{\alpha} \geq |\mathcal{V}_t|, \qquad N_t = \sum_{a \in \mathcal{V}_t} (n_{\mathrm{s}}(a) + n_{\mathrm{f}}(a)), \tag{4}$$

where $\alpha \in [0, 1]$ is a widening parameter, and otherwise another task evaluation is performed.

The evaluation policy samples a using Thompson sampling from the node-level posterior

$$\pi_a \sim \mathrm{Beta}(\kappa(1 + n_{\mathrm{s}}(a)), \kappa(1 + n_{\mathrm{f}}(a))), \tag{5}$$

where $\kappa > 0$ is the concentration parameter controlling the exploration–exploitation trade-off. The expansion policy samples a to expand using clade-level evidence. Let $C_t(a)$ denote the subtree rooted at a, and define

$$n_{\mathrm{s}}^{C}(a) = \sum_{a' \in C_t(a)} n_{\mathrm{s}}(a'), \qquad n_{\mathrm{f}}^{C}(a) = \sum_{a' \in C_t(a)} n_{\mathrm{f}}(a'). \tag{6}$$

HGM samples expansion candidates from

$$\pi_a^{C} \sim \mathrm{Beta}(\kappa(1 + n_{\mathrm{s}}^{C}(a)), \kappa(1 + n_{\mathrm{f}}^{C}(a))). \tag{7}$$

3 MENDEL GÖDEL MACHINE

MGM builds on the aforementioned tree-search framework of selection, evaluation, and expansion policies. Instead of relying on single-trajectory self-modification, MGM partitions the expansion operator Φ into three specialized sub-operators based on the type of diagnostic evidence E available in the archive: clonal mutation Φ_{CM}, reaction-norm mutation Φ_{RM}, and cross-lineage hybridization Φ_{CH}, as Figure 1 shows.

3.1 MENDELIAN SELF-MODIFICATION OPERATORS

Analogous to Mendelian genetics, which seeks to isolate heritable effects through controlled comparisons, we design self-modification operators as diagnostic processes that ask the selected agent to make general improvements to its genotype based on different phenotype evidence.

Clonal Mutation. Clonal mutation is the standard single-agent, single-trajectory self-improvement operator. It is used when MGM has only one informative failure or cannot construct a reliable comparison. Given a selected node i and a failed task $\tau \in F_i$, the evidence is

$$E_{\mathrm{CM}}(i, \tau) = \{(\varphi(a_i, \tau), r(a_i, \tau))\}. \tag{8}$$

The editor diagnoses the failure and modifies a_i to avoid similar failures in future tasks:

$$a' \leftarrow \Phi_{\mathrm{CM}}(a_i, E_{\mathrm{CM}}). \tag{9}$$

This operator preserves the behavior of HGM-style self-modification and ensures that the search can proceed even when the archive is still small.

Reaction-norm Mutation. A reaction norm describes how one genotype expresses different phenotypes under different environments (Woltereck, 1909; Pigliucci, 2001). MGM incorporates this idea and designs reaction-norm mutation Φ_{RM} by comparing multiple phenotypes of the same genotype across different tasks. If the same agent fails, or behaves inconsistently, across multiple tasks, the resulting pattern can provide richer diagnostic evidence of the agent's genuine weakness, rather than accidental task-specific errors.

Formally, Φ_{RM} becomes available for a_i when it has accumulated enough $\varphi(a_i, \tau)$ trajectories

$$|S_i| \geq m_{\mathrm{RM}}, \tag{10}$$

and there exist at least two trajectories, with a failed one as the target $\tau_{\mathrm{t}} \in S_i$. The reference τ_{r} may be any other trajectory in S_i, preferably a failed one if available. The evidence

$$E_{\mathrm{RM}}(a_i, \tau_{\mathrm{t}}, \tau_{\mathrm{r}}) = \{\big(\varphi(a_i, \tau_{\mathrm{t}}), r(a_i, \tau_{\mathrm{t}})\big), \big(\varphi(a_i, \tau_{\mathrm{r}}), r(a_i, \tau_{\mathrm{r}})\big)\}. \tag{11}$$

is given to the self-modification process, where the agent is asked to identify a recurring or contrastive behavioral pattern shared by the provided trajectories and to implement a general improvement

$$a' \leftarrow \Phi_{\mathrm{RM}}(a_i, E_{\mathrm{RM}}). \tag{12}$$

Cross-lineage Hybridization. Cross-lineage hybridization compares different genotypes under the same task environment. It is available when two nodes have attempted at least one common target task, and the task is not already solved by both:

$$\exists j \neq i,\ \exists \tau_{\mathrm{t}} \in S_i \cap S_j \qquad \text{s.t.} \quad \neg\big(r(a_i, \tau_{\mathrm{t}}) = 1 \wedge r(a_j, \tau_{\mathrm{t}}) = 1\big). \tag{13}$$

MGM designates one failed agent as the target a_{t} to improve. If the reference agent a_{r} fails likewise, MGM uses the comparison to identify complementary failure modes; if a_{r} solved τ_{t}, differences in their genotypes are used as corrective signals to guide a_{t}'s self-modification. The evidence is

$$E_{\mathrm{CH}}(a_{\mathrm{t}}, a_{\mathrm{r}}, \tau_{\mathrm{t}}) = \{(\varphi(a_{\mathrm{t}}, \tau_{\mathrm{t}}), r(a_{\mathrm{t}}, \tau_{\mathrm{t}})), (\varphi(a_{\mathrm{r}}, \tau_{\mathrm{t}}), r(a_{\mathrm{r}}, \tau_{\mathrm{t}}))\}. \tag{14}$$

where a_{t} is the target agent and a_{r} is the reference agent. The child is attached to the primary lineage:

$$a' \leftarrow \Phi_{\mathrm{CH}}(a_{\mathrm{t}}, E_{\mathrm{CH}}). \tag{15}$$

This hybridization operation is a diagnostic process. MGM does not splice source files from one agent into another. Instead, it asks the target agent itself to extract a transferable behavioral trait from the reference trajectory and adapt that trait to the target agent's own codebase. This encourages the failing lineage to derive and inherit a genuine improvement rather than task-specific behaviors.

3.2 Sampling Tasks and Operators

MGM additionally maintains a global pool

$$\mathcal{P}_t = \bigcup_{i \in \mathcal{V}_t} F_i, \tag{16}$$

which stores tasks that have exposed failures in any previously evaluated agent. This pool is not a separate evaluation benchmark and does not introduce extra φ-evaluations. It is maintained to control how future evaluation tasks are sampled. When selecting a new task for an agent, MGM samples from tasks not yet attempted by that agent, but assigns a predetermined weight to tasks in $\mathcal{P}_t$:

$$w_i(\tau) = \begin{cases} \beta_{\mathrm{fail}}, & \tau \in \mathcal{P}_t, \\ 1, & \tau \notin \mathcal{P}_t, \end{cases} \qquad \tau \notin S_i, \tag{17}$$

where β_{fail} is the failed-pool boost. This task sampling design encourages evaluation on tasks that are known to reveal weaknesses in at least one lineage, increasing the diagnostic value of each φ-evaluation. It also deliberately creates overlap across lineages. Since Φ_{CH} requires agents to have attempted a shared task, the pool increases the availability of controlled cross-lineage comparisons without spending additional evaluation budget, making transferable behavioral traits easier to discover.

For tree search framework, MGM inherits HGM, adopting UCB-AIR, which at each step chooses between initiating a φ-evaluation for an existing node or a Φ-expansion of the evolution tree, while Thompson Sampling utilizes the clade-level and node-level Beta posteriors defined in Section 2. The

difference lies in how an expansion is executed. MGM partitions the single Φ operator that HGM and DGM use (Φ_{CM}) into three sub-operators $\{\Phi_{\mathrm{CM}}, \Phi_{\mathrm{RM}}, \Phi_{\mathrm{CH}}\}$. When π selects a Φ-expansion for parent a_i, MGM first constructs the set of eligible operators $\Omega_i \subseteq \{\Phi_{\mathrm{CM}}, \Phi_{\mathrm{RM}}, \Phi_{\mathrm{CH}}\}$ from the archive. Φ_{CM} is eligible whenever the selected agent has at least one failed task, i.e., $F_i \neq \varnothing$. Φ_{RM} becomes eligible if the agent has been evaluated on at least m_{RM} distinct tasks (Equation 10), and there exist trajectories for two distinct tasks $\tau_{\mathrm{t}}, \tau_{\mathrm{r}} \in S_i$ with at least one failure. Φ_{CH} becomes eligible if agents from two lineages have attempted a shared task τ_{t} that is not solved by both (Equation 13).

MGM then samples among eligible operators with configurable weights $\lambda_{\mathrm{CM}}, \lambda_{\mathrm{RM}}, \lambda_{\mathrm{CH}}$:

$$\Pr(\sigma \mid i) = \frac{\lambda_\sigma}{\sum_{\sigma' \in \Omega_i} \lambda_{\sigma'}}, \qquad \sigma \in \Omega_i. \tag{18}$$

The selected operator determines the evidence E_σ, and the child is produced by

$$a' \leftarrow \Phi_\sigma(a_{\mathrm{t}}, E_\sigma). \tag{19}$$

For Φ_{CM} and Φ_{RM}, the target agent is the selected parent $a_{\mathrm{t}} = a_i$. For Φ_{CH}, if exactly one agent solves the shared task, the failing agent is edited using the successful agent as reference; if both fail, the higher-utility lineage serves as the primary target. If $\Omega_i = \varnothing$, MGM skips the expansion and π allocates another φ-evaluation instead.

The complete pseudocode of MGM is provided in Appendix C.1.

4 Simulations

To isolate why comparative evidence can improve self-modification, we study a controlled additive fitness landscape in which an agent contains latent scaffold-level defects and each task exposes only a subset of them. A failed trajectory therefore identifies a noisy set of plausible causes rather than the exact defect. The complete surrogate formulation is given in Appendix D.

Given evidence E, let $C_\sigma(E)$ denote the candidate defects induced by self-modification operator $\sigma \in \{\mathrm{CM}, \mathrm{RM}, \mathrm{CH}\}$, and let $M(a)$ denote the true defects of agent a. If an actually incorrect component is repaired with probability s, the effective fix probability is

$$p_f^\sigma = s \cdot \Pr_{\ell \sim C_\sigma(E)}\big[\ell \in M(a)\big]. \tag{20}$$

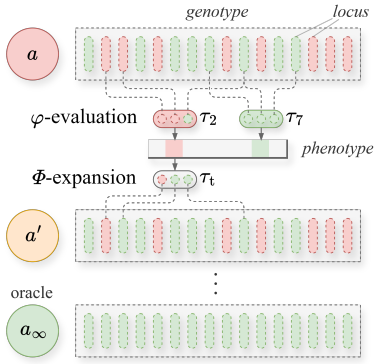


Figure 3: **Additive fitness landscape model.** Each agent has a latent binary genotype. Task-specific ϕ-evaluations reveal partial phenotype evidence, which guides Φ expansion toward an all-correct oracle genotype.

Clonal mutation conditions on a single failed trajectory and therefore retains all task-relevant explanations. Reaction-norm mutation compares trajectories of the same agent across tasks, while cross-lineage hybridization compares different agents on the same task. Under informative comparisons, both reduce irrelevant explanations and increase the density of true defects in $C_\sigma(E)$. This yields:

Under sound comparative evidence,

$$p_f^{\mathrm{RM}} > p_f^{\mathrm{CM}}, \qquad p_f^{\mathrm{CH}} > p_f^{\mathrm{CM}}. \tag{21}$$

The full derivation and assumptions are provided in Appendix D.2. Intuitively, MGM improves self-modification by reducing diagnostic uncertainty rather than by assuming a stronger editor.

We instantiate this advantage by

$$\rho = \frac{p_f^{\mathrm{RM}}}{p_f^{\mathrm{CM}}} = \frac{p_f^{\mathrm{CH}}}{p_f^{\mathrm{CM}}}, \tag{22}$$

where $\rho = 1$ is the null setting and $\rho > 1$ represents increasingly informative comparative evidence. DGM, HGM, and MGM are evaluated under the same total budget and equal evaluation/edit costs; MGM differs from HGM only through access to RM and CH. Full simulation parameters are provided in Table 7.

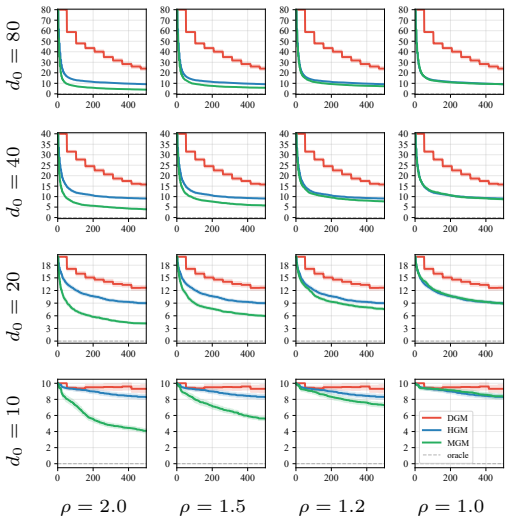


Figure 4: **Simulated performance evolution over cumulative budget spent.** Results compared across task difficulty and edit effectiveness. Rows vary the initial edit distance d_0; columns vary the fix-probability advantage ratio ρ. Each panel shows the edit distance results (lower is better) averaged across random seeds with 95 % CIs. Dashed lines mark oracle optima.

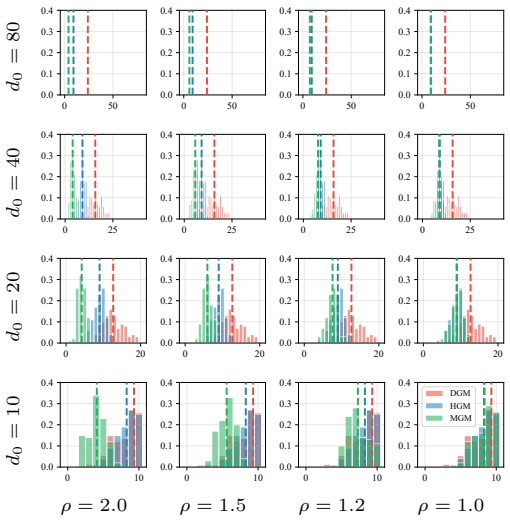


Figure 5: **Simulated final performance distribution.** Results compared across task difficulty and edit effectiveness. Rows vary the initial edit distance d_0; columns vary the fix-probability advantage ratio ρ. Each panel shows the distribution of final edit distance results (lower is better) across all random seeds, with dashed lines marking per-method means.

Figures 4 and 5 sweep both initial difficulty d_0 and diagnostic advantage ρ. MGM consistently converges faster and reaches lower final edit distance for $\rho > 1$, while the advantage largely disappears at $\rho = 1$. The improvement increases with ρ, supporting the mechanism predicted above: better comparative diagnosis translates into more efficient archive-level self-improvement.

5 Experiments

We evaluate MGM on challenging software-engineering benchmarks to answer three research questions: Does MGM exhibit better performance and efficiency? Can agents evolved by MGM generalize to other challenging tasks or models? What is the contribution of each key component in MGM?

Our experiments span several challenging and representative benchmarks, including SWE-bench Verified (Jimenez et al., 2024), SWE-bench Pro (Deng et al., 2025), SWE-bench Multilingual (Khandpur & the SWE-bench Team, 2025; Yang et al., 2025), and Polyglot (Gauthier, 2024). In all experiments, agents are not given access to private test cases or test results during evaluation. To ensure fair comparisons and control computational cost, unless otherwise specified, all results reported in this section are evaluated on the same two subsets of SWE-bench Verified and Polyglot as Wang et al. (2026) and Zhang et al. (2026a).

5.1 Performance

To assess whether MGM exhibits greater self-improvement potential than HGM, following the setting of Zhang et al. (2026a); Wang et al. (2026), we evaluate both methods on SWE-bench Verified and Polyglot under an identical computational budget of 200 evaluations. We adopt Qwen3.6-35B-A3B (Qwen Team, 2026) as the backbone LLM. To ensure a fair comparison, both methods start from the same ancestor agent. We report the accuracy of the best-belief agent evolved by each method.

As shown in Table 1, MGM consistently outperforms HGM. On SWE-bench Verified, both methods start from the same initial accuracy of 68.3%, which HGM improves to 73.3%, and MGM improves to 78.3%. On Polyglot, starting from the same 50.8% agent, HGM improves to 77.9%, while MGM reaches 93.2%. These result shows that MGM's comparative self-modification operators are effective

	SWE-bench Verified-60			Polyglot Subset			Avg.		
Agent	Initial	HGM	MGM	Initial	HGM	MGM	Initial	HGM	MGM
Accuracy	68.3	$73.3^{+5.0}$	$\mathbf{78.3^{+10.0}}$	50.8	$77.9^{+27.1}$	$\mathbf{93.2^{+42.4}}$	59.6	$75.6^{+16.0}$	$\mathbf{85.8^{+26.2}}$
% Impr.	–	↑7.3%	**↑14.6%**	–	↑53.3%	**↑83.5%**	–	↑26.8%	**↑44.0%**
Time	–	93.02 h	96.11 h	–	44.20 h	**40.14 h**	–	68.61 h	**68.12 h**

Table 1: **Performance of coding agents evolved on SWE-bench Verified and Polyglot.** Results evolved using Qwen3.6-35B-A3B after 200 φ-evaluations and 24 Φ-expansions. For each benchmark, HGM and MGM start from the same initial scaffold. Superscripts in accuracy denote absolute percentage-point improvements over corresponding initial agents. Time reported as CPU wall-clock time with 8×NVIDIA H100 GPUs.

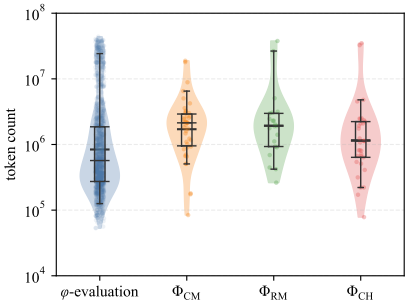


Figure 6: **Token costs of each operators for HGM and MGM evolved on Polyglot.** The distribution of token costs for all evaluation and expansion operations lies in similar order of magnitude.

on both standalone coding tasks as in Polyglot, and also repository-level software-engineering tasks, where failures often involve incompetency in localization, environment understanding, and repair workflows. Since both methods use exactly the same number of evaluation and expansion operations, this performance gap cannot be attributed to a larger search budget. Instead, it supports our central hypothesis that reusing archived trajectories through reaction-norm mutation and cross-lineage hybridization provides more informative diagnostic evidence than conditioning each self-modification step on a single failure trajectory. Evaluation on the full 225-task Polyglot benchmark (Figure 2 and Appendix G.1) further confirms the same results.

We further analyze the token consumption of each step in the experiments. Figure 6 shows that all evolutionary strategies used by HGM and MGM have comparable average token costs, indicating that the observed performance gains are not achieved simply by increased token expenditure.

5.2 Generalization

A key question for self-improving coding agents is whether the evolved improvements remain useful beyond the exact setting. Since MGM modifies the agent scaffold rather than the parameters of the underlying LLM, a successful evolution method should ideally discover reusable workflow-level improvements that transfer across both benchmarks and backbone models (Hu et al., 2025). We therefore evaluate generalization from two complementary perspectives: cross-benchmark generalization and cross-model scaffold transfer.

Cross-benchmark generalization. We first evaluate whether scaffolds evolved on Polyglot-60 can transfer to different software-engineering settings without further self-improvement. After evolution, we freeze the evolved scaffolds and evaluate them on subsets of SWE-bench Pro and SWE-bench Multilingual (Khandpur & the SWE-bench Team, 2025; Yang et al., 2025). We report the tasks used in Appendix F. We choose these two benchmarks because they test complementary forms of out-of-distribution generalization. SWE-bench Pro provides a more challenging repository-level software-engineering setting, where solving a task often requires understanding larger codebases, localizing the relevant files, and producing robust patches. SWE-bench Multilingual further stresses the language generality of the evolved scaffold by covering repositories implemented in diverse programming languages.

As shown in Table 2, the initial scaffold obtains 16.7% accuracy on SWE-bench Pro and 41.7% on SWE-bench Multilingual. The HGM-evolved scaffold shows limited transfer: it decreases to 13.3% on SWE-bench Pro, corresponding to a -3.4% change, and slightly improves to 43.3% on SWE-bench Multilingual, corresponding to a $+1.6\%$ gain. In contrast, MGM achieves positive transfer on both target benchmarks. It reaches 26.7% on SWE-bench Pro, giving a $+10.0\%$ improvement over the initial scaffold, and 55.0% on SWE-bench Multilingual, giving a $+13.3\%$ improvement.

	SWE-bench Pro			SWE-bench Multilingual			Avg.		
Agent	Initial	HGM	MGM	Initial	HGM	MGM	Initial	HGM	MGM
Accuracy	16.7	13.3$^{-3.4}$	**26.7$^{+10.0}$**	41.7	43.3$^{+1.6}$	**55.0$^{+13.3}$**	29.2	28.3$^{-0.9}$	**40.9$^{+11.7}$**
% Impr.	–	↓20.4%	↑**59.9%**	–	↑3.8%	↑**31.9%**	–	↓3.1%	↑**40.1%**

Table 2: **Cross-benchmark generalization from Polyglot to SWE-bench Pro and SWE-bench Multilingual.** All agents evolved on Polyglot are evaluated zero-shot on held-out SWE-bench variants, using Qwen3.6-35B-A3B. Superscripts in accuracy denote absolute percentage-point improvements over corresponding initial agents.

	Evolved			*Transferred*								
	Qwen3.6-35B-A3B			DeepSeek-V4-Flash			DeepSeek-V4-Pro			Avg.		
Agent	Initial	HGM	MGM	Initial	HGM	MGM	Initial	HGM	MGM	Initial	HGM	MGM
Acc.	68.3	73.3$^{+5.0}$	**78.3$^{+10.0}$**	50.0	60.0$^{+10.0}$	**66.7$^{+16.7}$**	45.0	70.0$^{+25.0}$	**75.0$^{+30.0}$**	47.5	65.0$^{+17.5}$	**70.8$^{+23.3}$**
% Impr.	–	↑7.3%	↑**14.6%**	–	↑20.0%	↑**33.3%**	–	↑55.6%	↑**66.7%**	–	↑36.8%	↑**49.1%**

Table 3: **Cross-model transfer on SWE-bench Verified-60.** The Qwen3.6-35B-A3B block reports the original evolved agents, while the DeepSeek blocks report transferred performance by using the scaffolds evolved on Qwen3.6-35B-A3B and evaluating with LLM backbone replaced. Superscripts in accuracy denote absolute percentage-point improvements over corresponding initial agents.

These results suggest that MGM's comparative self-modification operators help discover scaffold changes that remain useful beyond the source benchmark. The gains on both SWE-bench Pro and SWE-bench Multilingual indicate that MGM does not overfit to Polyglot-specific failure patterns and is more likely to acquire reusable software-engineering workflows that transfer across benchmark distributions and programming-language settings.

Cross-model generalization. We further evaluate whether the evolved scaffolds remain effective when paired with different backbone LLMs. In this experiment, the scaffolds are evolved on SWE-bench Verified-60 using Qwen3.6-35B-A3B. We then freeze the evolved scaffold and replace only the inference backbone with DeepSeek-V4-Flash and DeepSeek-V4-Pro (DeepSeek-AI et al., 2026). This protocol isolates whether the improvement is encoded in the scaffold itself, rather than being specific to the original Qwen3.6 backbone.

Table 3 shows that both HGM and MGM produce scaffolds that transfer across models, but MGM transfers more strongly and consistently. Under the original Qwen3.6-35B-A3B backbone, HGM improves the initial scaffold from 68.3% to 73.3%, whereas MGM improves it to 78.3%, giving MGM a +5.0% advantage over HGM. After transferring the same scaffolds to DeepSeek-V4-Flash, HGM reaches 60.0%, while MGM further improves to 66.7%. On DeepSeek-V4-Pro, HGM reaches 70.0%, whereas MGM reaches 75.0%. Averaged over the two transferred DeepSeek backbones, MGM achieves 70.8% accuracy, compared with 65.0% for HGM and 47.5% for the initial scaffold. The result confirms that MGM does not simply tune the scaffold to the quirks of a single foundation model. Instead, its evolved changes remain beneficial when the same scaffold is executed by substantially different inference backbones.

Beyond the subset-level transfer results above, we further evaluate whether the scaffold evolved on Polyglot with Qwen3.6-35B-A3B remains effective when paired with a stronger inference backbone. Specifically, we freeze the MGM-evolved Polyglot scaffold, replace only the backbone model with DeepSeek-V4-Pro, and evaluate it on the complete 225-task Polyglot benchmark. As Figure 2 shows, this transferred configuration achieves ***96.89%*** accuracy, further advancing the frontier result achieved with Qwen above. This result provides additional evidence that MGM truely discovers reusable workflow-level improvements that can be further amplified by stronger foundation models.

Together, the cross-benchmark and cross-model results demonstrate that MGM helps improve not only in-domain performance, but also the generalizability of the evolved agent scaffold across benchmarks and models. This points to a potential path for scalable self-improving agent development, where scaffolds evolved on smaller datasets and cheaper backbones can be reused on stronger models. Appendices A.1 and A.3 further provide qualitative analysis demonstrating how these gains actually correspond to reusable workflow-level skills.

5.3 Ablation Study

Agent	Polyglot-60 Initial	MGM	w/o Φ_{RM}	w/o Φ_{CH}
Accuracy	50.8	**93.2**$^{+42.4}$	79.7$^{+28.9}$	74.6$^{+23.8}$
% Impr.	–	↑**83.5%**	↑56.9%	↑46.9%
Time	–	40.14 h	43.01 h	39.95 h

Table 4: **Ablation study.** Superscripts in accuracy denote absolute percentage-point improvements over corresponding initial agents. Time reported as CPU wall-clock time with 8×NVIDIA H100 GPUs.

To understand the contribution of each component in MGM, we conduct an ablation study on Polyglot-60 with the same budget of 200 evaluations. We use Qwen3.6-35B-A3B as the backbone LLM for all settings. Each variant starts from the same initial ancestor agent. We compare the full MGM with two ablated variants: MGM without Φ_{RM} and without Φ_{CH}. Appendix G.2 visualizes the corresponding evolution trees for the full model, HGM, and ablations. As shown in Table 4, the full MGM achieves the best performance, and removing Φ_{RM} and Φ_{CH} leads to a clear performance degradation. This suggests that Φ_{RM} plays an important role in guiding the self-improvement process toward more promising agents, and that Φ_{CH} is even more critical for effective self-improvement, likely because it helps preserve and reuse useful evolutionary information across iterations. The ablated variants have comparable training hours, which also confirms that the gains of MGM come from the combined effect of its key components rather than from differences in computational cost.

6 Related Work

LLM agent systems and automated agent-design methods study how prompts, tools, and workflows can be optimized as executable scaffolds (Yao et al., 2023; Schick et al., 2023; Wu et al., 2023; Yang et al., 2024; Zhang et al., 2026b; Gao et al., 2026). Hu et al. (2025) and Hong et al. (2024) show that meta-agents can iteratively propose new agent designs from an archive of prior discoveries, yielding agents that transfer across tasks and models. This line of work establishes agent scaffolding as a searchable program space, with a focus on discovering new agent designs.

Self-improving coding agents instantiate this idea in an inherited self-modification setting (Madaan et al., 2023; Shinn et al., 2023; Xia et al., 2025). Robeyns et al. (2025) show that a coding agent can evaluate itself, edit its own codebase, and improve over iterations. Zhang et al. (2026a) extend this process into open-ended evolution by maintaining an archive of self-modified agents, while Wang et al. (2026) improve archive expansion by estimating the future self-improvement potential of clades. MGM follows this archive-based self-improvement setting, but targets a complementary bottleneck: improving the evidence used for each expansion. By reusing existing trajectories across tasks and lineages, MGM improves each self-modification step without requiring additional evaluations.

A detailed discussion of additional related work on self-evolving agents, runtime adaptation methods, and datasets is provided in Appendix B.

7 Conclusion

We introduce the Mendel Gödel Machine (MGM), an archive-based self-improving coding-agent framework that conditions self-modification on comparative evidence already present in evaluation trajectories. Beyond clonal mutation, MGM adds reaction-norm mutation and cross-lineage hybridization, which reuse archived phenotypes across tasks and lineages without additional task evaluations. Under an additive fitness landscape, theory and controlled simulations show that richer diagnostic evidence can raise effective fix probability and accelerate convergence relative to single-trajectory baselines. Experiments on SWE-bench and Polyglot confirm MGM's consistent gains over HGM in performance and efficiency under a matched budget. Ablations show that both comparative operators contribute to these gains. Held-out evaluations further suggest that the evolved scaffolds transfer across benchmarks and backbone LLMs, indicating that richer comparative conditioning can discover reusable workflow-level improvements rather than narrow task-specific patches. Within the constraints of scaffold-level evolution under sandboxed, budgeted archive search, MGM potentially points to a compute-efficient path for scalable self-improving agent development, where scaffolds evolved on smaller and cheaper backbones can later be reused on stronger models.

Ethical Considerations

MGM advances self-improving coding agents that can modify their own source code. Such systems have a different risk profile from static models because an erroneous or adversarial self-edit can persist in later descendants. In our experiments, self-modification and evaluation run inside isolated containers with no network access and with read-only mounts of the host file system, following the safety protocol of prior archive-based self-improving agents. Any deployment beyond this sandboxed coding setting requires a fresh review of the execution boundary. Giving a self-modifying agent access to production systems, external APIs, or writable storage outside controlled containers could allow unintended changes to spread in ways that are hard to audit or reverse. Practitioners should retain the isolated execution model and subject any broader action space to explicit safety review before deployment.

The same capability also has dual-use implications. Scaffold-level self-improvement can amplify both beneficial coding assistance and harmful automation, including generation of malicious software or exploitation of vulnerable systems, if isolation is removed. Our public release is intended for research on sandboxed self-evolution. We discourage unconstrained deployment of evolved scaffolds outside controlled environments.

The benchmarks and backbone models used here may also inherit harmful or biased content. SWE-bench, SWE-bench Pro, and Polyglot draw on public open-source repositories, and the backbone LLMs are trained on large web corpora, all of which may contain insecure code patterns and may overrepresent particular languages, ecosystems, or problem domains. We did not introduce dedicated bias or safety audits of the evolved agents, so inherited fairness and security issues may persist. In addition, public benchmark material may overlap with pretraining data, which limits how strongly offline scores should be read as evidence of robust real-world competence. This work focuses on self-evolution of coding agents only, and our evolved agents are not equipped to act outside software-engineering related/ repositories. Task-specific safety evaluation remains necessary before any use beyond general-purpose coding assistance.

AI use statement

Generative AI tools were used solely for language editing and writing polish, including improvements to grammar, clarity, and readability. They were not used to generate research ideas, develop methods, conduct experiments, analyze results, derive theoretical results, or draw scientific conclusions. All AI-assisted edits were reviewed by the authors, who take full responsibility for the final content of this work.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=qV83K9d5WB.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence, 2020. URL https://arxiv.org/abs/1905.10985.

Antoine Cully. Multi-emitter map-elites: improving quality, diversity and data efficiency with heterogeneous sets of emitters. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '21, pp. 84–92. ACM, 2021. doi: 10.1145/3449639.3459326. URL http://dx.doi.org/10.1145/3449639.3459326.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi

Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Jordan S. Ellenberg, Cristofero S. Fraser-Taliente, Thomas R. Harvey, Karan Srivastava, and Andrew V. Sutherland. Generative modeling for mathematical discovery, 2025. URL https://arxiv.org/abs/2503.11061.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru WANG, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856. URL https://openreview.net/forum?id=CTr3bovS5F. Survey Certification.

Paul Gauthier. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, December 2024. Accessed: 2026-05-22.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

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 B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning, 2022. URL https://arxiv.org/abs/2205.00445.

Kabir Khandpur and the SWE-bench Team. SWE-bench Multilingual. https://www.swebench.com/multilingual.html, 2025. 300 tasks across 9 languages and 42 repositories. Official citation guidance points to yang2025swesmith.

Bojie Li. Incompressible knowledge probes: Estimating black-box llm parameter counts via factual capacity, 2026. URL https://arxiv.org/abs/2604.24827.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

MiniMax, :, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Enwei Jiao, Haigang Zhou, Haimo Zhang, Han Ding, Haohai Sun, Haoyu Feng, Huaiguang Cai, Haichao Zhu, Jian Sun, Jiaqi Zhuang, Jiaren Cai, Jiayuan Song, Jin Zhu, Jingyang Li, Jinhao Tian, Jinli Liu, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kaiyi Feng, Ke Yang, Kecheng Xiao, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Li, Lin Zheng, Linge Du, Lingyu Yang, Lunbin Zeng, Minghui Yu, Mingliang Tao, Mingyuan Chi, Mozhi Zhang, Mujie Lin, Nan Hu, Nongyu Di, Peng Gao, Pengfei Li, Pengyu Zhao, Qibing Ren, Qidi Xu, Qile Li, Qin Wang, Rong Tian, Ruitao Leng, Shaoxiang Chen, Shaoyu Chen, Shengmin Shi, Shitong Weng, Shuchang Guan, Shuqi Yu, Sichen Li, Songquan Zhu, Tengfei Li, Tianchi Cai, Tianrun Liang, Weiyu Cheng, Weize Kong, Wenkai Li, Xiancai Chen, Xiangjun Song, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xinzhu Hou, Xuan Lu, Xun Zou, Xuyang Shen, Yan Gong, Yan Ma, Yang Wang, Yiqi Shi, Yiran Zhong, Yonghong Duan, Yongxiang Fu, Yongyi Hu, Yu Gao, Yuanxiang Fan, Yufeng Yang, Yuhao Li, Yulin Hu, Yunan Huang, Yunji Li, Yunzhi Xu, Yuxin Mao, Yuxuan Shi, Yuze Wenren, Zehan Li, Zelin Li, Zhanxu Tian, Zhengmao Zhu, Zhenhua Fan, Zhenzhen Wu, Zhichao Xu, Zhihang Yu, Zhiheng Lyu, Zhuo Jiang, Zibo Gao, Zijia Wu, Zijian Song, and Zijun Sun. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://arxiv.org/abs/2506.13585.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Massimo Pigliucci. *Phenotypic Plasticity: Beyond Nature and Nurture*. Johns Hopkins University Press, 2001.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.462. URL https://aclanthology.org/2023.findings-emnlp.462/.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution, 2025. URL https://arxiv.org/abs/2505.20286.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. In *Scaling Self-Improving Foundation Models without Human Supervision*, 2025. URL https://openreview.net/forum?id=rShJCyLsOr.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

Jurgen Schmidhuber. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 1987. URL https://mediatum.ub.tum.de/?id=813180.

Jürgen Schmidhuber. Goedel machines: Self-referential universal problem solvers making provably optimal self-improvements. *CoRR*, cs.LO/0309048, 2003. URL http://arxiv.org/abs/cs/0309048.

Jürgen Schmidhuber. *Gödel Machines: Fully Self-referential Optimal Universal Self-improvers*, pp. 199–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-68677-4. doi: 10.1007/978-3-540-68677-4_7. URL https://doi.org/10.1007/978-3-540-68677-4_7.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, Alex Makelov, Alex Neitz, Alex Wei, Alexandra Barr, Alexandre Kirchmeyer, Alexey Ivanov, Alexi Christakis, Alistair Gillespie, Allison Tam, Ally Bennett, Alvin Wan, Alyssa Huang, Amy McDonald Sandjideh, Amy Yang, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrei Gheorghe, Andres Garcia Garcia, Andrew Braunstein, Andrew Liu, Andrew Schmidt, Andrey Mereskin, Andrey Mishchenko, Andy Applebaum, Andy Rogerson, Ann Rajan, Annie Wei, Anoop Kotha, Anubha Srivastava, Anushree Agrawal, Arun Vijayvergiya, Ashley Tyra, Ashvin Nair, Avi Nayak, Ben Eggers, Bessie Ji, Beth Hoover, Bill Chen, Blair Chen, Boaz Barak, Borys Minaiev, Botao Hao, Bowen Baker, Brad Lightcap, Brandon McKinzie, Brandon Wang, Brendan

Quinn, Brian Fioca, Brian Hsu, Brian Yang, Brian Yu, Brian Zhang, Brittany Brenner, Callie Riggins Zetino, Cameron Raymond, Camillo Lugaresi, Carolina Paz, Cary Hudson, Cedric Whitney, Chak Li, Charles Chen, Charlotte Cole, Chelsea Voss, Chen Ding, Chen Shen, Chengdu Huang, Chris Colby, Chris Hallacy, Chris Koch, Chris Lu, Christina Kaplan, Christina Kim, CJ Minott-Henriques, Cliff Frey, Cody Yu, Coley Czarnecki, Colin Reid, Colin Wei, Cory Decareaux, Cristina Scheau, Cyril Zhang, Cyrus Forbes, Da Tang, Dakota Goldberg, Dan Roberts, Dana Palmie, Daniel Kappler, Daniel Levine, Daniel Wright, Dave Leo, David Lin, David Robinson, Declan Grabb, Derek Chen, Derek Lim, Derek Salama, Dibya Bhattacharjee, Dimitris Tsipras, Dinghua Li, Dingli Yu, DJ Strouse, Drew Williams, Dylan Hunn, Ed Bayes, Edwin Arbus, Ekin Akyurek, Elaine Ya Le, Elana Widmann, Eli Yani, Elizabeth Proehl, Enis Sert, Enoch Cheung, Eri Schwartz, Eric Han, Eric Jiang, Eric Mitchell, Eric Sigler, Eric Wallace, Erik Ritter, Erin Kavanaugh, Evan Mays, Evgenii Nikishin, Fangyuan Li, Felipe Petroski Such, Filipe de Avila Belbute Peres, Filippo Raso, Florent Bekerman, Foivos Tsimpourlas, Fotis Chantzis, Francis Song, Francis Zhang, Gaby Raila, Garrett McGrath, Gary Briggs, Gary Yang, Giambattista Parascandolo, Gildas Chabot, Grace Kim, Grace Zhao, Gregory Valiant, Guillaume Leclerc, Hadi Salman, Hanson Wang, Hao Sheng, Haoming Jiang, Haoyu Wang, Haozhun Jin, Harshit Sikchi, Heather Schmidt, Henry Aspegren, Honglin Chen, Huida Qiu, Hunter Lightman, Ian Covert, Ian Kivlichan, Ian Silber, Ian Sohl, Ibrahim Hammoud, Ignasi Clavera, Ikai Lan, Ilge Akkaya, Ilya Kostrikov, Irina Kofman, Isak Etinger, Ishaan Singal, Jackie Hehir, Jacob Huh, Jacqueline Pan, Jake Wilczynski, Jakub Pachocki, James Lee, James Quinn, Jamie Kiros, Janvi Kalra, Jasmyn Samaroo, Jason Wang, Jason Wolfe, Jay Chen, Jay Wang, Jean Harb, Jeffrey Han, Jeffrey Wang, Jennifer Zhao, Jeremy Chen, Jerene Yang, Jerry Tworek, Jesse Chand, Jessica Landon, Jessica Liang, Ji Lin, Jiancheng Liu, Jianfeng Wang, Jie Tang, Jihan Yin, Joanne Jang, Joel Morris, Joey Flynn, Johannes Ferstad, Johannes Heidecke, John Fishbein, John Hallman, Jonah Grant, Jonathan Chien, Jonathan Gordon, Jongsoo Park, Jordan Liss, Jos Kraaijeveld, Joseph Guay, Joseph Mo, Josh Lawson, Josh McGrath, Joshua Vendrow, Joy Jiao, Julian Lee, Julie Steele, Julie Wang, Junhua Mao, Kai Chen, Kai Hayashi, Kai Xiao, Kamyar Salahi, Kan Wu, Karan Sekhri, Karan Sharma, Karan Singhal, Karen Li, Kenny Nguyen, Keren Gu-Lemberg, Kevin King, Kevin Liu, Kevin Stone, Kevin Yu, Kristen Ying, Kristian Georgiev, Kristie Lim, Kushal Tirumala, Kyle Miller, Lama Ahmad, Larry Lv, Laura Clare, Laurance Fauconnet, Lauren Itow, Lauren Yang, Laurentia Romaniuk, Leah Anise, Lee Byron, Leher Pathak, Leon Maksin, Leyan Lo, Leyton Ho, Li Jing, Liang Wu, Liang Xiong, Lien Mamitsuka, Lin Yang, Lindsay McCallum, Lindsey Held, Liz Bourgeois, Logan Engstrom, Lorenz Kuhn, Louis Feuvrier, Lu Zhang, Lucas Switzer, Lukas Kondraciuk, Lukasz Kaiser, Manas Joglekar, Mandeep Singh, Mandip Shah, Manuka Stratta, Marcus Williams, Mark Chen, Mark Sun, Marselus Cayton, Martin Li, Marvin Zhang, Marwan Aljubeh, Matt Nichols, Matthew Haines, Max Schwarzer, Mayank Gupta, Meghan Shah, Melody Y. Guan, Melody Huang, Meng Dong, Mengqing Wang, Mia Glaese, Micah Carroll, Michael Lampe, Michael Malek, Michael Sharman, Michael Zhang, Michele Wang, Michelle Pokrass, Mihai Florian, Mikhail Pavlov, Miles Wang, Ming Chen, Mingxuan Wang, Minnia Feng, Mo Bavarian, Molly Lin, Moose Abdool, Mostafa Rohaninejad, Nacho Soto, Natalie Staudacher, Natan LaFontaine, Nathan Marwell, Nelson Liu, Nick Preston, Nick Turley, Nicklas Ansman, Nicole Blades, Nikil Pancha, Nikita Mikhaylin, Niko Felix, Nikunj Handa, Nishant Rai, Nitish Keskar, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Oona Gleeson, Pamela Mishkin, Patryk Lesiewicz, Paul Baltescu, Pavel Belov, Peter Zhokhov, Philip Pronin, Phillip Guo, Phoebe Thacker, Qi Liu, Qiming Yuan, Qinghua Liu, Rachel Dias, Rachel Puckett, Rahul Arora, Ravi Teja Mullapudi, Raz Gaon, Reah Miyara, Rennie Song, Rishabh Aggarwal, RJ Marsan, Robel Yemiru, Robert Xiong, Rohan Kshirsagar, Rohan Nuttall, Roman Tsiupa, Ronen Eldan, Rose Wang, Roshan James, Roy Ziv, Rui Shu, Ruslan Nigmatullin, Saachi Jain, Saam Talaie, Sam Altman, Sam Arnesen, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Sarah Yoo, Savannah Heon, Scott Ethersmith, Sean Grove, Sean Taylor, Sebastien Bubeck, Sever Banesiu, Shaokyi Amdo, Shengjia Zhao, Sherwin Wu, Shibani Santurkar, Shiyu Zhao, Shraman Ray Chaudhuri, Shreyas Krishnaswamy, Shuaiqi, Xia, Shuyang Cheng, Shyamal Anadkat, Simón Posada Fishman, Simon Tobin, Siyuan Fu, Somay Jain, Song Mei, Sonya Egoian, Spencer Kim, Spug Golden, SQ Mah, Steph Lin, Stephen Imm, Steve Sharpe, Steve Yadlowsky, Sulman Choudhry, Sungwon Eum, Suvansh Sanjeev, Tabarak Khan, Tal Stramer, Tao Wang, Tao Xin, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Degry, Thomas Shadwell, Tianfu Fu, Tianshi Gao, Timur Garipov, Tina Sriskandarajah, Toki Sherbakov, Tomek Korbak, Tomer Kaftan, Tomo Hiratsuka, Tongzhou Wang, Tony Song, Tony Zhao, Troy Peterson, Val Kharitonov, Victoria Chernova, Vineet Kosaraju, Vishal Kuo, Vitchyr Pong, Vivek Verma, Vlad Petrov, Wanning Jiang, Weixing Zhang, Wenda Zhou, Wenlei Xie, Went-

ing Zhan, Wes McCabe, Will DePue, Will Ellsworth, Wulfie Bain, Wyatt Thompson, Xiangning Chen, Xiangyu Qi, Xin Xiang, Xinwei Shi, Yann Dubois, Yaodong Yu, Yara Khakbaz, Yifan Wu, Yilei Qian, Yin Tat Lee, Yinbo Chen, Yizhen Zhang, Yizhong Xiong, Yonglong Tian, Young Cha, Yu Bai, Yu Yang, Yuan Yuan, Yuanzhi Li, Yufeng Zhang, Yuguang Yang, Yujia Jin, Yun Jiang, Yunyun Wang, Yushi Wang, Yutian Liu, Zach Stubenvoll, Zehao Dou, Zheng Wu, and Zhigang Wang. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, Yanru Chen, Yanxu Chen, Yicun Chen, Yimin Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Ziwei Chen, Dazhi Cheng, Minghan Chu, Jialei Cui, Jiaqi Deng, Muxi Diao, Hao Ding, Mengfan Dong, Mengnan Dong, Yuxin Dong, Yuhao Dong, Angang Du, Chenzhuang Du, Dikang Du, Lingxiao Du, Yulun Du, Yu Fan, Shengjun Fang, Qiulin Feng, Yichen Feng, Garimugai Fu, Kelin Fu, Hongcheng Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Chengyang Gong, Xiaochen Gong, Zhuoma Gongque, Qizheng Gu, Xinran Gu, Yicheng Gu, Longyu Guan, Yuanying Guo, Xiaoru Hao, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Jiaxi Hu, Yangyang Hu, Zhenxing Hu, Ke Huang, Ruiyuan Huang, Weixiao Huang, Zhiqi Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yu Jing, Guokun Lai, Aidi Li, C. Li, Cheng Li, Fang Li, Guanghe Li, Guanyu Li, Haitao Li, Haoyang Li, Jia Li, Jingwei Li, Junxiong Li, Lincan Li, Mo Li, Weihong Li, Wentao Li, Xinhang Li, Xinhao Li, Yang Li, Yanhao Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zheming Li, Weilong Liao, Jiawei Lin, Xiaohan Lin, Zhishan Lin, Zichao Lin, Cheng Liu, Chenyu Liu, Hongzhang Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Tianyu Liu, Weizhou Liu, Xiangyan Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yuanxin Liu, Yue Liu, Zhengying Liu, Zhongnuo Liu, Enzhe Lu, Haoyu Lu, Zhiyuan Lu, Junyu Luo, Tongxu Luo, Yashuo Luo, Long Ma, Yingwei Ma, Shaoguang Mao, Yuan Mei, Xin Men, Fanqing Meng, Zhiyong Meng, Yibo Miao, Minqing Ni, Kun Ouyang, Siyuan Pan, Bo Pang, Yuchao Qian, Ruoyu Qin, Zeyu Qin, Jiezhong Qiu, Bowen Qu, Zeyu Shang, Youbo Shao, Tianxiao Shen, Zhennan Shen, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Feifan Song, Pengwei Song, Tianhui Song, Xiaoxi Song, Hongjin Su, Jianlin Su, Zhaochen Su, Lin Sui, Jinsong Sun, Junyao Sun, Tongyu Sun, Flood Sung, Yunpeng Tai, Chuning Tang, Heyi Tang, Xiaojuan Tang, Zhengyang Tang, Jiawen Tao, Shiyuan Teng, Chaoran Tian, Pengfei Tian, Ao Wang, Bowen Wang, Chensi Wang, Chuang Wang, Congcong Wang, Dingkun Wang, Dinglu Wang, Dongliang Wang, Feng Wang, Hailong Wang, Haiming Wang, Hengzhi Wang, Huaqing Wang, Hui Wang, Jiahao Wang, Jinhong Wang, Jiuzheng Wang, Kaixin Wang, Linian Wang, Qibin Wang, Shengjie Wang, Shuyi Wang, Si Wang, Wei Wang, Xiaochen Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yipu Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhexu Wang, Zihan Wang, Zizhe Wang, Chu Wei, Ming Wei, Chuan Wen, Zichen Wen, Chengjie Wu, Haoning Wu, Junyan Wu, Rucong Wu, Wenhao Wu, Yuefeng Wu, Yuhao Wu, Yuxin Wu, Zijian Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Yuchong Xie, Yifei Xin, Bowei Xing, Boyu Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinbo Xu, Xinran Xu, Yangchuan Xu, Yichang Xu, Yuemeng Xu, Zelai Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Guangyao Yang, Hao Yang, Junwei Yang, Kai Yang, Ningyuan Yang, Ruihan Yang, Xiaofei Yang, Xinlong Yang, Ying Yang, Yi Yang, Yi Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Dan Ye, Wenjie Ye, Zhuorui Ye, Bohong Yin, Chengzhen Yu, Longhui Yu, Tao Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Xiaokun Yuan, Yang Yue, Weihao Zeng, Dunyuan Zha, Haobing Zhan, Dehao Zhang, Hao Zhang, Jin Zhang, Puqi Zhang, Qiao Zhang, Rui Zhang, Xiaobin Zhang, Y. Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yushun Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Chenguang Zhao, Feifan Zhao, Jinxiang Zhao, Shuai Zhao, Xiangyu Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Junfeng Zhong, Longguang Zhong, Weiming Zhong, M. Zhou, Runjie Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yuxuan Zhu, Zhen Zhu, Jingze Zhuang, Weiyu Zhuang, Ying Zou, and Xinxing Zu. Kimi k2.5: Visual agentic intelligence, 2026a. URL https://arxiv.org/abs/2602.02276.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong,

Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026b. URL https://arxiv.org/abs/2507.20534.

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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=T0EiEuhOOL.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022. URL https://proceedings.neurips.cc/paper%5Ffiles/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Richard Woltereck. Weitere experimentelle Untersuchungen über Artveränderung, speziell über das Wesen quantitativer Artunterschiede bei Daphniden. *Verhandlungen der deutschen zoologischen Gesellschaft*, 19:110–173, 1909. URL https://cir.nii.ac.jp/crid/1573668924237108864.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly?, 2025. URL https://arxiv.org/abs/2511.13646.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International*

Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=pUpzQZTvGY.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026b. URL https://arxiv.org/abs/2603.19461.

Appendix Contents

A Additional Discussions and Limitations **20**

A.1 When Does Hybridization Help? A Case Study 20

A.2 Bigger Model Does Not Always Lead to Better Results 20

A.3 Are skills evolved from MGM really more general? 22

A.4 Limitations 22

B Additional Related Work **23**

B.1 From Agent Design to Inherited Self-Modification 23

B.2 Runtime Adaptation versus Training-Time Self-Evolution 24

B.3 Open-Ended Search and Algorithm Discovery 24

B.4 Benchmark Coverage and Evaluation Motivation 25

C Method Details **25**

C.1 Pseudocode of Mendel Gödel Machine 25

C.2 Primary-Lineage Selection in Cross-Lineage Hybridization 25

D Theory and Simulations **26**

D.1 Full Additive Fitness Landscape 26

D.2 Theoretical Justification of Comparative Fix Probability 27

D.3 Monte Carlo Protocol and Parameters 31

E Experiment Details **31**

E.1 Compute and Software Environment 31

E.2 Hyper-parameter Settings 32

F Benchmark Details **33**

G Additional Results **35**

G.1 Full Evaluation on Polyglot 35

G.2 Visualizations of Evolution Trees 36

H Best Discovered Agents **38**

H.1 MGM on Polyglot 38

H.2 MGM on SWE-bench Verified 38

A Additional Discussions and Limitations

A.1 When Does Hybridization Help? A Case Study

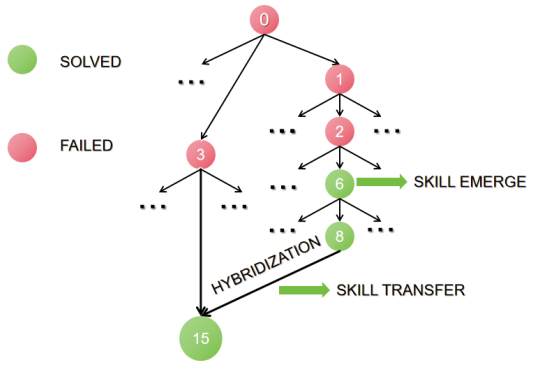


Figure 7: **Illustration of Cross-lineage Hybridization**. Nodes are colored by their outcome on the shared diagnostic task `javascript__queen-attack`: red nodes fail and green nodes solve it.

Just as Figure 7 illustrates, **Cross-lineage Hybridization** enables the evolutionary process to transfer skills discovered in one archive to another, thereby improving evolutionary efficiency. In this example, both node #1 and node #2 initially fail on the same diagnostic task, `javascript__queen-attack`. Along the donor lineage, node #2 later produces node #6, which is the first descendant in this branch to solve the task. The evolved behavior can be summarized as a test-contract skill. Before implementing, the agent learns to read tests as strict behavioral contracts, preserving exact public APIs, expected values, and error messages. This capability is then preserved in node #8, which also solves `javascript__queen-attack`. Crucially, node #8 does not merely improve its own lineage. Through Cross-lineage Hybridization, it serves as a successful context donor for a separate failing lineage rooted at node #3. The resulting hybrid child, node #15, also solves the previously failed task. This case shows that hybridization is not simply another mutation operator, instead it acts as a cross-archive mechanism for capability transfer, allowing a reusable skill evolved in one branch to accelerate progress in another branch that had not discovered it independently.

A.2 Bigger Model Does Not Always Lead to Better Results

Since recursive self-improvement ultimately edits the agent's own codebase, it is tempting to view self-evolution as another coding task. Under this view, a larger or more coding-specialized foundation model should naturally lead to stronger self-improvement: if a model is better at coding benchmarks, it should also be better at modifying the agent scaffold. However, our experiments here show that this intuition is incomplete. Self-improvement is indeed implemented through code editing, but the quality of the edit depends critically on preceding diagnosis steps, where the model must infer why the current scaffold failed and what general modification should be made.

	Qwen3-Coder-Next-80B-A3B			Qwen3.6-35B-A3B		
Agent	Initial	HGM	MGM	Initial	HGM	MGM
Accuracy	33.3	$40.0^{+6.7\%}$	$\mathbf{41.7}^{+8.4\%}$	68.3	$73.3^{+5.0\%}$	$\mathbf{78.3}^{+10.0\%}$
% Impr.	–	↑ 20.1%	↑ **25.2%**	–	↑ 7.3%	↑ **14.6%**
Time	–	150 h	**148 h**	–	93.02 h	96.11 h

Table 5: **Performance of coding agents evolved on SWE-bench Verified with different models.** Results evolved using 200 φ-evaluations and 24 Φ-expansions. For each benchmark, HGM and MGM start from the same initial scaffold. Superscripts in accuracy denote absolute percentage-point improvements over corresponding initial agents. Time reported as CPU wall-clock time with 8×NVIDIA H100 GPUs.

To examine this, we conduct an additional experiment on SWE-bench Verified using Qwen3-Coder-Next-80B-A3B as the backbone model. Qwen3-Coder-Next is larger and explicitly optimized for coding with comparable coding ability but poorer reasoning ability, so one might expect it to produce

Benchmark	Main Capability	Qwen3-Coder-Next	Qwen3.6-35B-A3B
MMLU-Redux	General knowledge / robust reasoning	91.18	**93.3**$^{+2.12}$
MMLU-Pro	Harder general knowledge reasoning	80.52	**85.2**$^{+4.68}$
GPQA	Graduate-level science reasoning	74.49	**86.0**$^{+11.51}$
SuperGPQA	Broader and harder QA	57.45	**64.7**$^{+7.25}$
HMMT Feb 2025	Competition-level mathematics	70.21	**90.7**$^{+20.49}$
HMMT Nov 2025	Competition-level mathematics	75.57	**89.1**$^{+13.53}$

Table 6: **Performance comparison between models.** Results are compared across general knowledge, science, and mathematics reasoning benchmarks.

stronger self-improving agents. Surprisingly, this is not what we observe. As shown in Table 5, the initial scaffold obtains 33.3% accuracy. Under the same budget of 200 task evaluations and 24 self-modification expansions, HGM improves the scaffold to 40.0%, while MGM improves it to 41.7%. MGM still outperforms HGM under this backbone, but the final evolved performance is substantially lower than the corresponding Qwen3.6-35B-A3B setting, where MGM reaches 78.3% on SWE-bench Verified-60.

This suggests that self-improvement performance cannot be predicted solely from model size or coding specialization. The model must inspect the trajectory, diagnose the underlying failure, and transform that diagnosis into a robust scaffold-level edit. This diagnosis step is not equivalent to ordinary code completion or local bug fixing. It requires the model to reason about the interaction between prompts, tools, control flow, repository exploration, execution feedback, and prior agent behavior. If the diagnosis is shallow or incorrect, the resulting code edit may still be syntactically valid, but it can target the wrong mechanism, overfit to superficial symptoms, or introduce brittle workflow changes.

To better understand this result, as shown in Table 6, we compare the broader reasoning abilities of Qwen3-Coder-Next-80B-A3B and Qwen3.6-35B-A3B. Although Qwen3-Coder-Next is larger and coding-oriented, Qwen3.6-35B-A3B performs substantially better on a wide range of general and reasoning-intensive benchmarks. Qwen3.6 scores 93.3 on MMLU-Redux, compared with 91.18 for Qwen3-Coder-Next, and 85.2 on MMLU-Pro, compared with 80.52. The gaps become larger on more difficult reasoning tasks: Qwen3.6 outperforms Qwen3-Coder-Next by 11.51 points on GPQA, 7.25 points on SuperGPQA, 20.49 points on HMMT Feb 2025, 13.53 points on HMMT Nov 2025, and 21.47 points on LiveCodeBench v6. These results indicate that Qwen3.6 has a stronger general reasoning profile, despite having fewer total parameters.

This reasoning advantage helps explain why Qwen3.6 leads to stronger self-improvement. In a self-evolving coding agent, the model is not only asked to write code; it is asked to reason about why the current agent failed and how the scaffold should be changed to prevent similar failures in future tasks. The edit itself is a coding operation, but deciding what to edit is a diagnosis and abstraction problem. A coding-specialized model may be strong at implementing local changes, yet still produce weaker self-improvement if it cannot reliably infer the correct failure mechanism from trajectories. Conversely, a model with stronger reasoning ability can generate more accurate diagnoses and therefore produce more reusable scaffold-level modifications.

This observation is particularly important for MGM. MGM introduces reaction-norm mutation and cross-lineage hybridization, both of which increase the amount of comparative evidence available to the self-modification step. However, richer evidence is only useful if the backbone model can reason over it. Reaction-norm mutation requires the model to compare multiple trajectories of the same agent and identify recurring behavioral patterns. Cross-lineage hybridization requires the model to compare different agents on the same task and extract transferable scaffold-level traits. These operations make the diagnosis stage more informative, but also more reasoning-intensive.

Overall, our results do not suggest that larger or more coding-specialized models are ineffective for self-improvement. Qwen3-Coder-Next still enables both HGM and MGM to improve over the initial scaffold, showing that strong coding backbones can support meaningful self-evolution. However, the improvement is not necessarily monotonic with model size or coding specialization. Although self-evolution ultimately edits code, the quality of the edit depends on whether the model can diagnose the failure mechanism before modifying the scaffold. A larger coding model may therefore produce valid self-edits, but not necessarily better self-edits, if its trajectory-level diagnosis and abstraction

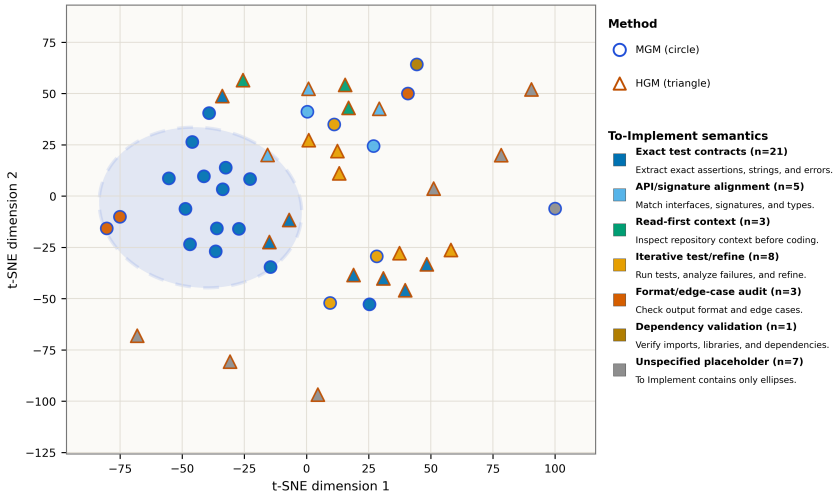


Figure 8: **Semantic visualization of evolved skills.** A content-level view of the evolved "To Implement" descriptions from MGM and HGM. Each point corresponds to one evolved skill or workflow change, embedded by the semantics of its To Implement text. The marker shape indicates the evolution method, while color indicates the main semantic theme of the proposed change.

ability are weaker. This suggests that backbone selection for self-improving agents should consider a broader capability profile, especially general reasoning and failure-diagnosis ability, rather than relying only on parameter count or coding benchmark performance.

A.3 Are skills evolved from MGM really more general?

The visualization 8 suggests that MGM produces changes that concentrate around a reusable workflow-level capability. Most MGM points lie in a compact semantic region centered on exact test-contract extraction, API-contract adherence, and pre-implementation verification. These are not tied to a particular programming language, benchmark instance, or repository-specific bug. Instead, they jointly describe a general procedure incorporating reading tests and existing interfaces, extracting the expected contract, implementing against that contract, and verifying exact agreement before finalizing the patch. This kind of skills can transfer across many programming tasks because almost all repair problems involve some form of latent contract between tests, existing code, and the expected implementation.

In contrast, HGM points are more widely dispersed in the semantic map. HGM also discovers useful ideas, such as iterative testing, API alignment, context discovery, and auxiliary tool utilities, but these ideas appear less consolidated into a single reusable default workflow. The broader spread of HGM points suggests a more heterogeneous search over possible interventions. Some HGM changes add or wire in tools, while others propose isolated prompt phases or iterative loops. These can be beneficial, but they are less consistently organized around one general mechanism that would apply by default across unseen tasks.

This distinction is important because generalizability should not be measured only by whether a change uses broad language such as general capability. A more operational criterion is whether the evolved skill abstracts away from the original training instance and becomes a reusable procedure for a broad class of future tasks. Under this criterion, MGM appears more general at the workflow level. It repeatedly evolves skills that convert task-specific feedback into a language-agnostic contract-extraction and verification protocol. HGM, by comparison, evolves a wider variety of interventions, but they are more fragmented and less clearly integrated into a stable general workflow.

A.4 Limitations

Self-improving coding-agent experiments remain expensive. Even though MGM reuses existing trajectories for its comparative operators, evolution and evaluation on repository-level tasks still

consume substantial wall-clock and GPU resources. This constrains the number of independent evolution seeds and the breadth of hyperparameter sweeps we can report.

MGM is history-dependent. Its advantage comes from comparative evidence in the archive. Reaction-norm mutation needs multiple trajectories from the same agent, and cross-lineage hybridization needs overlapping tasks across lineages. When the archive is small, task overlap is sparse, or informative failures have not yet appeared, MGM has little comparative evidence and may behave like single-trajectory baselines. The failed-task pool partially mitigates this by increasing diagnostic overlap, but it cannot create informative contrasts before failures accumulate.

MGM improves the evidence given to self-modification, but it does not guarantee that the resulting edit is correct, general, or maintainable. The Mendelian operators expose useful behavioral contrasts, yet the actual scaffold change is still produced by an LLM-based editor. High-quality evidence therefore need not yield a high-quality modification, and failed edits can waste budget. Relatedly, comparative evidence helps only when the backbone can diagnose failure mechanisms from trajectories. A stronger coding specialist with weaker general reasoning may still produce weaker self-improvement under the same operators.

Our formal analysis and Monte Carlo study use an additive fitness surrogate. They isolate diagnostic compression under controlled assumptions and do not capture the full complexity of editable agent scaffolds. Empirically, primary evolution is reported on fixed 60-task subsets under a single matched budget, with broader checks on full Polyglot and held-out SWE-bench variants. Subset selection and limited seed diversity leave residual uncertainty about variance across random restarts and alternative task samples. Finally, our claims concern coding-agent scaffolds evaluated on public software-engineering benchmarks. They do not establish that the same operators would transfer unchanged to non-coding agents or open-ended real-world software maintenance.

B Additional Related Work

As foundation models continue to advance, including DeepSeek-AI et al. (2025; 2026); Team et al. (2026b;a); Singh et al. (2026); MiniMax et al. (2025), the attainable performance ceiling of coding agents is also steadily increasing. The development of agentic systems has progressed from human-designed scaffolds (Cai et al., 2024; Qian et al., 2023), to automated agent design(Hu et al., 2025), and more recently to self-improving agents(Zhang et al., 2026a; Wang et al., 2026; Qiu et al., 2025; Gao et al., 2026).

B.1 From Agent Design to Inherited Self-Modification

Modern LLM agents build on a broad line of work on tool use, reasoning-action interleaving, and multi-agent orchestration, including modular tool-augmented systems (Karpas et al., 2022), ReAct-style reasoning-and-acting (Yao et al., 2023), self-supervised tool use (Schick et al., 2023), and multi-agent frameworks such as AutoGen and MetaGPT (Wu et al., 2023; Hong et al., 2024). These general agentic paradigms become especially important in software engineering, where an agent must inspect repositories, edit files, execute commands, and validate patches. Systems such as SWE-agent emphasize the importance of the agent-computer interface: a model's ability to inspect files, edit code, execute commands, and run tests depends heavily on the tools and interaction protocol exposed to it (Yang et al., 2024). More recently, Hyperagents integrates a task agent and a meta-agent into a single editable program, allowing the meta-agent to modify both the task agent and itself, including the mechanism that generates future improvements. This makes the meta-level self-improvement procedure itself evolvable, extending self-improvement beyond modifications to a fixed task-agent scaffold. (Zhang et al., 2026b). These systems show that scaffold design is central to coding-agent performance, but the scaffold is still largely engineered by humans.

ADAS shifts scaffold design from manual engineering to automated search. Meta Agent Search represents agents as code and uses a meta-agent to invent new agent programs from an archive of prior discoveries (Hu et al., 2025). This view is important because it treats prompts, tools, and control flow as searchable program components rather than fixed infrastructure. However, ADAS-style methods usually preserve a separation between the designer and the designed agent. The meta-agent is responsible for generating new target agents, whereas the target agent does not necessarily improve itself through its own execution history.

Self-improving coding agents reduce this separation. SICA shows that a coding agent can use basic file-editing tools to modify its own codebase and improve on benchmarks (Robeyns et al., 2025). DGM extends this into open-ended evolution by maintaining an archive of self-modified agents and sampling from it to create new descendants (Zhang et al., 2026a). HGM further observes that an agent's current benchmark performance is not always aligned with its future self-improvement potential, and therefore estimates descendant-based metaproductivity to guide archive expansion (Wang et al., 2026). These works establish the XGM setting: an agent population evolves through persistent, heritable edits to executable scaffolds.

MGM focuses on a different bottleneck inside this setting. Prior Gödel Machine-style methods mainly decide which node in the archive should be expanded. MGM instead asks what evidence should be given to the editor once an expansion is triggered. A single failed trajectory can be noisy: it may reflect a task-specific accident, a bad local choice, or a general weakness in the agent's design. MGM reduces this ambiguity by constructing controlled comparisons from trajectories that are already present in the archive. Reaction-norm mutation compares the same genotype across multiple environments, making recurring failures more likely to reveal a stable design weakness. Cross-lineage hybridization compares different genotypes on the same task, making behavioral differences easier to interpret as transferable skills. Therefore, MGM improves the diagnostic quality of self-modification without requiring additional task evaluations.

B.2 Runtime Adaptation versus Training-Time Self-Evolution

Live-SWE-agent is the most relevant concurrent work to distinguish from MGM. It starts from a minimal agent scaffold and lets the agent expand or revise its own capabilities during the process of solving a real-world software issue (Xia et al., 2025). This is a powerful runtime adaptation mechanism: the agent can create helper tools, refine its own execution procedure, and specialize its workflow to the current repository. Its central advantage is immediacy. It does not require an offline evolution loop before deployment, and the agent can adapt to the concrete structure of the issue it is currently solving.

MGM addresses a different question. Rather than asking how an agent should adapt inside one episode, MGM asks how a population of agents should improve across many episodes so that future agents inherit better scaffolds. The modifications produced by MGM are persistent changes to the agent program. They are evaluated over a task distribution and stored as descendants in an archive. This makes MGM a training-time self-evolution method, even though the training signal is not gradient-based. The objective is to discover general capabilities—better planning, validation, context management, debugging, or tool usage—that remain useful beyond the tasks that exposed them.

This distinction is similar to the difference between chain-of-thought prompting and policy optimization. Chain-of-thought prompting allocates more inference-time computation to a single response, improving the current trajectory without changing the model parameters (Wei et al., 2022). GRPO, by contrast, is a reinforcement learning algorithm that updates the policy so that future trajectories improve (Shao et al., 2024). Live-SWE-agent plays a role analogous to test-time reasoning or skill construction: it improves the current problem-solving process. MGM plays a role analogous to policy learning: it changes the inherited scaffold that future agents use. These paradigms are complementary. A strong practical system could first use MGM to evolve robust base agents offline and then use live runtime adaptation to specialize the evolved agent to the current issue.

B.3 Open-Ended Search and Algorithm Discovery

MGM is also related to open-ended and quality-diversity search. In these paradigms, progress does not come only from optimizing a single incumbent solution, but from maintaining an archive of diverse candidates that can serve as stepping stones for future discovery. Quality-diversity methods such as MAP-Elites maintain structured archives of high-performing yet behaviorally diverse solutions, improving both search coverage and downstream adaptability (Mouret & Clune, 2015; Cully, 2021). Open-ended evolution further emphasizes that preserving diverse lineages can expose stepping stones that would be missed by purely exploitative optimization (Clune, 2020).

Recent foundation-model-based discovery systems instantiate a related idea in code and scientific domains. AlphaEvolve uses a coding agent to iteratively propose, evaluate, and improve programs

for scientific and algorithmic discovery (Novikov et al., 2025). Generative modeling has also been used to search for mathematical objects and conjecture-relevant structures, suggesting that learned generative models can act as proposal mechanisms for discovery under external evaluators (Ellenberg et al., 2025). MGM differs from these systems in that its search space is not a standalone algorithm or mathematical object, but the executable scaffold of a coding agent itself. Rather than only preserving diverse candidates as archive entries, MGM reuses the archive as a source of comparative diagnostic evidence for inherited self-modification.

B.4 Benchmark Coverage and Evaluation Motivation

SWE-bench is the canonical benchmark for repository-level issue resolution. Unlike function-level benchmarks such as HumanEval (Chen et al., 2021) or MBPP (Austin et al., 2021), SWE-bench gives the model a real repository and a natural-language GitHub issue, and evaluates whether the generated patch passes tests derived from the corresponding pull request (Jimenez et al., 2024). This setting stresses repository navigation, fault localization, patch construction, and validation. SWE-bench Verified further improves reliability by using a human-filtered subset, which is why it has become a standard benchmark for comparing software-engineering agents.

However, SWE-bench Verified alone is not sufficient for evaluating self-evolving agents. First, it is mostly Python-centric, so an evolved agent may overfit to Python idioms, test frameworks, or repository layouts. Second, many tasks are relatively short compared with professional software engineering work. SWE-bench Pro addresses the second limitation by introducing harder long-horizon tasks from a broader range of actively maintained repositories, often requiring deeper investigation and multi-file changes (Deng et al., 2025). Polyglot addresses the first limitation by testing coding across several languages, including C++, Go, Java, JavaScript, Python, and Rust (Gauthier, 2024). Multilingual repository-level variants of SWE-bench provide another related direction by extending issue resolution beyond Python repositories (Khandpur & the SWE-bench Team, 2025; Yang et al., 2025).

These benchmark choices are aligned with MGM's objective. Reaction-norm mutation is intended to identify general weaknesses that recur across tasks, so it should be evaluated on settings where task diversity matters. Cross-lineage hybridization is intended to transfer useful behaviors between agents, so it should be tested on benchmarks where different strategies may solve different subsets of tasks. SWE-bench Verified, SWE-bench Pro, and Polyglot therefore provide complementary evidence: standard real-world issue resolution, long-horizon robustness, and cross-language generality.

C Method Details

C.1 Pseudocode of Mendel Gödel Machine

Here, we present pseudocode of MGM, which helps understand the full evolution pipeline.

Algorithm 1 gives the complete MGM loop. The procedure follows the standard archive-based self-improvement structure: each budgeted step either evaluates an existing agent on a task or expands the archive by self-modifying an agent. Evaluations update both the agent archive and the failed-task pool. During expansion, MGM constructs the set of eligible operators from already collected trajectories and samples one of clonal mutation, reaction-norm mutation, or cross-lineage hybridization. The selected operator determines how evidence is assembled for the editor: from one failed trajectory, multiple trajectories of the same agent, or shared-task trajectories across lineages. Therefore, MGM improves the informativeness of each self-modification step without adding extra task evaluations.

C.2 Primary-Lineage Selection in Cross-Lineage Hybridization

In the main algorithm, the policy π first selects an archive node as an anchor for expansion. For Clonal Mutation and Reaction-norm Mutation, this anchor is also the primary agent whose codebase is edited. Cross-lineage Hybridization differs because the purpose of the operator is to transfer a useful behavior from one lineage to another. Therefore, the anchor node is used to retrieve a comparable peer, but the final primary lineage is determined by the shared-task outcomes.

Algorithm 1: Mendel Gödel Machine

Input: Initial coding agent a_0, benchmark suite $\mathcal{B}$, maximum iterations T, strategy weights $\lambda_A, \lambda_B, \lambda_C$, failed task pool weights β_{fail}

Output: Archive of agents $\mathcal{A}$

$r(a_0, \mathcal{B}) \leftarrow \text{evaluate}(a_0, \mathcal{B})$ // Evaluate the base agent

initialize $\mathcal{A} \leftarrow \{(a_0, r(a_0, \mathcal{B}))\}$ // Start with the base agent

initialize failed-task pool $\mathcal{P} \leftarrow \emptyset$ // Store informative failed tasks

for $t \leftarrow 1$ **to** T **do**

  decide action $u \in \{\text{expand}, \text{evaluate}\}$ // Use Selection Policy

  if $u = \text{expand}$ **then**

    $a' \leftarrow \text{SelectParent}(\mathcal{A})$ // Select a promising lineage

    $\Omega_{a'} \leftarrow \text{EligibleStrategies}(a', \mathcal{A})$ // Find available mutation operators

    $\sigma \leftarrow \text{SampleStrategy}(\Omega_{a'}; \lambda_{\text{CM}}, \lambda_{\text{RM}}, \lambda_{\text{CH}})$ // Choose CM, RM, or CH

    $E_\sigma \leftarrow \text{BuildEvidence}(a', \sigma, \mathcal{A})$ // Construct diagnostic evidence

    $c \leftarrow \text{Edit}_\sigma(a', E_\sigma)$ // Self-modification

    if c.is_valid() **then**

      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(c, \emptyset)\}$ // Keep valid child agent

  else

    $a \leftarrow \text{SelectAgent}(\mathcal{A})$ // Choose agent to test

    $\tau \leftarrow \text{SelectTask}(a, \mathcal{B}, \mathcal{P}, \beta_{\text{fail}})$ // Consider informative failed tasks

    $r(a, \tau) \leftarrow \text{evaluate}(a, \tau)$ // Evaluate on one task

    update $\mathcal{A}$ with $(a, \tau, r(a, \tau))$ // Store trajectory and result

    if $r(a, \tau) = 0$ **then**

      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\tau\}$ // Update failed-task pool

return $\mathcal{A}$

Concretely, suppose the anchor agent a_i and a peer agent a_j have both attempted a shared task $\tau^\star$, and the task is not solved by both agents. MGM constructs the comparison evidence

$$E_C(i, j, \tau^\star) = \{(\varphi(a_i, \tau^\star), r(a_i, \tau^\star)), (\varphi(a_j, \tau^\star), r(a_j, \tau^\star))\}. \tag{23}$$

If exactly one of the two agents solves $\tau^\star$, the failing agent is selected as the primary agent a_p, and the successful agent is used as the context donor a_q. The resulting child is attached to the failing lineage:

$$a' \leftarrow \Phi_C(a_p, E_C),\ a' \in \text{children}(a_p). \tag{24}$$

This direction is intentional: the goal is not to further edit the already successful lineage, but to let the failing lineage inherit a transferable behavior observed in the successful trajectory.

If both agents fail on $\tau^\star$, neither trajectory provides a direct success demonstration. In this case, MGM uses the higher-utility lineage as the primary agent and the other lineage as contrastive context. This fallback preserves the archive policy's preference for promising lineages while still allowing the editor to diagnose complementary failure modes from the same-task comparison.

Thus, the node selected by π should be interpreted as an anchor for constructing a controlled comparison, not necessarily as the final parent edited by CH. The final child is always attached to the primary lineage determined by the CH comparison. This implementation matches the diagnostic role of hybridization: successful trajectories act as donors when available, and failed–failed comparisons are used to identify general weaknesses rather than task-specific patches.

D Theory and Simulations

D.1 Full Additive Fitness Landscape

Section 4 gives the high-level intuition of the controlled surrogate used to isolate the effect of comparative evidence. Here we provide the complete formulation.

Each agent a is represented by a binary genotype

$$g(a) \in \{0, 1\}^L,$$

where each locus corresponds to a minimal scaffold-level capability or implementation choice. A fixed oracle genotype $g^\star \in \{0,1\}^L$ represents the optimal scaffold; without loss of generality, we set $g^\star = \mathbf{1}$.

We measure self-improvement progress by the Hamming distance to the oracle,

$$d(g) = \sum_{\ell=1}^{L} \mathbf{1}[g_\ell \neq g^\star_\ell]. \tag{25}$$

Thus, $d(g) = 0$ if and only if $g = g^\star$. Each simulation run starts from an initial genotype with d_0 mismatched loci.

Define the set of incorrect loci of agent a as

$$M(a) = \{\ell \in [L] : g_\ell(a) \neq g^\star_\ell\}. \tag{26}$$

The task pool contains N tasks. Each task τ examines a subset $R_\tau \subseteq [L]$ of k loci. The task is solved if and only if all required loci are correct:

$$r(a,\tau) = 1 \iff R_\tau \cap M(a) = \emptyset. \tag{27}$$

Under uniform sampling without replacement, an agent at edit distance d therefore has per-task success probability

$$\Pr(r = 1 \mid d) = \frac{\binom{L-d}{k}}{\binom{L}{k}}. \tag{28}$$

At initialization, this becomes

$$\Pr(r = 1 \mid d_0) = \frac{\binom{L-d_0}{k}}{\binom{L}{k}}.$$

This surrogate captures the fact that solving a coding task may require multiple scaffold-level capabilities—such as localization, reasoning, editing, and validation—to be simultaneously correct. The genotype itself is latent: ϕ-evaluation reveals only task-level phenotype outcomes rather than the exact mismatched loci, as illustrated in Figure 3.

D.2 Theoretical Justification of Comparative Fix Probability

This section provides the full derivation for Eq. (21) in Section 4. The goal is to justify why the comparative operators used by MGM can induce a higher effective fix probability than single-trajectory clonal mutation.

Therefore, a failed task only reveals that at least one of its required loci is incorrect:

$$r(a,\tau) = 0 \quad \Longrightarrow \quad R_\tau \cap M(a) \neq \varnothing. \tag{29}$$

We view a self-modification operator Φ_σ as a diagnostic procedure. Given evidence E, it constructs an implicit candidate set $C_\sigma(E) \subseteq [L]$ of loci that may explain the observed failure. The editor then attempts to modify one locus in $C_\sigma(E)$. Suppose that if the selected locus is truly incorrect, the editor repairs it with probability $s \in (0,1]$. Then the effective fix probability of operator σ is

$$p_f^\sigma = s \cdot \Pr_{\ell \sim C_\sigma(E)}[\ell \in M(a)]. \tag{30}$$

Thus, an operator has higher fix probability when its evidence produces a candidate set with a higher density of truly incorrect loci.

Clonal mutation. Clonal mutation uses a single failed trajectory $(\varphi(a,\tau_t), 0)$. Since this evidence only implies that at least one locus in R_{τ_t} is incorrect, the natural candidate set is

$$C_{\mathrm{CM}} = R_{\tau_t}. \tag{31}$$

Let the set of causal defects for the target failure be

$$T_t = R_{\tau_t} \cap M(a), \qquad c = |T_t|. \tag{32}$$

Since $|R_{\tau_t}| = k$, we have

$$p_f^{\mathrm{CM}} = s \cdot \frac{c}{k}. \tag{33}$$

In the sparse-defect case where a failure is caused by one dominant missing capability, $c = 1$, and therefore

$$p_f^{\mathrm{CM}} = \frac{s}{k}. \tag{34}$$

Diagnostic compression. The central effect of comparative evidence can be expressed generally. Suppose an operator σ produces a candidate set satisfying

$$T_t \subseteq C_\sigma(E) \subsetneq R_{\tau_t}. \tag{35}$$

The first inclusion means that the comparison preserves all causal defects responsible for the target failure, while the strict second inclusion means that it removes at least one non-causal candidate.

Let

$$q_\sigma = k - |C_\sigma(E)|, \qquad q_\sigma > 0. \tag{36}$$

Since all c causal defects remain in the compressed candidate set,

$$p_f^\sigma = s \cdot \frac{c}{k - q_\sigma} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{37}$$

Therefore, any sound comparative signal that preserves the true causes while eliminating at least one irrelevant explanation strictly increases the effective fix probability. We next instantiate this condition for the comparison cases used by RM and CH.

Reaction-norm mutation: failed–failed comparison. Reaction-norm mutation compares multiple trajectories of the same genotype. First consider two failed tasks τ_t and τ_r of the same agent a:

$$r(a, \tau_t) = 0, \qquad r(a, \tau_r) = 0. \tag{38}$$

For a sound failed–failed comparison, we assume that the causal defects responsible for the target failure recur in the reference failure:

$$T_t \subseteq R_{\tau_r}. \tag{39}$$

This models the case where repeated failures expose the same scaffold-level weakness rather than unrelated task-specific accidents.

RM can then restrict the explanatory region to

$$C_{\mathrm{RM}}^{\mathrm{F/F}} = R_{\tau_t} \cap R_{\tau_r}. \tag{40}$$

By equation 39,

$$T_t \subseteq C_{\mathrm{RM}}^{\mathrm{F/F}}, \tag{41}$$

so no causal target defect is removed.

For an informative comparison, the two task-relevant sets are not identical:

$$R_{\tau_t} \neq R_{\tau_r}. \tag{42}$$

Since both contain k loci,

$$|R_{\tau_t} \cap R_{\tau_r}| < k. \tag{43}$$

Therefore,

$$p_f^{\mathrm{RM,F/F}} = s \cdot \frac{c}{|R_{\tau_t} \cap R_{\tau_r}|} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{44}$$

In the sparse-defect case, $T_t = \{b\}$, so the sound-comparison condition reduces to

$$b \in R_{\tau_t} \cap R_{\tau_r}. \tag{45}$$

Then

$$p_f^{\mathrm{RM,F/F}} = \frac{s}{|R_{\tau_t} \cap R_{\tau_r}|} > \frac{s}{k} = p_f^{\mathrm{CM}}. \tag{46}$$

Thus, failed–failed RM improves fix probability when the two failures share the causal weakness of interest while differing in some task-specific requirements.

Reaction-norm mutation: failed–successful comparison. RM may also compare a failed target task τ_t with a successful reference task τ_r from the same agent:

$$r(a, \tau_t) = 0, \qquad r(a, \tau_r) = 1. \tag{47}$$

Because the genotype is identical across the two tasks, the successful reference implies

$$R_{\tau_r} \cap M(a) = \varnothing. \tag{48}$$

Hence every locus shared by the target and reference task is known not to be a causal defect:

$$R_{\tau_t} \cap R_{\tau_r} \subseteq R_{\tau_t} \setminus T_t. \tag{49}$$

RM can therefore safely remove this overlap:

$$C_{\mathrm{RM}}^{\mathrm{F/S}} = R_{\tau_t} \setminus R_{\tau_r}. \tag{50}$$

Since the successful reference contains no incorrect locus,

$$T_t \subseteq C_{\mathrm{RM}}^{\mathrm{F/S}}. \tag{51}$$

Let

$$q_{\mathrm{RM}}^{\mathrm{F/S}} = |R_{\tau_t} \cap R_{\tau_r}|. \tag{52}$$

Whenever the two tasks share at least one locus,

$$q_{\mathrm{RM}}^{\mathrm{F/S}} > 0, \tag{53}$$

the candidate set is strictly smaller than the original failed-task region:

$$|C_{\mathrm{RM}}^{\mathrm{F/S}}| = k - q_{\mathrm{RM}}^{\mathrm{F/S}} < k. \tag{54}$$

Therefore,

$$p_f^{\mathrm{RM,F/S}} = s \cdot \frac{c}{k - q_{\mathrm{RM}}^{\mathrm{F/S}}} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{55}$$

Thus, a successful trajectory of the same genotype provides negative diagnostic evidence: task-relevant loci already known to be correct can be excluded from the explanation of the target failure. If the two tasks have no task-relevant overlap, the comparison removes no candidate and provides no strict fix-probability advantage.

Cross-lineage hybridization. Cross-lineage hybridization compares different genotypes under the same task. Let a_t denote the failing target agent and a_r a reference agent evaluated on the same task τ. The target failure implies

$$T_t = R_\tau \cap M(a_t) \neq \varnothing, \tag{56}$$

with

$$c = |T_t|. \tag{57}$$

Clonal mutation must search over the entire task-relevant region:

$$p_f^{\mathrm{CM}} = s \cdot \frac{c}{k}. \tag{58}$$

Unlike RM, the compared trajectories come from different genotypes. Therefore, the reference outcome alone does not directly determine which loci are correct in the target agent. Instead, CH relies on trajectory-level contrastive evidence to eliminate explanations that are inconsistent with the behavioral difference between the two lineages.

Let

$$Q_{\mathrm{CH}}(E) \subseteq R_\tau \setminus T_t \tag{59}$$

denote the set of non-causal target candidates that can be soundly excluded using the reference trajectory. The resulting candidate set is

$$C_{\mathrm{CH}}(E) = R_\tau \setminus Q_{\mathrm{CH}}(E). \tag{60}$$

For an informative comparison,

$$|Q_{\mathrm{CH}}(E)| > 0. \tag{61}$$

Because only non-causal candidates are removed,

$$T_t \subseteq C_{\mathrm{CH}}(E). \tag{62}$$

Consequently,

$$p_f^{\mathrm{CH}} = s \cdot \frac{c}{k - |Q_{\mathrm{CH}}(E)|} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{63}$$

We next distinguish the two possible outcome configurations.

Cross-lineage hybridization: failed target and successful reference. Consider

$$r(a_t, \tau) = 0, \qquad r(a_r, \tau) = 1. \tag{64}$$

The reference success implies

$$R_\tau \cap M(a_r) = \varnothing. \tag{65}$$

Thus, the reference provides a successful behavioral control for the same task. Comparing the two trajectories can identify behaviors or components that are shared with the successful reference and are therefore unlikely to explain the target-specific failure.

Let

$$Q_{\mathrm{CH}}^{\mathrm{F/S}} \subseteq R_\tau \setminus T_t \tag{66}$$

be the set of such soundly excluded candidates. If

$$q_{\mathrm{CH}}^{\mathrm{F/S}} = |Q_{\mathrm{CH}}^{\mathrm{F/S}}| > 0, \tag{67}$$

then

$$C_{\mathrm{CH}}^{\mathrm{F/S}} = R_\tau \setminus Q_{\mathrm{CH}}^{\mathrm{F/S}}, \tag{68}$$

and therefore

$$p_f^{\mathrm{CH,F/S}} = s \cdot \frac{c}{k - q_{\mathrm{CH}}^{\mathrm{F/S}}} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{69}$$

The strict improvement does not follow from reference success alone; it follows when the trajectory comparison uses that successful control to soundly eliminate at least one non-causal explanation of the target failure.

Cross-lineage hybridization: failed–failed comparison. CH may also compare two agents that both fail the same task:

$$r(a_t, \tau) = 0, \qquad r(a_r, \tau) = 0. \tag{70}$$

The two failure labels alone do not provide a strict advantage, since the two agents may fail for exactly the same reason. However, the full trajectories can expose complementary failure modes. For example, the reference agent may correctly execute an intermediate behavior on which the target agent fails, even though the reference later fails elsewhere.

Let

$$Q_{\mathrm{CH}}^{\mathrm{F/F}}(E) \subseteq R_\tau \setminus T_t \tag{71}$$

denote the non-causal candidates that this trajectory-level comparison can soundly eliminate. For an informative comparison,

$$q_{\mathrm{CH}}^{\mathrm{F/F}} = |Q_{\mathrm{CH}}^{\mathrm{F/F}}(E)| > 0. \tag{72}$$

The resulting candidate set is

$$C_{\mathrm{CH}}^{\mathrm{F/F}} = R_\tau \setminus Q_{\mathrm{CH}}^{\mathrm{F/F}}(E), \tag{73}$$

and still satisfies

$$T_t \subseteq C_{\mathrm{CH}}^{\mathrm{F/F}}. \tag{74}$$

Hence

$$p_f^{\mathrm{CH,F/F}} = s \cdot \frac{c}{k - q_{\mathrm{CH}}^{\mathrm{F/F}}} > s \cdot \frac{c}{k} = p_f^{\mathrm{CM}}. \tag{75}$$

Thus, failed–failed CH can also improve self-modification when the trajectory contrast reveals at least one non-causal explanation that can be discarded without removing the true cause of the target failure.

Summary. The four cases above share the same underlying mechanism. For $\sigma \in \{\mathrm{RM}, \mathrm{CH}\}$, if the comparative evidence satisfies

$$T_t \subseteq C_\sigma(E) \subsetneq R_{\tau_t}, \tag{76}$$

then

$$p_f^\sigma > p_f^{\mathrm{CM}}. \tag{77}$$

Accordingly, under sound and informative comparative evidence,

$$p_f^{\mathrm{RM}} > p_f^{\mathrm{CM}}, \qquad p_f^{\mathrm{CH}} > p_f^{\mathrm{CM}}. \tag{78}$$

Discussion of assumptions. The above derivation does not claim that comparative operators are unconditionally superior. Strict improvement requires the comparative evidence to be both *sound*, meaning that it does not eliminate any causal target defect, and *informative*, meaning that it eliminates at least one non-causal candidate.

For failed–failed RM, this corresponds to recurring causal weaknesses across tasks. For failed–successful RM, success of the same genotype provides direct evidence that overlapping loci are not defective. For CH, because the compared trajectories come from different genotypes, both successful-reference and failed–failed comparisons require the trajectory-level contrast to identify non-causal target explanations reliably. Finally, the editor must be capable of exploiting this comparative evidence.

When a comparison is sound but removes no irrelevant candidate, $C_\sigma(E) = R_{\tau_t}$ and the operator reduces to the clonal fix probability. If the comparative evidence is misleading and removes a causal defect, no improvement guarantee follows. The null setting $\rho = 1$ in our simulation represents the case where comparative operators provide no effective fix-quality advantage.

D.3 Monte Carlo Protocol and Parameters

The theoretical analysis above characterizes how comparative evidence can increase the effective fix probability through diagnostic compression. In the Monte Carlo study, we abstract this mechanism into a controlled operator-level fix probability p_f^σ, rather than explicitly simulating the candidate-set construction at each edit.

Section 4 reports the resulting convergence and final performance in Figure 4 and Figure 5. Here we specify the complete simulation protocol.

Method instantiation. The simulation isolates the axes on which DGM, HGM, and MGM differ. DGM allocates a fixed n_{eval} evaluations to each population member per generation, distributing evaluation budget uniformly across the population. HGM instead uses adaptive allocation, concentrating evaluations on promising nodes and triggering edits according to its tree-search policy. Both DGM and HGM use only clonal mutation Φ_{CM}. MGM follows the same evaluation–expansion framework as HGM but additionally uses Φ_{RM} and Φ_{CH} whenever their eligibility conditions are satisfied.

Edit quality and cost control. To isolate diagnostic quality rather than unequal compute, all task evaluations and self-modification operators are assigned equal unit cost:

$$c_\tau = c_{\text{CM}} = c_{\text{RM}} = c_{\text{CH}} = 1. \tag{79}$$

The clonal fix probability is p_f^{CM}, while the comparative operators use $p_f^{\text{RM}} = \rho p_f^{\text{CM}}$ and $p_f^{\text{CH}} = \rho p_f^{\text{CM}}$. All operators share the same break probability p_b, which models the possibility that an edit corrupts an otherwise correct component. Thus ρ controls only the diagnostic advantage of comparative evidence.

Monte Carlo evaluation. Each method runs under a fixed total budget B for n_{seeds} independent Monte Carlo seeds. For every seed, we regenerate both the initial genotype and the task pool. At evenly spaced budget checkpoints, we record the minimum edit distance among all active nodes in the archive. Final performance and convergence curves are aggregated across seeds with 95% confidence intervals.

We sweep the initial edit distance $d_0 \in \{10, 20, 40, 80\}$ and diagnostic-advantage ratio $\rho \in \{1.0, 1.2, 1.5, 2.0\}$. The $\rho = 1$ condition is the null setting in which comparative operators have no fix-quality advantage over clonal mutation. All remaining parameters are listed in Table 7.

E Experiment Details

E.1 Compute and Software Environment

All experiments were run on cluster nodes with eight NVIDIA H100 GPUs (80 GB GPU memory each) and approximately 2 TB of host memory. The Qwen models used in our experiments[2] was served with vLLM through an OpenAI-compatible API using tensor parallelism across the visible GPUs. We set the maximum context length to 262,144 tokens and the GPU memory utilization

[2] `https://huggingface.co/Qwen/Qwen3.6-35B-A3B` and `https://huggingface.co/Qwen/Qwen3-Coder-Next`

Type	Parameter	Symbol	Value
Genotype & Phenotype	Chain length	L	100
	Initial edit distance	d_0	$\{10, 20, 40, 80\}$
	Task pool size	N	200
	Loci examined per task	k	5
Costs	Task evaluation	c_τ	1.0
	Φ_{CM} edit	c_{CM}	1.0
	Φ_{RM} edit	c_{RM}	1.0
	Φ_{CH} edit	c_{CH}	1.0
Edit quality	Fix prob. (Φ_{CM})	p_f^{CM}	0.25
	Fix prob. (Φ_{RM})	p_f^{RM}	$\rho \cdot \Phi_{\mathrm{CM}}$
	Fix prob. (Φ_{CH})	p_f^{CH}	$\rho \cdot \Phi_{\mathrm{CM}}$
	Fix-ratio	ρ	$\{1.0, 1.2, 1.5, 2.0\}$
	Break prob. (all Φ operators)	p_b	0.05
DGM	Population size	n_{pop}	5
	Evals per node	n_{eval}	10
	Selection fraction	s_{frac}	0.4
HGM	Min evals before edit	n_{min}	3
	Max evals before edit	n_{max}	20
	Failure threshold	τ_{edit}	0.5
MGM	$\Pr(\Phi_{\mathrm{RM}} \mid \text{available})$	π_{RM}	0.45
	$\Pr(\Phi_{\mathrm{CH}} \mid \text{available})$	π_{CH}	0.45
	Min tasks for Φ_{RM}	m_{RM}	2
Monte Carlo	Total budget	B	500
	Seeds	n_{seeds}	100
	Budget checkpoints	n_{chk}	500

Table 7: **Parameter settings for simulation.** All edit costs are equal so that the HGM–MGM comparison isolates diagnostic quality alone. The fix-probability advantage ratio for the main simulation is $\rho = p_f^{\mathrm{RM}}/p_f^{\mathrm{CM}} = 2.0$; the sweep grid varies ρ and d_0 to test robustness.

Hyper-parameter	Value
β_{fail}	1.0
λ_{CM}	0.10
λ_{RM}	0.45
λ_{CH}	0.45
m_{RM}	2

Table 8: **Hyper-parameter settings used in all experiments.**

to 92%. Benchmark tasks were executed inside Apptainer containers built from a Python 3.10 base image, with the repository bind-mounted into the container. By default, host networking was disabled, and the container addressed the vLLM endpoint through the host node IP. For DeepSeek-V4 models, we directly accessed their preview release via the official DeepSeek API[3]. For generation, the Qwen models were run with thinking enabled, a temperature of 1.0, top- p of 0.95, and top-k of 20, following the sampling configuration loaded by vLLM from the models' generation configuration. No explicit reasoning-effort level or thinking-token budget was imposed. The DeepSeek-V4 models were evaluated in non-thinking mode, with the API-default temperature and top-p values of 1.0. Parallel tool calls were disabled for both model families.

E.2 Hyper-parameter Settings

In this section, to improve transparency and reproducibility, we present the hyper-parameter settings used in our code implementations, as shown in Table 8.

[3] `https://api-docs.deepseek.com/news/news260424/`

F BENCHMARK DETAILS

To control evaluation cost, instead of evaluating MGM on the full SWE-bench Pro and SWE-bench Multilingual, we follow the procedures by Zhang et al. (2026a), randomly and separately choose 60 representative tasks for each benchmark.

For SWE-bench Pro, the selective 60-task subset is required to be representative and involve all the six Programming Languages, `JavaScript`, `Python`, `Java`, `C++`, `TypeScript`, `Go`. Here we report the 60 tasks we used for our SWE-bench Pro evaluation:

- `ansible__ansible-0ea40e09d1b35bcb69ff4d9cecf3d0defa4b36e8`
- `ansible__ansible-189fcb37f973f0b1d52b555728208eeb9a6fce83`
- `ansible__ansible-3889ddeb4b780ab4bac9ca2e75f8c1991bcabe83`
- `ansible__ansible-5260527c4a71bfed99d803e687dd19619423b134`
- `ansible__ansible-a20a52701402a12f91396549df04ac55809f68e9`
- `internetarchive__openlibrary-308a35d6999427c02b1dbf5211c033ad3b352556`
- `internetarchive__openlibrary-30bc73a1395fba2300087c7f307e54bb5372b60a`
- `internetarchive__openlibrary-4b7ea2977be2747496ba792a678940baa985f7ea`
- `internetarchive__openlibrary-7edd1ef09d91fe0b435707633c5cc9af41dedddf`
- `internetarchive__openlibrary-9bdfd29fac883e77dcbc4208cab28c06fd963ab2`
- `qutebrowser__qutebrowser-66cfa15c372fa9e613ea5a82d3b03e4609399fb6`
- `qutebrowser__qutebrowser-6b320dc18662580e1313d2548fdd6231d2a97e6d`
- `qutebrowser__qutebrowser-8cd06741bb56cdca49f5cdc0542da97681154315`
- `qutebrowser__qutebrowser-8f46ba3f6dc7b18375f7aa63c48a1fe461190430`
- `qutebrowser__qutebrowser-99029144b5109bb1b2a53964a7c129e009980cd9`
- `flipt-io__flipt-02e21636c58e86c51119b63e0fb5ca7b813b07b1`
- `flipt-io__flipt-5aef5a14890aa145c22d864a834694bae3a6f112`
- `flipt-io__flipt-9d25c18b79bc7829a6fb08ec9e8793d5d17e2868`
- `flipt-io__flipt-b68b8960b8a08540d5198d78c665a7eb0bea4008`
- `flipt-io__flipt-e2bd19dafa7166c96b082fb2a59eb54b4be0d778`
- `future-architect__vuls-78b52d6a7f480bd610b692de9bf0c86f57332f23`
- `future-architect__vuls-86b60e1478e44d28b1aff6b9ac7e95ceb05bc5fc`
- `future-architect__vuls-e049df50fa1eecdccc5348e27845b5c783ed7c76`
- `gravitational__teleport-3fa6904377c006497169945428e8197158667910`
- `gravitational__teleport-3ff75e29fb2153a2637fe7f83e49dc04b1c99c9f`
- `gravitational__teleport-73cc189b0e9636d418c4470ecce0d9af5dae2f02`
- `gravitational__teleport-ba6c4a135412c4296dd5551bd94042f0dc024504`
- `navidrome__navidrome-66b74c81f115c78cb69910b0472eeb376750efc4`
- `navidrome__navidrome-812dc2090f20ac4f8ac271b6ed95be5889d1a3ca`
- `navidrome__navidrome-c90468b895f6171e33e937ff20dc915c995274f0`
- `element-hq__element-web-1077729a19c0ce902e713cf6fab42c91fb7907f1`
- `element-hq__element-web-41dfec20bfe9b62cddbbbf621bef2e9aa9685157`
- `element-hq__element-web-53a9b6447bd7e6110ee4a63e2ec0322c250f08d1`
- `element-hq__element-web-9a31cd0fa849da810b4fac6c6c015145e850b282`
- `element-hq__element-web-b007ea81b2ccd001b00f332bee65070aa7fc00f9`
- `NodeBB__NodeBB-397835a05a8e2897324e566b41c5e616e172b4af`
- `NodeBB__NodeBB-51d8f3b195bddb13a13ddc0de110722774d9bb1b`
- `NodeBB__NodeBB-97c8569a798075c50e93e585ac741ab55cb7c28b`

- NodeBB__NodeBB-be43cd25974681c9743d424238b7536c357dc8d3
- NodeBB__NodeBB-f48ed3658aab7be0f1165d4c1f89af48d7865189
- protonmail__webclients-01b519cd49e6a24d9a05d2eb97f54e420740072e
- protonmail__webclients-08bb09914d0d37b0cd6376d4cab5b77728a43e7b
- protonmail__webclients-51742625834d3bd0d10fe0c7e76b8739a59c6b9f
- protonmail__webclients-6f8916fbadf1d1f4a26640f53b5cf7f55e8bedb7
- protonmail__webclients-8142704f447df6e108d53cab25451c8a94976b92
- tutao__tutanota-09c2776c0fce3db5c6e18da92b5a45dce9f013aa
- tutao__tutanota-12a6cbaa4f8b43c2f85caca0787ab55501539955
- tutao__tutanota-1e516e989b3c0221f4af6b297d9c0e4c43e4adc3
- tutao__tutanota-1ff82aa365763cee2d609c9d19360ad87fdf2ec7
- tutao__tutanota-219bc8f05d7b980e038bc1524cb021bf56397a1b
- tutao__tutanota-40e94dee2bcec2b63f362da283123e9df1874cc1
- tutao__tutanota-4b4e45949096bb288f2b522f657610e480efa3e8
- tutao__tutanota-51818218c6ae33de00cbea3a4d30daac8c34142e
- tutao__tutanota-8513a9e8114a8b42e64f4348335e0f23efa054c4
- tutao__tutanota-b4934a0f3c34d9d7649e944b183137e8fad3e859
- tutao__tutanota-d1aa0ecec288bfc800cfb9133b087c4f81ad8b38
- tutao__tutanota-db90ac26ab78addf72a8efaff3c7acc0fbd6d000
- tutao__tutanota-de49d486feef842101506adf040a0f00ded59519
- tutao__tutanota-fb32e5f9d9fc152a00144d56dd0af01760a2d4dc
- tutao__tutanota-fe240cbf7f0fdd6744ef7bef8cb61676bcdbb621

SWE-bench Multilingual contains 300 tasks from 42 repositories and spans nine programming languages: C, C++, Go, Java, JavaScript, TypeScript, PHP, Ruby, and Rust. We construct the subset to cover these language groups while keeping the evaluation cost manageable. Here we report the exact tasks used in our SWE-bench Multilingual evaluation below.

- apache__druid-14092
- apache__lucene-13494
- apache__lucene-13704
- astral-sh__ruff-15356
- astral-sh__ruff-15443
- axios__axios-4731
- babel__babel-16130
- briannesbitt__carbon-3005
- briannesbitt__carbon-3041
- briannesbitt__carbon-3103
- caddyserver__caddy-4943
- caddyserver__caddy-5404
- caddyserver__caddy-5870
- caddyserver__caddy-5995
- facebook__docusaurus-9183
- fastlane__fastlane-19207
- fastlane__fastlane-20642
- fastlane__fastlane-20975
- fluent__fluentd-3917
- fmtlib__fmt-2317

- fmtlib__fmt-2457
- gohugoio__hugo-12579
- google__gson-1014
- immutable-js__immutable-js-2006
- jekyll__jekyll-8771
- jqlang__jq-2235
- jqlang__jq-2658
- jqlang__jq-2839
- jqlang__jq-2919
- laravel__framework-51195
- laravel__framework-53914
- laravel__framework-53949
- nushell__nushell-13605
- php-cs-fixer__php-cs-fixer-7635
- phpoffice__phpspreadsheet-3570
- phpoffice__phpspreadsheet-4114
- preactjs__preact-2757
- preactjs__preact-3454
- preactjs__preact-3562
- preactjs__preact-4436
- projectlombok__lombok-3009
- projectlombok__lombok-3350
- projectlombok__lombok-3422
- projectlombok__lombok-3479
- projectlombok__lombok-3594
- prometheus__prometheus-10633
- prometheus__prometheus-12874
- prometheus__prometheus-14861
- redis__redis-11734
- redis__redis-13115
- rubocop__rubocop-13375
- rubocop__rubocop-13479
- rubocop__rubocop-13627
- sharkdp__bat-2650
- tokio-rs__axum-1730
- tokio-rs__tokio-6752
- tokio-rs__tokio-6838
- uutils__coreutils-6575
- uutils__coreutils-6682
- vuejs__core-11739

G Additional Results

G.1 Full Evaluation on Polyglot

To verify that the Polyglot improvement reported in the main experiments is not an artifact of the 60-task subset, we additionally evaluate the best MGM-discovered agent on the full Polyglot benchmark. This evaluation is

performed only after evolution is complete: the agent is not further updated, and the full benchmark is used only for post-evolution evaluation and is not used to further update the agent.

As shown in Figure 9, the MGM-evolved agent solves 210 out of 225 tasks, achieving an overall accuracy of ***93.3%***. This closely matches the 93.2% accuracy observed on Polyglot-60 in the main experiment, suggesting that the improvement is stable when moving from the subset evaluation to the full benchmark. The gains are also broadly distributed across languages: the agent solves 25/26 C++ tasks, 36/39 Go tasks, 43/47 Java tasks, 46/49 JavaScript tasks, 33/34 Python tasks, and 27/30 Rust tasks.

These results support the claim that MGM learns reusable, language-agnostic workflow improvements rather than narrow heuristics tied to a particular language or subset. In particular, the consistently high resolution rate across C++, Go, Java, JavaScript, Python, and Rust is consistent with the role of reaction-norm mutation and cross-lineage hybridization: the former identifies recurring agent-level weaknesses across tasks, while the latter transfers useful behavioral traits across lineages. The remaining failures are not concentrated in a single language, indicating that future improvements should likely target harder residual failure modes rather than language-specific specialization.

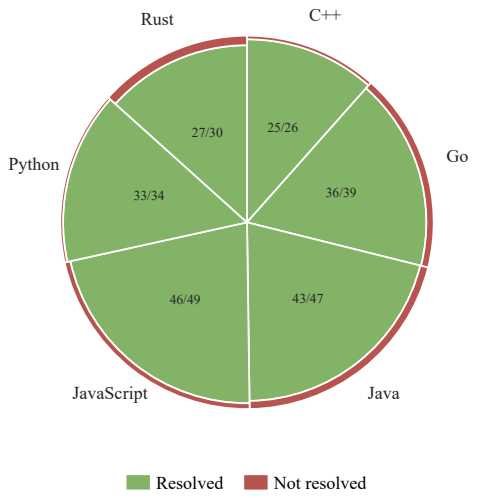


Figure 9: **Polyglot-225 performance of the MGM-discovered agent by language**. The agent is evaluated after evolution without further self-modification and solves 210 out of 225 tasks overall. The area of pie slices report the proportion of resolved and unresolved tasks within each language, with labels showing the ratio of resolved to total tasks. MGM maintains high accuracy across C++, Go, Java, JavaScript, Python, and Rust, indicating that the evolved improvements transfer across languages rather than specializing to a single language subset.

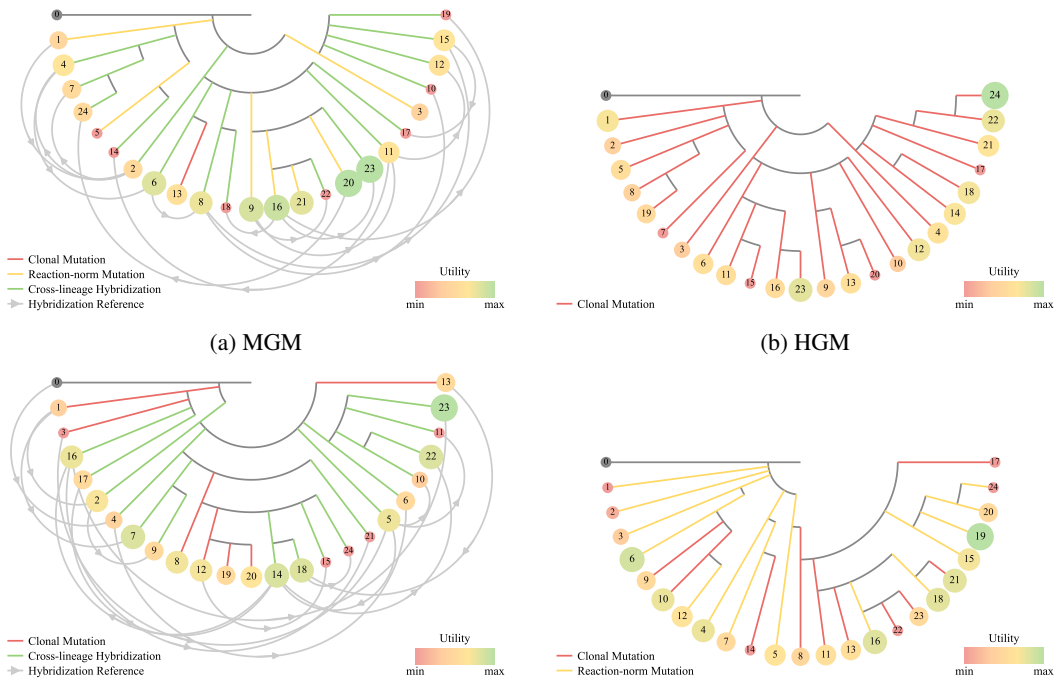
G.2 Visualizations of Evolution Trees

All four figures use the same visual encoding: node fill color denotes evaluation accuracy, while edge color denotes the child node's self-improvement operator ($\Phi_{\mathrm{CM}}/\Phi_{\mathrm{RM}}/\Phi_{\mathrm{CH}}$). Starred nodes mark the representative agent chosen for each experimental setting.

Figure 10a shows the evolution tree produced by the full MGM system under a budget of 200 task evaluations, yielding 24 nodes. All three operators are active: Clonal Mutation, Reaction-norm Mutation, and Cross-lineage Hybridization. Node fill color encodes evaluation accuracy. The node #20 and #23 has the highest utility 1.00 but only with 15 and 4 evaluations, respectively. The node #16 has utility 0.91 after 35 evaluations and is selected as the final result.

Figure 10b shows the evolution tree of the HGM baseline, which uses Clonal Mutation as its only self-modification operator. Under the same 200-evaluation budget, HGM also produces 24 nodes, but all child edges correspond to single-trajectory clonal edits. Compared with full MGM, HGM still explores multiple branches, but it lacks the additional comparative evidence channels represented by edges with different colors. Although node #24 has the highest utility 1.00, it has only been evaluated 5 times, so its color reflects a high-variance estimate rather than a reliable final selection. The node #18 is selected as result, with utility 0.71 after 14 evaluations.

Figure 10c shows the ablation that removes Reaction-norm Mutation. Clonal Mutation and Cross-lineage Hybridization remain enabled. The node #23 with highest utility 1.00 got only 2 evaluations, and the node #14 with utility 0.84 after 50 evaluations is selected as result.



(a) MGM

(b) HGM

(c) w/o Φ_{RM} (Φ_{CM} and Φ_{CH} only)

(d) w/o Φ_{CH} (Φ_{CM} and Φ_{RM} only)

Figure 10: **Evolution trees of MGM, HGM, and MGM's ablated variants.** Results after 200 φ-evaluations and 24 Φ-expansions on Polyglot. Nodes are independently colored by utility estimates aggregated over each tree's φ-evaluation results.

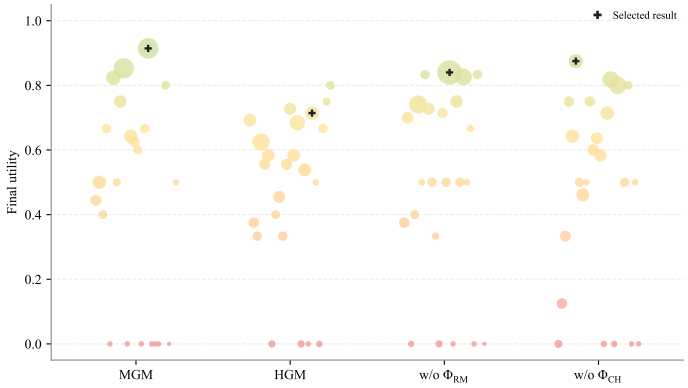


Figure 11: **Final per-node utilities for MGM, HGM, and the ablations.** Each point is one of the 24 evolved nodes. Vertical position and fill color encode utility. Marker area is proportional to the number of evaluations. The marked node in each column is the selected final result.

Figure 10d shows the ablation that removes Cross-lineage Hybridization. Clonal Mutation and Reaction-norm Mutation remain active. All improvement is confined to within-lineage evolution. The node #19 has utility 1.00 after only 2 evaluations, and the selected result is node #6 with utility 0.88, after 16 evaluations.

In Figure 11 we additionally report the final utilities of all 24 nodes under each configuration. Point size reflects evaluation count, making high-utility nodes with few measurements visually distinct from better-supported estimates. The raw maximum is often attained by a sparsely evaluated node, whereas the selected result favors a high utility backed by more evaluations.

H Best Discovered Agents

H.1 MGM on Polyglot

Compared with the initial agent, whose `forward()` routine performs a single-turn code generation call without structured repository analysis or test feedback, agent (node #16) accumulates three successive modifications along the lineage *initial* $\rightarrow$ #2 $\rightarrow$ #9 $\rightarrow$ #16. The first patch (#2, introduced via cross-lineage hybridization) adds a read-first prompt directive that requires the agent to inspect stubs, tests, and class hierarchies with the `editor` tool and to implement only the interfaces already defined in the repository. The second patch (#9, a reaction-norm mutation) replaces the single-turn workflow with a two-phase pipeline: in Phase 1, the agent extracts a structured `contract_plan` JSON containing exact class signatures, constructors, method definitions, and verbatim error messages from test and stub files. In Phase 2, it implements the solution under strict adherence to that plan and performs an explicit checklist-based self-audit before submission. The third patch (#16, also a reaction-norm mutation) extends this design with a test-driven verification loop: after implementation, the agent repeatedly executes the task's test suite (via a new `run_tests()` method and supporting utilities in `utils/test_utils.py`), feeds raw test output back to the model together with the remaining contract, and iteratively repairs the solution for up to five attempts until tests pass or the budget is exhausted. Cumulatively, these changes transform the initial agent from a prompt-only code generator into a test-contract-guided, self-auditing, and empirically self-correcting coding agent capable of reducing API hallucination, enforcing exact test constraints, and recovering from runtime failures through language-agnostic test feedback.

H.2 MGM on SWE-bench Verified

Listing below shows an example self-modification discovered by MGM on SWE-bench Verified. Unlike a task-specific repository patch, this modification changes the agent's general debugging workflow. The evolved scaffold turns the original single-pass repair process into a more trace-aware and patch-constrained procedure.

Concretely, the agent first extracts test function names from the problem statement and test description, and then builds an explicit code-path trace before editing. When relevant failing tests are available, the scaffold generates a `code_path_trace` that links the failing test to the likely implementation path and provides a targeted fix direction. After the initial repair attempt, the agent further inspects the message history to identify remaining failed tests and can perform additional revision rounds conditioned on the traced failure path. This makes the repair process less dependent on a vague natural-language issue description and more directly grounded in executable regression signals.

The modification also introduces lightweight provenance tracking for repository exploration. By registering callbacks around the editor tool, the agent records which files were actually viewed during the debugging process. This information is then used by a new diff-minimality filter, which removes patch blocks that are weakly related to the problem statement or to the files inspected by the agent. As a result, the evolved scaffold encourages localized fixes and discourages broad, accidental, or speculative edits.

The concrete utility added in this example focuses on tracing docstring and autodoc-style failures, but the underlying scaffold-level skill is more general: MGM discovers a workflow that first grounds the repair in failing tests, then traces the relevant code path, and finally constrains the submitted diff to files supported by the agent's own investigation. This provides qualitative evidence that MGM can evolve reusable repository-level debugging habits on SWE-bench Verified, rather than merely memorizing a solution to a single benchmark instance.

Figure 17: Code modification on Polyglot

```
diff --git a/coding_agent.py b/coding_agent.py
index 39cb47e..d824c3d 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -172,7 +172,21 @@ class AgenticSystem:

 Your task is to make changes to the files in the {self.git_dir} directory to address the <
     problem_description>. I have already taken care of the required dependencies.
 """
-        instruction = f"{task}\n\nPlease analyze the problem description carefully. Then make edits to the
     code files to complete the instruction."
+        inspection_directive = """
+**CRITICAL PRE-IMPLEMENTATION REQUIREMENT:**
+
+Before writing any code, carefully inspect the existing repository structure. Use the `editor` tool to `view
     ` all relevant stub files, test files, and related classes. Extract and strictly adhere to their exact
     method signatures, class hierarchies, and constructor requirements. Do not assume or invent interfaces;
      implement exactly what the existing code expects.
+
+**Specific Instructions:**
+1. First, explore the repository structure using `view` commands to understand the file layout
+2. Read all stub files, interface definitions, and test files to understand the expected API
+3. Summarize the required method signatures, constructor parameters, and class hierarchies
+4. Only after thoroughly understanding the existing code structure, proceed with your implementation
+5. Implement exactly what the stubs and tests expect - never invent new interfaces
+
+**Reminder:** The coding agent needs to deal with different languages including C++, Go, Java, JavaScript,
     Python, and Rust. Adhere to the exact method signatures, class hierarchies, and constructor
     requirements defined in the existing code, regardless of programming language.
+"""
+        instruction = f"{task}\n\n{inspection_directive}\nPlease analyze the problem description carefully.
     Then make edits to the code files to complete the instruction."
         chat_history, n_llm_calls_used = chat_with_agent(
             instruction,
             model=self.code_model,

diff --git a/coding_agent.py b/coding_agent.py
index d824c3d..25f5411 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -1,6 +1,7 @@
 # This file is adapted from https://github.com/jennyzzt/dgm.

 import argparse
+import json
 import logging
 import os
 import subprocess
@@ -160,41 +161,242 @@ class AgenticSystem:

         return new_msg_history

+    def extract_json_from_response(self, response_content):
+        """
+        Extract JSON content from LLM response which may contain markdown or other text.
+        """
+        # Try to find JSON in code blocks
+        import re
+
+        json_pattern = r"```(?:json)?\s*(\{.*?\})\s*```"
+        match = re.search(json_pattern, response_content, re.DOTALL)
+        if match:
+            try:
+                return json.loads(match.group(1))
+            except json.JSONDecodeError:
+                pass
+
+        # Try to find a JSON object directly
+        try:
+            # Find the first opening brace and last closing brace
+            start = response_content.find("{")
+            end = response_content.rfind("}") + 1
+            if start >= 0 and end > start:
+                json_str = response_content[start:end]
+                return json.loads(json_str)
+        except (json.JSONDecodeError, ValueError):
+            pass
+
+        # If all else fails, return the raw content
```

```
+        return response_content
+
     def forward(self, timeout):
         """
         The forward function for the AgenticSystem.
+        Implements a two-phase workflow:
+        1. Contract Extraction Phase: Agent reads test/stub files and outputs a structured contract_plan
     JSON
+        2. Implementation & Verification Phase: Agent implements solution adhering to the plan with self-
    audit
         """
-        task = f"""I have uploaded a code repository in the directory {self.git_dir}. Help solve the
     following problem.
+        # Phase 1: Contract Extraction
+        contract_plan_phase_instruction = f"""I have uploaded a code repository in the directory {self.
    git_dir}. Help solve the following problem.

 <problem_description>
 {self.problem_statement}
 </problem_description>

-Your task is to make changes to the files in the {self.git_dir} directory to address the <
    problem_description>. I have already taken care of the required dependencies.
+# PHASE 1: CONTRACT EXTRACTION
+
+Your FIRST task is to extract the exact API contract from the repository. Do NOT write any implementation
    code in this phase.
+
+**Instructions for Phase 1:**
+
+1. **Explore the repository structure** using the `editor` tool to understand the file layout.
+
+2. **Read ALL relevant files**:
+   - Test files (files containing tests that define expected behavior)
+   - Stub/Interface files (files containing class definitions, method signatures, type hints)
+   - Any specification or documentation files
+
+3. **Extract the exact contract** which includes:
+   - **Class names and their exact hierarchy** (parent classes, interfaces implemented)
+   - **Constructor signatures** (exact parameter names, types, and order)
+   - **Method signatures** (exact parameter names, types, return types)
+   - **Error messages** (verbatim error strings that must be raised)
+   - **Module-level functions** (exact signatures if applicable)
+   - **File paths** that need modification
+
+4. **Output your findings as a JSON object** with the following structure:
+
+```json
+{{
+  "classes": [
+    {{
+      "name": "ClassName",
+      "parent": "ParentClassName or null",
+      "fields": [
+        {{"name": "field_name", "type": "Type"}}
+      ],
+      "methods": [
+        {{
+          "name": "method_name",
+          "parameters": [
+            {{"name": "param", "type": "Type", "required": true}}
+          ],
+          "return_type": "ReturnType",
+          "is_constructor": false
+        }}
+      ],
+      "file_path": "path/to/file"
+    }}
+  ],
+  "functions": [
+    {{
+      "name": "function_name",
+      "parameters": [
+        {{"name": "param", "type": "Type", "required": true}}
+      ],
+      "return_type": "ReturnType",
+      "file_path": "path/to/file"
+    }}
+  ],
+  "error_messages": [
+    {{"message": "exact error string from tests", "context": "when this error is raised"}}
```

```
+  ],
+  "files_to_modify": ["path/to/file1", "path/to/file2"],
+  "test_files": ["path/to/test_file1", "path/to/test_file2"],
+  "key_observations": [
+    "Observation 1: e.g., Constructor takes no arguments",
+    "Observation 2: e.g., Error 'NotImplementedError' must be raised with message 'feature not implemented'"
+  ]
+}}
+```
+
+**CRITICAL REQUIREMENTS FOR PHASE 1:**
+- Extract signatures EXACTLY as defined in the test/stub files
+- Do NOT infer or guess missing information - if something is unclear, note it in key_observations
+- Preserve exact error message strings (case-sensitive, including punctuation)
+- Note the EXACT file paths that contain the contracts
+- This phase is for ANALYSIS ONLY - do not write any implementation code
+
+Output only the JSON object and a brief summary of your findings. No implementation code should be generated in this phase.
 """
-        inspection_directive = """
-**CRITICAL PRE-IMPLEMENTATION REQUIREMENT:**

-Before writing any code, carefully inspect the existing repository structure. Use the `editor` tool to `view` all relevant stub files, test files, and related classes. Extract and strictly adhere to their exact method signatures, class hierarchies, and constructor requirements. Do not assume or invent interfaces; implement exactly what the existing code expects.
+        safe_log("=" * 50)
+        safe_log("PHASE 1: CONTRACT EXTRACTION")
+        safe_log("=" * 50)
+
+        # Phase 1: Extract contract from test/stub files
+        contract_chat_history, contract_calls = chat_with_agent(
+            contract_plan_phase_instruction,
+            model=self.code_model,
+            msg_history=[],
+            logging=safe_log,
+            timeout=timeout,
+        )
+
+        # Extract the JSON contract plan from the response
+        contract_response = ""
+        if contract_chat_history:
+            # Get the last assistant message
+            for msg in reversed(contract_chat_history):
+                if isinstance(msg, dict):
+                    if msg.get("role") == "assistant":
+                        content = msg.get("content", "")
+                        if content:
+                            contract_response = content
+                            break
+                else:
+                    # Handle non-dict response objects
+                    contract_response = str(msg)
+                    break
+
+        contract_plan = self.extract_json_from_response(contract_response)
+
+        if not isinstance(contract_plan, dict):
+            safe_log(f"Warning: Could not extract valid JSON contract plan. Raw response: {contract_response[:500]}")
+            contract_plan = {
+                "classes": [],
+                "functions": [],
+                "error_messages": [],
+                "files_to_modify": [],
+                "test_files": [],
+                "key_observations": [
+                    "Could not parse structured contract. Agent response should be reviewed."
+                ],
+            }
+
+        safe_log(f"Extracted contract plan with {len(contract_plan.get('classes', []))} classes and {len(contract_plan.get('functions', []))} functions")
+        safe_log(f"Files to modify: {contract_plan.get('files_to_modify', [])}")
+        safe_log(f"Key observations: {contract_plan.get('key_observations', [])}")
+
+        # Phase 2: Implementation & Verification
+        phase2_instruction = f"""I have uploaded a code repository in the directory {self.git_dir}. Help solve the following problem.
+
```

```
+<problem_description>
+{self.problem_statement}
+</problem_description>
+
+# PHASE 2: IMPLEMENTATION & VERIFICATION

-**Specific Instructions:**
-1. First, explore the repository structure using `view` commands to understand the file layout
-2. Read all stub files, interface definitions, and test files to understand the expected API
-3. Summarize the required method signatures, constructor parameters, and class hierarchies
-4. Only after thoroughly understanding the existing code structure, proceed with your implementation
-5. Implement exactly what the stubs and tests expect - never invent new interfaces
+## CONTRACT PLAN (from Phase 1)

-**Reminder:** The coding agent needs to deal with different languages including C++, Go, Java, JavaScript,
      Python, and Rust. Adhere to the exact method signatures, class hierarchies, and constructor
      requirements defined in the existing code, regardless of programming language.
+Based on analysis of test files and stub files, here is the exact API contract you MUST adhere to:
+
+```json
+{json.dumps(contract_plan, indent=2)}
+```
+
+## YOUR TASK
+
+Now you must implement the solution. You are STRICTLY BOUND by the contract above.
+
+### Step 1: Review the Contract
+- Carefully read through the contract_plan
+- Identify all classes, methods, constructors, and error messages that need to be implemented
+- Note the exact file paths where modifications are needed
+
+### Step 2: Implement the Solution
+- Use the `editor` tool to make changes to the files listed in `files_to_modify`
+- Follow the exact signatures from the contract - do NOT invent new methods or change existing signatures
+- Implement error handling that raises the EXACT error messages specified in `error_messages`
+- Ensure all constructors match the parameters exactly as specified
+
+### Step 3: Self-Audit Checklist
+Before outputting your patch, you MUST perform a self-audit. Review your implementation against this
      checklist:
+
+1. [ ] **Class Names**: Do all class names match exactly (including inheritance)?
+2. [ ] **Constructor Signatures**: Do all constructor signatures match the contract parameters exactly?
+3. [ ] **Method Signatures**: Do all method signatures (name, parameters, return types) match?
+4. [ ] **Error Messages**: Do all error raises use the EXACT error strings specified?
+5. [ ] **File Paths**: Are all modifications made to the correct files?
+6. [ ] **No Extra Code**: Have you avoided adding methods or features not in the contract?
+
+For each item, explicitly state PASS or FAIL and provide evidence from your implementation.
+
+### Step 4: Output the Final Patch
+After the self-audit (all items must PASS), output your final implementation. Make sure:
+- You use the `editor` tool to edit the files
+- The implementation strictly follows the contract
+- All methods are properly implemented to pass the tests
+
+**IMPORTANT**: Do NOT write any code that deviates from the contract. Every method signature and error
      message must match exactly.
  """
-        instruction = f"{task}\n\n{inspection_directive}\nPlease analyze the problem description carefully.
      Then make edits to the code files to complete the instruction."
-        chat_history, n_llm_calls_used = chat_with_agent(
-            instruction,
+
+        safe_log("=" * 50)
+        safe_log("PHASE 2: IMPLEMENTATION & VERIFICATION")
+        safe_log("=" * 50)
+
+        # Phase 2: Implement and verify
+        implementation_chat_history, impl_calls = chat_with_agent(
+            phase2_instruction,
             model=self.code_model,
-            msg_history=[],
+            msg_history=contract_chat_history,  # Continue from phase 1
             logging=safe_log,
             timeout=timeout,
         )
-        chat_history_str = str(chat_history)
+
+        # Return the full chat history for logging
```

```
+        return implementation_chat_history


 def main():

diff --git a/coding_agent.py b/coding_agent.py
index 25f5411..e98de16 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -193,9 +193,10 @@ class AgenticSystem:
     def forward(self, timeout):
         """
         The forward function for the AgenticSystem.
-        Implements a two-phase workflow:
+        Implements a multi-phase workflow:
         1. Contract Extraction Phase: Agent reads test/stub files and outputs a structured contract_plan
     JSON
-        2. Implementation & Verification Phase: Agent implements solution adhering to the plan with self-
     audit
+        2. Implementation Phase: Agent implements solution adhering to the plan
+        3. Verification Loop: Agent runs tests, analyzes results, and iterates until passing or max attempts
         """
         # Phase 1: Contract Extraction
         contract_plan_phase_instruction = f"""I have uploaded a code repository in the directory {self.
    git_dir}. Help solve the following problem.
@@ -329,14 +330,14 @@ Output only the JSON object and a brief summary of your findings. No implementat
         safe_log(f"Files to modify: {contract_plan.get('files_to_modify', [])}")
         safe_log(f"Key observations: {contract_plan.get('key_observations', [])}")

-        # Phase 2: Implementation & Verification
+        # Phase 2: Implementation
         phase2_instruction = f"""I have uploaded a code repository in the directory {self.git_dir}. Help
    solve the following problem.

 <problem_description>
 {self.problem_statement}
 </problem_description>

-# PHASE 2: IMPLEMENTATION & VERIFICATION
+# PHASE 2: IMPLEMENTATION

 ## CONTRACT PLAN (from Phase 1)

@@ -373,20 +374,17 @@ Before outputting your patch, you MUST perform a self-audit. Review your impleme

 For each item, explicitly state PASS or FAIL and provide evidence from your implementation.

-### Step 4: Output the Final Patch
-After the self-audit (all items must PASS), output your final implementation. Make sure:
-- You use the `editor` tool to edit the files
-- The implementation strictly follows the contract
-- All methods are properly implemented to pass the tests
+### Step 4: Output the Final Implementation
+After the self-audit (all items must PASS), make your final implementation using the `editor` tool.

 **IMPORTANT**: Do NOT write any code that deviates from the contract. Every method signature and error
     message must match exactly.
 """

         safe_log("=" * 50)
-        safe_log("PHASE 2: IMPLEMENTATION & VERIFICATION")
+        safe_log("PHASE 2: IMPLEMENTATION")
         safe_log("=" * 50)

-        # Phase 2: Implement and verify
+        # Phase 2: Implement
         implementation_chat_history, impl_calls = chat_with_agent(
             phase2_instruction,
             model=self.code_model,
@@ -395,8 +393,129 @@ After the self-audit (all items must PASS), output your final implementation. Ma
             timeout=timeout,
         )

+        # Phase 3: Verification Loop
+        current_chat_history = implementation_chat_history
+        max_iterations = 5
+
+        # Determine if we should run tests based on language
+        should_run_tests = True
+
+        if should_run_tests:
```

```
+            # Prepare the current edits for the agent to see
+            current_edits_msg = self.get_current_edits()
+
+            for iteration in range(1, max_iterations + 1):
+                safe_log("=" * 50)
+                safe_log(f"PHASE 3: VERIFICATION - Iteration {iteration}/{max_iterations}")
+                safe_log("=" * 50)
+
+                # Run tests
+                test_result = self.run_tests()
+                test_output = test_result.get("output", "")
+                test_exit_code = test_result.get("exit_code", -1)
+
+                safe_log(f"Test exit code: {test_exit_code}")
+
+                if test_exit_code == 0:
+                    # Tests pass!
+                    safe_log("All tests passed!")
+                    break
+
+                # Tests failed - provide raw output to agent for analysis
+                verification_instruction = f"""## TEST RESULTS (Iteration {iteration})
+
+Tests failed with exit code {test_exit_code}. Here is the raw test output - analyze it to understand what
      went wrong:
+
+{test_output}
+
+## REMAINING CONTRACT
+
+```json
+{json.dumps(contract_plan, indent=2)}
+```
+
+## YOUR TASK
+
+1. **Analyze the test output above** to understand what's failing
+2. **Identify which specific tests or functionality are failing**
+3. **Use the `editor` tool to fix the implementation**
+4. **Focus on fixing the issues revealed by the test output**
+
+Make targeted fixes to pass the failing tests. Then we will run tests again.
+"""
+
+                # Add current edits context
+                if current_edits_msg and current_edits_msg != []:
+                    verification_instruction += "\n\n## CURRENT CHANGES\n"
+                    if isinstance(current_edits_msg, list) and len(current_edits_msg) > 0:
+                        edit_msg = current_edits_msg[-1] if isinstance(current_edits_msg[-1], dict) else
      current_edits_msg
+                        if isinstance(edit_msg, dict) and edit_msg.get("content"):
+                            content = edit_msg["content"]
+                            if isinstance(content, list):
+                                for item in content:
+                                    if isinstance(item, dict) and item.get("type") == "input_text":
+                                        verification_instruction += item.get("text", "")
+                                    elif isinstance(item, str):
+                                        verification_instruction += item
+                            else:
+                                verification_instruction += str(content)
+
+                # Continue with agent to fix issues
+                current_chat_history, _ = chat_with_agent(
+                    verification_instruction,
+                    model=self.code_model,
+                    msg_history=current_chat_history,
+                    logging=safe_log,
+                    timeout=timeout,
+                )
+
+                safe_log(f"Agent made changes in iteration {iteration}. Testing again...")
+
         # Return the full chat history for logging
-        return implementation_chat_history
+        return current_chat_history
+
+    def run_tests(self):
+        """
+        Run tests for the current repository.
+        Returns a dict with 'output' (str) and 'exit_code' (int).
+        """
```

```
+        try:
+            # Import the test utility
+            from utils.test_utils import run_test_command
+
+            # Get the test command for the language
+            test_cmd = TEST_COMMANDS.get(self.language, [["pytest", "-rA", "--tb=short"]])
+
+            # Build the command as a string for bash
+            if isinstance(test_cmd, list):
+                if len(test_cmd) == 1 and isinstance(test_cmd[0], list):
+                    cmd_parts = test_cmd[0]
+                else:
+                    cmd_parts = test_cmd
+            else:
+                cmd_parts = test_cmd
+
+            # Convert to bash command string
+            if isinstance(cmd_parts[0], list):
+                bash_cmd = " && ".join([' '.join(cmd) for cmd in cmd_parts])
+            else:
+                bash_cmd = " && ".join(cmd_parts)
+
+            # Run the test command using the utility
+            result = run_test_command(bash_cmd, cwd=self.git_dir)
+
+            return {
+                "output": result["output"],
+                "exit_code": result["exit_code"]
+            }
+        except Exception as e:
+            safe_log(f"Error running tests: {str(e)}")
+            return {
+                "output": f"Error running tests: {str(e)}",
+                "exit_code": -1
+            }


 def main():
diff --git a/tests/test_test_utils.py b/tests/test_test_utils.py
new file mode 100644
index 0000000..8ac038f
--- /dev/null
+++ b/tests/test_test_utils.py
@@ -0,0 +1,119 @@
+# Tests for test_utils.py
+
+import pytest
+from utils.test_utils import parse_test_output
+
+
+class TestParseTestOutput:
+    """Tests for parse_test_output function."""
+
+    def test_all_tests_passed(self):
+        """Test parsing output when all tests pass."""
+        output = """
+============================= test session starts ==============================
+platform linux -- Python 3.10.20, pytest-9.0.3, pluggy-1.6.0
+rootdir: /test
+collected 5 items
+
+tests/test_a.py ..                                                       [ 40%]
+tests/test_b.py ...                                                      [100%]
+
+============================== 5 passed in 0.1s ===============================
+"""
+        result = parse_test_output(output, 0)
+        assert result["passed"] == 5
+        assert result["failed"] == 0
+        assert result["error"] == 0
+        assert result["success"] == True
+        assert result["exit_code"] == 0
+
+    def test_some_tests_failed(self):
+        """Test parsing output when some tests fail."""
+        output = """
+============================= test session starts ==============================
+platform linux -- Python 3.10.20, pytest-9.0.3, pluggy-1.6.0
+rootdir: /test
+collected 10 items
+
```

```
+tests/test_a.py ..F...                                                  [ 80%]
+tests/test_b.py ...                                                     [100%]
+
+================================== FAILURES ===================================
+_____________________________ TestClass.test_failed ______________________________
+
+    def test_failed(self):
+>       assert False
+E       assert False
+
+tests/test_a.py:6: AssertionError
+========================== 1 failed, 9 passed in 0.2s ==========================
+"""
+        result = parse_test_output(output, 1)
+        assert result["passed"] == 9
+        assert result["failed"] == 1
+        assert result["success"] == False
+        assert result["exit_code"] == 1
+
+    def test_tests_with_errors(self):
+        """Test parsing output when there are errors."""
+        output = """
+============================= test session starts ==============================
+platform linux -- Python 3.10.20, pytest-9.0.3, pluggy-1.6.0
+rootdir: /test
+collected 5 items
+
+tests/test_a.py E...                                                     [ 80%]
+tests/test_b.py ..                                                       [100%]
+
+==================================== ERRORS ====================================
+_________________________ ERROR at setup of test_error __________________________
+FileNotFoundError: [Errno 2] No such file or directory
+========================== 1 error, 4 passed in 0.1s ===========================
+"""
+        result = parse_test_output(output, 1)
+        assert result["passed"] == 4
+        assert result["error"] == 1
+        assert result["success"] == False
+
+    def test_tests_with_skipped(self):
+        """Test parsing output when there are skipped tests."""
+        output = """
+============================= test session starts ==============================
+platform linux -- Python 3.10.20, pytest-9.0.3, pluggy-1.6.0
+rootdir: /test
+collected 5 items
+
+tests/test_a.py ..s..                                                    [100%]
+
+========================== 4 passed, 1 skipped in 0.1s ===========================
+"""
+        result = parse_test_output(output, 0)
+        assert result["passed"] == 4
+        assert result["skipped"] == 1
+        assert result["success"] == True
+
+    def test_empty_output(self):
+        """Test parsing empty output."""
+        result = parse_test_output("", 0)
+        assert result["passed"] == 0
+        assert result["failed"] == 0
+        assert result["success"] == True
+
+    def test_exit_code_parameter(self):
+        """Test exit code parameter is used."""
+        output = "test completed"
+        result = parse_test_output(output, 5)
+        assert result["exit_code"] == 5
+        assert result["success"] == False
+
+    def test_success_summary(self):
+        """Test summary for passing tests."""
+        output = "5 passed in 0.1s"
+        result = parse_test_output(output, 0)
+        assert result["summary"] == "Tests passed"
+
+    def test_failure_summary(self):
+        """Test summary for failing tests."""
+        output = "2 failed, 8 passed in 0.2s"
+        result = parse_test_output(output, 1)
```

```
+        assert "2 test(s) failed" in result["summary"]
diff --git a/utils/test_utils.py b/utils/test_utils.py
new file mode 100644
index 0000000..3284dda
--- /dev/null
+++ b/utils/test_utils.py
@@ -0,0 +1,112 @@
+# Utility functions for running tests and analyzing results.
+
+import re
+from typing import Dict, Any
+
+
+def parse_test_output(output: str, exit_code: int = 0) -> Dict[str, Any]:
+    """
+    Parse test output to extract relevant information.
+    Returns a dict with 'passed', 'failed', 'error_count', 'summary'.
+
+    Args:
+        output: Raw test output string
+        exit_code: Exit code from test execution
+
+    Returns:
+        Dict with parsed test results
+    """
+    result = {
+        "passed": 0,
+        "failed": 0,
+        "error": 0,
+        "skipped": 0,
+        "exit_code": exit_code,
+        "success": exit_code == 0,
+        "summary": "",
+        "raw_output": output
+    }
+
+    # Count patterns
+    passed_match = re.search(r"(\d+)\s+passed", output)
+    if passed_match:
+        result["passed"] = int(passed_match.group(1))
+
+    failed_match = re.search(r"(\d+)\s+failed", output)
+    if failed_match:
+        result["failed"] = int(failed_match.group(1))
+
+    error_match = re.search(r"(\d+)\s+error", output, re.IGNORECASE)
+    if error_match:
+        result["error"] = int(error_match.group(1))
+
+    skipped_match = re.search(r"(\d+)\s+skipped", output)
+    if skipped_match:
+        result["skipped"] = int(skipped_match.group(1))
+
+    # Determine success from exit code and content
+    if exit_code == 0:
+        result["success"] = True
+        result["summary"] = "Tests passed"
+    else:
+        result["success"] = False
+        if result["failed"] > 0:
+            result["summary"] = f"{result['failed']} test(s) failed"
+        elif result["error"] > 0:
+            result["summary"] = f"{result['error']} error(s) occurred"
+        else:
+            result["summary"] = f"Tests failed with exit code {exit_code}"
+
+    return result
+
+
+def run_test_command(command: str, cwd: str = None) -> Dict[str, Any]:
+    """
+    Run a test command and return the results.
+
+    Args:
+        command: Test command to run
+        cwd: Working directory
+
+    Returns:
+        Dict with 'output', 'exit_code', and parsed results
+    """
+    try:
```

```
+        from tools.bash import tool_function
+
+        # Execute the command
+        if cwd:
+            full_command = f"cd {cwd} && {command}"
+        else:
+            full_command = command
+
+        output = tool_function(full_command)
+
+        # Try to extract exit code from output
+        exit_code = 0
+        exit_match = re.search(r"exit code[:\s]+(\d+)", output, re.IGNORECASE)
+        if exit_match:
+            exit_code = int(exit_match.group(1))
+
+        # Parse the output
+        test_result = parse_test_output(output, exit_code)
+
+        return {
+            "output": output,
+            "exit_code": test_result["exit_code"],
+            "success": test_result["success"],
+            "summary": test_result["summary"],
+            "passed": test_result["passed"],
+            "failed": test_result["failed"],
+            "error": test_result["error"],
+        }
+    except Exception as e:
+        return {
+            "output": f"Error running tests: {str(e)}",
+            "exit_code": -1,
+            "success": False,
+            "summary": f"Test execution failed: {str(e)}",
+            "passed": 0,
+            "failed": 0,
+            "error": 0,
+        }
```

Figure 18: Code modification on SWE-bench

```
diff --git a/coding_agent.py b/coding_agent.py
index d19efd6..98f364a 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -3,16 +3,19 @@
 import argparse
 import logging
 import os
+import re
 import subprocess
 import threading
 from logging.handlers import RotatingFileHandler
 from time import time

 from llm_withtools import (CLAUDE_MODEL, OPENAI_MODEL, chat_with_agent,
-                           convert_msg_history)
-from utils.eval_utils import (get_report_score, msg_history_to_report,
-                              score_tie_breaker)
+                           convert_msg_history, set_explored_files_callback,
+                           clear_explored_files_callback)
+from utils.eval_utils import (enforce_diff_minimality, get_report_score,
+                              msg_history_to_report, score_tie_breaker)
 from utils.git_utils import apply_patch, diff_versus_commit, reset_to_commit
+from utils.trace_utils import extract_test_function_names, generate_trace_instruction

 # Thread-local storage for logger instances
 thread_local = threading.local()
@@ -96,6 +99,9 @@ class AgenticSystem:
         self.instance_id = instance_id if not self_improve else "hgm"
         self.code_model = model

+        # Track all files explored by the agent via the editor tool's view command
+        self.explored_files = []
+
         # Initialize logger and store it in thread-local storage
         self.logger = setup_logger(chat_history_file)

@@ -172,12 +178,88 @@ Your task is to run the regression tests in the {self.git_tempdir} directory to
         )
         return test_report

+    def _extract_test_failures_from_history(self, msg_history):
+        failed_tests = []
+        for msg in msg_history:
+            if not isinstance(msg, dict):
+                continue
+            content = msg.get("content") or msg.get("output") or ""
+            if not isinstance(content, str):
+                content = str(content)
+            for pattern in (
+                r"FAILED\s+\S+?\.py::(test_\w+)",
+                r"\b(test_\w+)\b.*FAILED",
+            ):
+                for match in re.findall(pattern, content, re.IGNORECASE):
+                    name = match[0] if isinstance(match, tuple) else match
+                    if name.startswith("test_") and name not in failed_tests:
+                        failed_tests.append(name)
+        return failed_tests
+
+    def _build_trace_context(self, test_names):
+        if not test_names:
+            return ""
+        blocks = []
+        for test_name in test_names[:2]:
+            blocks.append(
+                generate_trace_instruction(
+                    instance_id=self.instance_id,
+                    test_function_name=test_name,
+                    target_object=test_name.replace("test_", ""),
+                    git_tempdir=self.git_tempdir,
+                )
+            )
+        return "\n".join(blocks)
+
+    def _apply_diff_minimality(self, diff_str):
+        cleaned_diff = enforce_diff_minimality(
+            problem_statement=self.problem_statement,
+            explored_files=self.explored_files,
+            diff_str=diff_str,
```

```
+            git_tempdir=self.git_tempdir,
+        )
+        if cleaned_diff != diff_str:
+            reset_to_commit(self.git_tempdir, self.base_commit)
+            result = subprocess.run(
+                ["git", "-C", self.git_tempdir, "apply", "-"],
+                input=cleaned_diff,
+                text=True,
+                capture_output=True,
+            )
+            if result.returncode != 0:
+                print(f"Warning: Failed to apply minimized diff: {result.stderr}")
+            else:
+                print("Applied minimized diff successfully")
+        safe_log(f"Diff minimality complete. Explored files: {self.explored_files}")
+
     def forward(self, timeout=3600):
         timeout -= 60
         start_time = time()
-        """
-        The forward function for the AgenticSystem.
-        """
+        set_explored_files_callback(self.explored_files.append)
+
+        hinted_tests = extract_test_function_names(self.test_description or "")
+        hinted_tests += extract_test_function_names(self.problem_statement or "")
+        deduped = []
+        seen = set()
+        for test_name in hinted_tests:
+            if test_name not in seen:
+                seen.add(test_name)
+                deduped.append(test_name)
+        hinted_tests = deduped
+
+        pre_trace_context = self._build_trace_context(hinted_tests)
+        trace_preamble = ""
+        if pre_trace_context:
+            trace_preamble = (
+                f"\n<code_path_trace>\n{pre_trace_context}\n</code_path_trace>\n"
+                "Use this trace to locate the correct fix path before editing.\n"
+            )
+        else:
+            trace_preamble = (
+                "\nBefore editing, run the most relevant failing test(s) for this issue "
+                f"in {self.git_tempdir} so you can observe actual vs expected output.\n"
+            )
+
         instruction = f"""I have uploaded a Python code repository in the directory {self.git_tempdir}. Help
       solve the following problem.

 <problem_description>
@@ -187,17 +269,47 @@ Your task is to run the regression tests in the {self.git_tempdir} directory to
 <test_description>
 {self.test_description}
 </test_description>
-
-Your task is to make changes to the files in the {self.git_tempdir} directory to address the <
     problem_description>. I have already taken care of the required dependencies.
+{trace_preamble}
+Make changes in {self.git_tempdir} to address the problem.
 """
-        chat_history, n_llm_calls_used = chat_with_agent(
+        msg_history, _ = chat_with_agent(
             instruction,
             model=self.code_model,
             msg_history=[],
             logging=safe_log,
             timeout=timeout - (time() - start_time),
         )
-        chat_history_str = str(chat_history)
+
+        generic_history = convert_msg_history(msg_history, self.code_model)
+        failed_tests = self._extract_test_failures_from_history(generic_history)
+        if not failed_tests and hinted_tests:
+            failed_tests = hinted_tests[:1]
+
+        for attempt in range(2):
+            remaining = timeout - (time() - start_time)
+            if not failed_tests or remaining <= 120:
+                break
+            trace_instruction = self._build_trace_context(failed_tests[:1])
```

```
+            follow_up = f"""The following test(s) failed: {', '.join(failed_tests)}.
+
+{trace_instruction}
+
+Revise your fix in {self.git_tempdir} so the patch addresses the traced code path and makes the failing test
      pass.
+"""
+            msg_history, _ = chat_with_agent(
+                follow_up,
+                model=self.code_model,
+                msg_history=msg_history,
+                logging=safe_log,
+                timeout=remaining,
+            )
+            generic_history = convert_msg_history(msg_history, self.code_model)
+            failed_tests = self._extract_test_failures_from_history(generic_history)
+            if not failed_tests:
+                break
+
+        clear_explored_files_callback()
+        self._apply_diff_minimality(self.get_current_edits())


 def main():
diff --git a/llm_withtools.py b/llm_withtools.py
index ba7ea87..b35bae3 100644
--- a/llm_withtools.py
+++ b/llm_withtools.py
@@ -13,6 +13,28 @@ import openai
 from llm import create_client
 from tools import load_all_tools

+# Module-level callback for tracking explored files
+_EXPLORED_FILES_CALLBACK = None
+
+
+def set_explored_files_callback(callback):
+    global _EXPLORED_FILES_CALLBACK
+    _EXPLORED_FILES_CALLBACK = callback
+
+
+def clear_explored_files_callback():
+    global _EXPLORED_FILES_CALLBACK
+    _EXPLORED_FILES_CALLBACK = None
+
+
+def _track_editor_view(tool_name, tool_input):
+    """Helper to track when editor view is called."""
+    if _EXPLORED_FILES_CALLBACK and tool_name == "editor":
+        command = tool_input.get("command", "")
+        path = tool_input.get("path", "")
+        if command == "view" and path:
+            _EXPLORED_FILES_CALLBACK(path)
+
 CLAUDE_MODEL = "anthropic/claude-sonnet-4"
 OPENAI_MODEL = "gpt-5"
 MAX_XML_TOOL_FORMAT_RETRIES = 2
@@ -80,6 +102,9 @@ def _assistant_message_with_tool_call(message, tool_use):


 def process_tool_call(tools_dict, tool_name, tool_input):
+    # Track editor tool usage
+    _track_editor_view(tool_name, tool_input)
+
     try:
         if tool_name in tools_dict:
             return tools_dict[tool_name]["function"](**tool_input)
diff --git a/utils/eval_utils.py b/utils/eval_utils.py
index 1c6e117..15a9d83 100644
--- a/utils/eval_utils.py
+++ b/utils/eval_utils.py
@@ -125,3 +125,167 @@ Your response will be automatically parsed, so ensure that the string response i
     except Exception as e:
         logging(f"Error in score_tie_breaker: {e}")
     return best_score_index
+
+
+def _extract_file_from_diff_header(line: str) -> str:
+    """
+    Extract the file path from a git diff header line like:
+    'diff --git a/path/to/file.py b/path/to/file.py'
```

```
+    Returns the path after 'a/' (which should match the path after 'b/').
+    """
+    # Match the pattern: diff --git a/... b/...
+    import re
+    match = re.search(r'diff --git a/(.+?) b/(.+)$', line)
+    if match:
+        # Return the path from either side (they should be the same)
+        return match.group(1).split('/')[-1] or match.group(2).split('/')[-1]
+    return None
+
+
+def _get_relevant_test_files(explored_files: list, all_files: set) -> set:
+    """
+    Identify test files that are likely related to the explored files.
+    A test file is relevant if:
+    - It's in a 'test_' prefix or '_test' suffix pattern
+    - It's in a 'tests/' or 'test/' directory
+    - The base module name matches (e.g., inspect.py -> test_inspect.py)
+    """
+    relevant = set()
+    for explored in explored_files:
+        # Get the filename without extension
+        import os
+        base = os.path.splitext(os.path.basename(explored))[0]
+        # Check if this is a test-related file (e.g., test_inspect.py for inspect.py)
+        # Look for files that have test_ prefix or _test suffix with matching base name
+        for f in all_files:
+            f_base = os.path.splitext(os.path.basename(f))[0]
+            f_dir = os.path.dirname(f)
+            # Check test directory pattern
+            if os.path.basename(f_dir).lower() in ('test', 'tests'):
+                relevant.add(f)
+                continue
+            # Check naming pattern: test_<module>.py or <module>_test.py
+            if (f_base == f'test_{base}' or
+                f_base == f'{base}_test' or
+                f == f'test_{base}.py' or
+                f == f'{base}_test.py'):
+                relevant.add(f)
+    return relevant
+
+
+def enforce_diff_minimality(
+    problem_statement: str,
+    explored_files: list,
+    diff_str: str,
+    git_tempdir: str = None,
+) -> str:
+    """
+    Enforces diff minimality by filtering out changes to files that are not
+    relevant to the problem statement or the files explored by the agent.
+
+    This function:
+    1. Parses the diff to extract modified file paths
+    2. Computes a relevance score for each file based on:
+       - Whether the file appears in the problem statement (exact match or keyword match)
+       - Whether the file was explored by the agent
+       - Whether it's a test file related to an explored file
+    3. Returns a filtered diff containing only high-relevance files
+
+    Args:
+        problem_statement: The original problem statement/issue description
+        explored_files: List of file paths that the agent viewed during execution
+        diff_str: The complete diff string to filter
+        git_tempdir: Optional path to the git repository (used to find all files)
+
+    Returns:
+        A filtered diff string containing only changes to relevant files
+    """
+    import os
+    if not diff_str:
+        return diff_str
+
+    # Normalize problem statement for matching
+    problem_lower = problem_statement.lower()
+
+    # Extract file paths from the diff
+    import re
+    diff_lines = diff_str.split('\n')
+    diff_blocks = []
+    current_block = []
```

```
+
+    for line in diff_lines:
+        if line.startswith('diff --git'):
+            if current_block:
+                diff_blocks.append('\n'.join(current_block))
+            current_block = [line]
+        else:
+            current_block.append(line)
+    if current_block:
+        diff_blocks.append('\n'.join(current_block))
+
+    # Analyze each diff block
+    relevant_blocks = []
+    for block in diff_blocks:
+        if not block.strip():
+            continue
+
+        # Get the file path from the diff header
+        header_line = block.split('\n')[0]
+        filename = _extract_file_from_diff_header(header_line)
+        if filename is None:
+            # If we can't parse the header, keep the block to be safe
+            relevant_blocks.append(block)
+            continue
+
+        # Compute relevance score
+        score = 0
+
+        # Check if the file was explored by the agent
+        for explored in explored_files:
+            explored_name = os.path.basename(explored) if explored else None
+            if explored_name == filename:
+                score += 10  # High relevance - was explored
+                break
+            # Also check if the path contains the filename
+            if filename in explored:
+                score += 5
+
+        # Check if filename appears in problem statement
+        if filename.lower() in problem_lower:
+            score += 10
+
+        # Check if any parent directories appear in problem statement
+        for part in filename.split('/'):
+            if part.lower() in problem_lower:
+                score += 5
+                break
+
+        # Check if it's a test file related to explored modules
+        if git_tempdir:
+            test_files = _get_relevant_test_files(explored_files, set())
+            if any(filename in f for f in test_files):
+                score += 8
+
+        # Keep blocks with sufficient relevance score
+        # Threshold of 5 ensures we only keep clearly relevant changes
+        if score >= 5:
+            relevant_blocks.append(block)
+        else:
+            # Log for debugging (optional)
+            print(f"[diff_minimality] Filtering out {filename} (score: {score})")
+
+    # Reconstruct the diff with only relevant blocks
+    result = '\n\n'.join(relevant_blocks)
+
+    # Ensure trailing newline
+    if result and not result.endswith('\n'):
+        result += '\n'
+
+    return result
+
+
+def _ensure_patch_trailing_newline(patch_str):
+    """Ensure patch string ends with newline."""
+    if patch_str and not patch_str.endswith('\n'):
+        return patch_str + '\n'
+    return patch_str
diff --git a/utils/trace_utils.py b/utils/trace_utils.py
new file mode 100644
index 0000000..b030135
--- /dev/null
```

```
+++ b/utils/trace_utils.py
@@ -0,0 +1,475 @@
+# Utility functions for tracing docstring code paths in autodoc-related issues.
+
+import os
+import re
+import subprocess
+from pathlib import Path
+from typing import Dict, List, Optional, Tuple
+
+
+def run_grep(pattern: str, directory: str, file_pattern: str = "*.py", ignore_case: bool = True, extended:
     bool = False) -> List[str]:
+    """
+    Run grep in a directory for a pattern in files matching file_pattern.
+    Returns a list of matching lines.
+
+    Args:
+        pattern: The regex pattern to search for
+        directory: The directory to search in
+        file_pattern: The file pattern to match (e.g., "*.py")
+        ignore_case: Whether to use case-insensitive matching
+        extended: Whether to use extended regex mode (-E flag)
+    """
+    cmd = ["grep", "-r", "--include=" + file_pattern]
+    if ignore_case:
+        cmd.append("-i")
+    if extended:
+        cmd.append("-E")
+    cmd.extend(["-n", "--color=never", pattern, directory])
+    try:
+        result = subprocess.run(
+            cmd,
+            capture_output=True,
+            text=True,
+            timeout=30,
+        )
+        if result.returncode in (0, 1):  # 0 = match found, 1 = no match
+            return result.stdout.strip().split("\n") if result.stdout.strip() else []
+        return []
+    except subprocess.TimeoutExpired:
+        return []
+    except Exception:
+        return []
+
+
+def run_agrep(pattern: str, directory: str, file_pattern: str = "*.py") -> List[str]:
+    """
+    Run ag (the silver searcher) for faster grep-like searching.
+    Falls back to grep if ag is not available.
+    """
+    cmd = ["ag", "--python", "--literal", "--color=never", pattern, directory]
+    try:
+        result = subprocess.run(
+            cmd,
+            capture_output=True,
+            text=True,
+            timeout=30,
+        )
+        if result.returncode in (0, 1):
+            return result.stdout.strip().split("\n") if result.stdout.strip() else []
+        return []
+    except (subprocess.TimeoutExpired, FileNotFoundError):
+        # Fall back to grep
+        return run_grep(pattern, directory, file_pattern)
+
+
+def find_test_file(git_tempdir: str, test_function_name: str) -> Optional[str]:
+    """
+    Find the test file containing the given test function name.
+    Returns the file path relative to git_tempdir, or None if not found.
+    """
+    # Use grep with extended regex to find the function definition
+    # Pattern: def test_function_name(
+    results = run_grep(
+        rf'def\s+{re.escape(test_function_name)}\s*\(',
+        git_tempdir,
+        file_pattern="*.py",
+        extended=True
+    )
+    if results:
```

```
+        for line in results:
+            if ":" in line and line.strip():
+                # Extract file path (format is usually "filename:line_number:content")
+                file_path = line.split(":")[0]
+                return file_path
+    return None
+
+
+def extract_do_autodoc_call(test_content: str) -> Optional[Dict[str, str]]:
+    """
+    Extract the do_autodoc(app, '<type>', '<object>') call from test content.
+    Returns a dict with 'type' and 'object' keys, or None if not found.
+    """
+    # Match patterns like: do_autodoc(app, 'class', 'SomeClass')
+    # or: do_autodoc(app, "class", "SomeClass")
+    pattern = r"do_autodoc\s*\(\s*app\s*,\s*['\"]([^'\"]+)['\"]\s*,\s*['\"]([^'\"]+)['\"]\s*\)"
+    match = re.search(pattern, test_content)
+    if match:
+        return {
+            "type": match.group(1),
+            "object": match.group(2),
+        }
+    return None
+
+
+def extract_expected_output(test_content: str) -> Optional[str]:
+    """
+    Extract the expected output from an assertion in the test.
+    Looks for patterns like:
+    - assert 'expected text' in result
+    - assert result == 'expected text'
+    - assert "expected text" in result
+    """
+    # Pattern for assert result == 'expected' or assert result == "expected"
+    pattern = r"assert\s+.*?==\s*['\"](.+?)['\"]"
+    matches = re.findall(pattern, test_content, re.DOTALL)
+    if matches:
+        return matches[-1]  # Return the last match (usually the most relevant)
+
+    # Pattern for assert 'expected' in result or assert "expected" in result
+    pattern = r"assert\s+['\"](.+?)['\"]\s+in\s+.*?"
+    matches = re.findall(pattern, test_content, re.DOTALL)
+    if matches:
+        return matches[-1]
+
+    return None
+
+
+def find_documenter_class(git_tempdir: str, obj_type: str) -> Optional[str]:
+    """
+    Find the documenter class that handles the given object type.
+    For example, 'class' -> 'DataDocumenter', 'module' -> 'ModuleDocumenter', etc.
+    """
+    # Map common autodoc types to documenter naming patterns
+    type_to_pattern = {
+        "class": r"(?:^|\s)(DataDocumenter|ClassDocumenter)\b",
+        "function": r"(?:^|\s)(FunctionDocumenter|MethodDocumenter)\b",
+        "method": r"(?:^|\s)(MethodDocumenter)\b",
+        "attribute": r"(?:^|\s)(AttributeDocumenter|DataDocumenter)\b",
+        "module": r"(?:^|\s)(ModuleDocumenter)\b",
+        "exception": r"(?:^|\s)(ExceptionDocumenter|DataDocumenter)\b",
+        "data": r"(?:^|\s)(DataDocumenter)\b",
+    }
+
+    pattern = type_to_pattern.get(obj_type.lower(), r"(?:^|\s)(\w+Documenter)\b")
+
+    # Search for documenter classes with extended regex
+    results = run_grep(pattern, git_tempdir, file_pattern="*.py", extended=True)
+    for line in results:
+        match = re.search(pattern, line)
+        if match:
+            return match.group(1)
+
+    # If no specific match, look for Documenter base class references
+    results = run_grep(r"class\s+\w+Documenter\s*\(", git_tempdir, file_pattern="*.py", extended=True)
+    for line in results:
+        if "Documenter" in line:
+            match = re.search(r"class\s+(\w+Documenter)", line)
+            if match:
+                return match.group(1)
+
```

```
+    return None
+
+
+def trace_get_object_doc(git_tempdir: str, documenter_class: str) -> Optional[Dict]:
+    """
+    Trace the get_object_doc method in the given documenter class.
+    Returns information about how docstrings are read.
+    """
+    # Use extended regex for better pattern matching
+    results = run_grep(
+        r'def\s+get_object_doc\s*\(',
+        git_tempdir,
+        file_pattern="*.py",
+        extended=True
+    )
+
+    for line in results:
+        if documenter_class.lower() in line.lower() or "Documenter" in line:
+            # Extract file path
+            parts = line.split(":")
+            if len(parts) >= 2:
+                file_path = parts[0]
+                # Read the function to check for __doc__ access
+                try:
+                    content = Path(file_path).read_text()
+                    # Look for self.object.__doc__ or self.object.__doc__
+                    if "self.object.__doc__" in content or "self.object.__doc__" in content:
+                        return {
+                            "file": file_path,
+                            "method": "get_object_doc",
+                            "doc_source": "__doc__",
+                            "uses_self_object_doc": True,
+                        }
+                    # Check for .__doc__ attribute access more generally
+                    doc_access_pattern = r"self\.object\s*\.\s*__doc__"
+                    if re.search(doc_access_pattern, content):
+                        return {
+                            "file": file_path,
+                            "method": "get_object_doc",
+                            "doc_source": "__doc__",
+                            "uses_self_object_doc": True,
+                        }
+                except Exception:
+                    pass
+    return None
+
+
+def check_module_analyzer(git_tempdir: str) -> dict:
+    """
+    Check if ModuleAnalyzer is imported and used in the codebase.
+    Returns a dict with import status and usage information.
+    """
+    # Check for ModuleAnalyzer imports using extended regex
+    # Use \S+ instead of \w+ to handle dotted module paths
+    import_patterns = [
+        r'from\s+\S+\s+import\s+.*ModuleAnalyzer',
+        r'import\s+\S+\.ModuleAnalyzer',
+    ]
+
+    imported = False
+    import_lines = []
+
+    for pattern in import_patterns:
+        results = run_grep(pattern, git_tempdir, file_pattern="*.py", extended=True)
+        import_lines.extend(results)
+        if results:
+            imported = True
+
+    # Check for get_comments usage (ModuleAnalyzer's method for getting #: comments)
+    get_comments_results = run_grep(
+        r'get_comments',
+        git_tempdir,
+        file_pattern="*.py",
+        extended=True
+    )
+    uses_get_comments = len(get_comments_results) > 0
+
+    # Check for comment-based docstring handling
+    comment_patterns = [
+        r'#:\s*\w',
+        r'comment.*docstring',
```

```
+        r'docstring.*comment',
+    ]
+
+    has_comment_handling = False
+    for pattern in comment_patterns:
+        if run_grep(pattern, git_tempdir, file_pattern="*.py", extended=True):
+            has_comment_handling = True
+            break
+
+    return {
+        "is_imported": imported,
+        "import_lines": import_lines[:5],  # Limit to first 5 lines
+        "uses_get_comments": uses_get_comments,
+        "has_comment_handling": has_comment_handling,
+    }
+
+
+def find_fix_location(git_tempdir: str, documenter_class: str, doc_info: dict) -> str:
+    """
+    Determine the likely fix location based on the analysis.
+    Returns a human-readable description of where to fix.
+    """
+    if not doc_info:
+        return "Could not determine fix location from code path analysis."
+
+    file_path = doc_info.get("file", "unknown")
+    doc_source = doc_info.get("doc_source", "unknown")
+
+    module_analyzer_info = check_module_analyzer(git_tempdir)
+
+    if doc_source == "__doc__" and not module_analyzer_info.get("is_imported", False):
+        fix_suggestion = (
+            f"Fix: Modify `{documenter_class}.get_object_doc()` to fall back to "
+            f"`ModuleAnalyzer.get_comments()` when `__doc__` is empty. "
+            f"File: `{file_path}`"
+        )
+    elif doc_source == "__doc__" and module_analyzer_info.get("is_imported", False):
+        fix_suggestion = (
+            f"Fix: Modify `{documenter_class}.get_object_doc()` in `{file_path}` to check "
+            f"`ModuleAnalyzer.get_comments()` as a fallback when `self.object.__doc__` is None/empty. "
+            f"The current code only reads from `__doc__` and ignores comment-based docstrings."
+        )
+    else:
+        fix_suggestion = (
+            f"Examine `{documenter_class}.get_object_doc()` in `{file_path}` to understand "
+            f"how docstrings are currently retrieved. Consider adding fallback logic."
+        )
+
+    return fix_suggestion
+
+
+def trace_docstring_path(
+    instance_id: str,
+    test_function_name: str,
+    target_object: str,
+    git_tempdir: str,
+) -> str:
+    """
+    Trace the docstring code path for autodoc-related issues.
+
+    This function:
+    1. Parses the test file to extract the do_autodoc(app, '<type>', '<object>') call
+    2. Finds the documenter class that handles the given type
+    3. Traces through get_object_doc() to identify where self.object.__doc__ is read
+    4. Checks if ModuleAnalyzer is imported and used
+    5. Outputs a summary of the code path with fix suggestions
+
+    Args:
+        instance_id: The instance ID for the issue
+        test_function_name: The name of the failing test function
+        target_object: The target object being documented
+        git_tempdir: Path to the git repository directory
+
+    Returns:
+        A formatted summary string describing the code path and fix location.
+    """
+    # Step 1: Find and parse the test file
+    test_file = find_test_file(git_tempdir, test_function_name)
+
+    if not test_file:
+        return (
```

```
+            f"[Trace Summary] Could not find test file containing '{test_function_name}' "
+            f"in {git_tempdir}. Manual investigation required."
+        )
+
+    # Read the test file content
+    try:
+        test_path = os.path.join(git_tempdir, test_file)
+        test_content = Path(test_path).read_text()
+    except Exception as e:
+        return f"[Trace Summary] Could not read test file {test_file}: {e}"
+
+    # Step 2: Extract do_autodoc call information
+    autodoc_info = extract_do_autodoc_call(test_content)
+
+    if autodoc_info:
+        obj_type = autodoc_info["type"]
+        obj_name = autodoc_info["object"]
+    else:
+        # Fallback: use target_object if autodoc call not found
+        obj_type = "class"  # Default assumption
+        obj_name = target_object
+
+    expected_output = extract_expected_output(test_content)
+
+    # Step 3: Find the documenter class
+    documenter_class = find_documenter_class(git_tempdir, obj_type)
+
+    if not documenter_class:
+        return (
+            f"[Trace Summary] Could not identify the documenter class handling '{obj_type}' type. "
+            f"Test function: {test_function_name}, Target: {target_object}"
+        )
+
+    # Step 4: Trace get_object_doc
+    doc_info = trace_get_object_doc(git_tempdir, documenter_class)
+
+    # Step 5: Check ModuleAnalyzer
+    module_analyzer_info = check_module_analyzer(git_tempdir)
+
+    # Step 6: Find fix location
+    fix_suggestion = find_fix_location(git_tempdir, documenter_class, doc_info)
+
+    # Build the summary
+    summary_lines = [
+        "=" * 60,
+        "DOCSTRING CODE PATH TRACE SUMMARY",
+        "=" * 60,
+        "",
+        f"Instance: {instance_id}",
+        f"Test Function: {test_function_name}",
+        f"Target Object: {target_object}",
+        f"Test File: {test_file}",
+        "",
+        "--- Analysis Details ---",
+        "",
+        f"1. autodoc() call detected: do_autodoc(app, '{obj_type}', '{obj_name}')",
+    ]
+
+    if expected_output:
+        summary_lines.append(f"   Expected output contains: '{expected_output[:100]}...'")
+
+    summary_lines.extend([
+        "",
+        f"2. Documenter class: {documenter_class} (handles '{obj_type}' objects)",
+    ])
+
+    if doc_info:
+        summary_lines.append(
+            f"3. Docstring source: {documenter_class}.get_object_doc() "
+            f"in {doc_info.get('file', 'unknown')} reads from {doc_info.get('doc_source', 'unknown')}"
+        )
+
+    summary_lines.extend([
+        "",
+        "4. ModuleAnalyzer status:",
+        f"   - Imported: {'Yes' if module_analyzer_info['is_imported'] else 'No'}",
+        f"   - Uses get_comments(): {'Yes' if module_analyzer_info['uses_get_comments'] else 'No'}",
+        f"   - Has comment-based handling: {'Yes' if module_analyzer_info['has_comment_handling'] else 'No
 '}",
+        "",
+        "--- Recommended Fix ---",
```

```
+        "",
+        fix_suggestion,
+        "",
+        "=" * 60,
+    ])
+
+    return "\n".join(summary_lines)
+
+
+def extract_test_function_names(text: str) -> List[str]:
+    """Extract pytest-style test function names mentioned in free text."""
+    if not text:
+        return []
+    names = re.findall(r"\b(test_[A-Za-z0-9_]+)\b", text)
+    seen = set()
+    out = []
+    for name in names:
+        if name not in seen:
+            seen.add(name)
+            out.append(name)
+    return out
+
+
+def generate_trace_instruction(instance_id: str, test_function_name: str, target_object: str, git_tempdir:
      str) -> str:
+    """
+    Generate a formatted instruction for the agent to trace the docstring path.
+    This is a wrapper that formats the output for consumption by the agent.
+    """
+    trace_result = trace_docstring_path(instance_id, test_function_name, target_object, git_tempdir)
+
+    instruction = f"""
+<code_path_trace>
+{trace_result}
+</code_path_trace>
+
+Please examine the code path identified above. The trace shows how docstrings are being read and where the
      fix should be applied. Focus on the recommended fix location and ensure your changes address the root
      cause.
+"""
+    return instruction
+
+
+def test_trace_docstring_path():
+    """
+    Test function to verify trace_docstring_path works correctly.
+    This is a self-test function that doesn't require a real repository.
+    """
+    # This is a basic sanity check - actual testing requires a real repository
+    try:
+        result = trace_docstring_path(
+            instance_id="test_instance",
+            test_function_name="test_placeholder",
+            target_object="TestClass",
+            git_tempdir="/tmp"
+        )
+        assert "Trace Summary" in result or "Could not" in result
+        return True
+    except Exception as e:
+        print(f"Test error: {e}")
+        return False
+
+
+if __name__ == "__main__":
+    import sys
+    if len(sys.argv) >= 4:
+        instance_id = sys.argv[1]
+        test_function_name = sys.argv[2]
+        target_object = sys.argv[3]
+        git_tempdir = sys.argv[4] if len(sys.argv) > 4 else "/tmp"
+
+        result = trace_docstring_path(instance_id, test_function_name, target_object, git_tempdir)
+        print(result)
+    else:
+        # Run self-test
+        success = test_trace_docstring_path()
+        print(f"Self-test {'passed' if success else 'failed'}")
```