Mendel Gödel Machine: Comparative Evolution Enables State-of-the-Art Self-Improving Coding Agents

Anonymous ACL submission

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 strictly faster and better convergence to oracle over single-trajectory baselines. Experiments on SWE-bench and Polyglot confirm MGM's consistent improvement in performance and efficiency that matches state-of-the-art results. 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 idea (Yang et al., 2024a; 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 code base 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 actions.

However, progress so far has focused on improving the archive storing agents generated and traces evaluated, and the process sampling which agent to evaluate or edit next. The *self-modification* process, during which the agent edits its own source code, has remained essentially underexplored: each self-modification step at a moment is conditioned only on one agent's single trajectory (typically a recent failure) on one task. The expanding archive, which records all agent variants ever generated together with their behavior on all tasks evaluated, is utilized only 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 the same agent is evaluated across multiple tasks, its joint pattern of successes and failures forms a reaction norm (Woltereck, 1909; Pigliucci, 2001): a stable, genotype-specific profile of how performance varies across environments. Failure modes recurring across tasks indicate more plausibly genotype-level defects rather than task-specific accidents. Second, when different agents have attempted the same task, their contrastive traces reveal transferable behavioral traits, conditioning edits on which can enable targeted ability transfer across lineages and reduces redundant exploration.

Analogous to Mendelian genetics, which isolates heritable effects through controlled comparisons, we introduce the **Mendel Gödel Machine (MGM)**. As shown in Figure 1, we structure self-modification into three operators: the *clonal mu-*

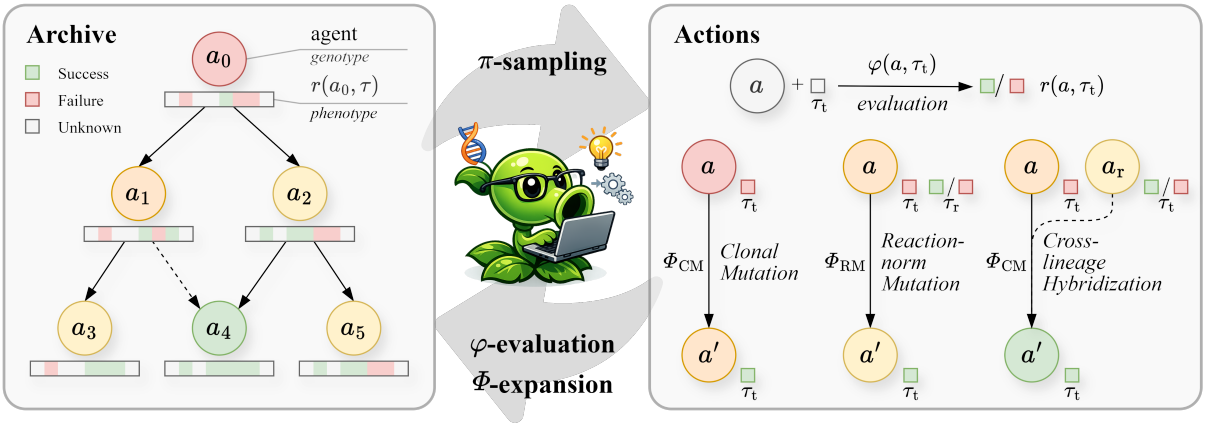


Figure 1: **Mendel Gödel Machine**. MGM organizes self-modification via controlled inheritance based on evidence across tasks and lineages. The archive maintains a lineage tree of agent variants, each stores its source code as genotype and evaluation outcomes as phenotype. Each iteration applies π-sampling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

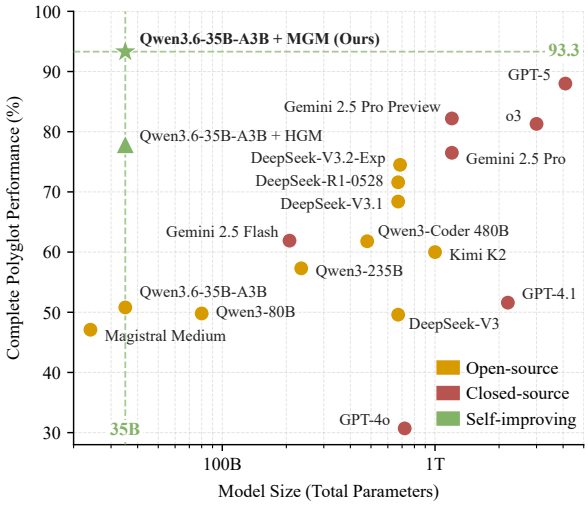


Figure 2: **Complete Polyglot 225-task performance of frontier models and self-improving coding agents.** MGM surpasses state-of-the-art closed-source GPT-5 by running on 35B open-source Qwen3.6 with $\sim$117$\times$ less parameters. Closed-source model sizes are estimation reported by Li (2026).

tation stays as the standard single-agent single-trajectory self-modification; the *reaction-norm mutation* edits an agent conditioned on its trajectories from multiple tasks; and the *cross-lineage hybridization* edits an agent conditioned on a reference agent's trajectory on the same task. Both strategies newly introduced reuse trajectories already accumulated during routine archive evaluation, hence require no additional task evaluations.

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 the genotype space, and self-improvement progress is measured as reduction in Hamming distance to an oracle genotype. We prove that MGM yields a strictly faster expected convergence than single-trajectory baselines, and validate this in controlled Monte Carlo surrogate simulations.

Experiments on Polyglot (Gauthier, 2024) and SWE-bench (Jimenez et al., 2024) show consistent gains over baselines in both performance and efficiency. As Figure 2 shows, MGM advances Qwen3.6-35B-A3B on full Polyglot-225 from 50.8 to 93.3, surpassing closed-source GPT-5 with $\sim$117$\times$ fewer parameters, purely through evolving the agent's scaffolding, demonstrating that MGM's richer comparative conditioning can give rise to state-of-the-art self-improving coding agents.

2 Preliminaries

We denote an executable coding agent $a \in \mathcal{A}$ with its auxiliary scaffolding as its 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

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

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

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

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

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

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

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

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

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

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

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

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

HGM samples expansion candidates from

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

3 Mendel Gödel Machine

MGM develops the aforementioned tree-search framework with the general idea 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 isolating heritable effects through controlled comparisons, we design self-modification operators as diagnostic processes that ask the selected agent to make general improvement in 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

$$E_{\mathrm{CM}}(i, \tau) = \{(\varphi(a_i, \tau), r(a_i, \tau))\}. \tag{1}$$

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

$$a' \leftarrow \Phi_{\mathrm{CM}}(a_i, E_{\mathrm{CM}}). \tag{2}$$

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 design reaction-norm mutation Φ_{RM} as 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 revealing 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

$$|S_i| \geq m_{\mathrm{RM}}, \tag{3}$$

and there exist at least one failed trajectory on a target task $\tau_{\mathrm{t}} \in S_i$. The reference task τ_{r} may be any other trajectory in S_i, with preference given to failed ones when present. The evidence

$$\begin{aligned} E_{\mathrm{RM}}(a_i, \tau_{\mathrm{t}}, \tau_{\mathrm{r}}) = \{&\big(\varphi(a_i, \tau_{\mathrm{t}}), r(a_i, \tau_{\mathrm{t}})\big), \\ &\big(\varphi(a_i, \tau_{\mathrm{r}}), r(a_i, \tau_{\mathrm{r}})\big)\}. \end{aligned} \tag{4}$$

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

$$a' \leftarrow \Phi_{\mathrm{RM}}(a_i, E_{\mathrm{RM}}). \tag{5}$$

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:

$$\begin{aligned} &\exists j \neq i, \; \exists \tau_{\mathrm{t}} \in S_i \cap S_j \\ &\quad \text{s.t.} \quad \neg\big(r(a_i, \tau_{\mathrm{t}}) = 1 \wedge r(a_j, \tau_{\mathrm{t}}) = 1\big). \end{aligned} \tag{6}$$

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}, the differences in their genotypes is used as corrective signals to guide a_{t}'s self-modification. The evidence is

$$\begin{aligned} E_{\mathrm{CH}}(a_{\mathrm{t}}, a_{\mathrm{r}}, \tau_{\mathrm{t}}) = \{&(\varphi(a_{\mathrm{t}}, \tau_{\mathrm{t}}), r(a_{\mathrm{t}}, \tau_{\mathrm{t}})), \\ &(\varphi(a_r, \tau_{\mathrm{t}}), r(a_r, \tau_{\mathrm{t}}))\}. \end{aligned} \tag{7}$$

where a_p is the primary agent and a_q is the reference agent. The child is attached to the primary lineage:

$$a' \leftarrow \Phi_{\mathrm{CH}}(a_p, E_{\mathrm{CH}}). \tag{8}$$

Hybridization is diagnostic rather than mechanical. 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 primary agent's own code base. This encourages the failing lineage to derive and inherit the genuine improvement rather than task-specific behaviors.

3.2 Task Sampling

MGM additionally maintains a global pool

$$\mathcal{P}_t = \bigcup_{i \in \mathcal{V}_t} F_i, \tag{9}$$

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 actions. Instead, it changes 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 pre-determined weight to tasks in $\mathcal{P}_t$:

$$w_i(\tau) = \begin{cases} \beta_{\text{fail}}, & \tau \in \mathcal{P}_t, \\ 1, & \tau \notin \mathcal{P}_t, \end{cases} \qquad \tau \notin S_i, \tag{10}$$

where β_{fail} is the failed-pool boost.

This task sampling design concentrates 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, facilitating transferable or missing behavioral traits to be discovered more efficiently.

3.3 Strategy Selection

MGM inherits HGM's Thompson-sampling π policy, which at each step chooses between initiating a φ evaluation for an existing node or a Φ expansion of the tree, using 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}) becomes eligible if the selected agent has at least one failed task, $F_i \neq \varnothing$. Φ_{RM}) is eligible if the agent has been evaluated on at least m_{RM} distinct tasks (Equation 3), and there exist two distinct tasks $\tau_{\mathrm{t}}, \tau_{\mathrm{r}} \in S_i$ with at least one failure. Φ_{CH}) is eligible if agents from two lineages has attempted a shared task that is not solved by both (Equation 6).

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

$$\Pr(\sigma \mid i) = \frac{\lambda_\sigma}{\sum_{\sigma' \in \Omega_i} \lambda_{\sigma'}}, \qquad \sigma \in \Omega_i. \tag{11}$$

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

$$a' \leftarrow \Phi_\sigma(a_{\mathrm{t}}, E_\sigma). \tag{12}$$

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 pseudo-code of MGM is provided in the Appendix E.5.

4 Simulations

To validate MGM's design decisions in isolation of implementation details and benchmark choices, we instantiate surrogate models for Section 2 and 3 and compare all three methods under controlled environments in a Monte Carlo simulation.

Genotype model. Each agent's genotype can be modeled as a binary vector $\mathbf{g} \in \{0,1\}^L$ with L loci. A fixed oracle genotype $\mathbf{g}^* \in \{0,1\}^L$ represents the optimal program; without loss of generality we set $\mathbf{g}^* = \mathbf{1}$. The performance of $\mathbf{g}$ is measured as its Hamming distance to the oracle target,

$$d(\mathbf{g}) \;=\; \sum_{\ell=1}^{L} \mathbf{1}[g_\ell \neq g_\ell^*]\,, \tag{13}$$

where $d(\mathbf{g}) = 0$ if and only if $\mathbf{g} = \mathbf{g}^*$. This oracle serves as the zero-error lower bound.d_0 loci are initialized to 0 (mismatch), so every run begins at edit distance d_0, representing the exact loci that self-improvement should modify to completely solve the benchmark. The task pool contains N tasks, each examining k loci; a task is solved only when all k sampled loci are correct, as how the agent is only able to solve the task when it has equipped and learned to use all necessary tools correctly in its

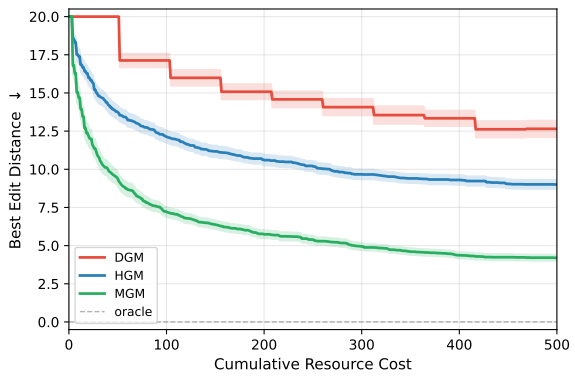


Figure 3: **Simulated evolution performance with cumulative budget spent.** Results averaged across random seeds, with shaded bands representing 95 % CI. MGM achieves the fastest convergence and highest performance as Φ_{RM} and Φ_{CH} become increasingly eligible throughout the run.

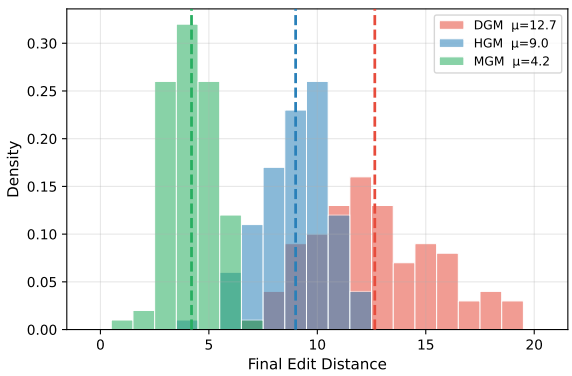


Figure 4: **Simulated evolution performance distribution.** Dashed lines mark per-method means. MGM achieves the leftmost mean and the most concentrated distribution, indicating that higher-quality diagnostic operators reduce both the average and variance of residual errors.

scaffold. The resulting per-task success probability

$$P(r{=}1 \mid d) = \left(\frac{L-d}{L}\right)^k \tag{14}$$

equals $\big((L-d_0)/L\big)^k$ at the start of each run. All parameter values are listed in Table 6.

Simulation and evaluation. The simulation isolates axes on which the three methods differ. For budget allocation, DGM assigns a fixed n_{eval} evaluations to every population member per generation, distributing budget uniformly regardless of node quality. HGM uses adaptive allocation with parameters concentrating evaluations on promising nodes and triggering edits when observed failure rates exceed the threshold. DGM and HGM use only Φ_{CM}. MGM additionally employs Φ_{RM} and Φ_{CH} when the availability conditions hold. All three edit operators incur configurable costs, and here we implement $c_{\mathrm{CM}} = c_{\mathrm{RM}} = c_{\mathrm{CH}} = c_{\varphi}$ to control that any observed difference between MGM and others is attributable to richer comparative evidence signals that can brings higher per-locus fix probability $p_f^{\mathrm{RM}} = p_f^{\mathrm{CH}} > p_f^{\mathrm{CM}}$.

Each method runs for a fixed total budget of B resource units with n_{seeds} independent Monte Carlo seeds; at each seed a fresh task pool and initial genotype are regenerated. The minimum edit distance across all active nodes is recorded at evenly spaced budget checkpoints; final performance is reported as mean $\pm$ 95 % CI across seeds. A separate parameter sweep varies the initial edit distance d_0 and the fix-probability advantage ratio $\rho = p_f^{\mathrm{RM}}/p_f^{\mathrm{CM}} = p_f^{\mathrm{CH}}/p_f^{\mathrm{CM}}$ to test robustness of the ordering. Detailed configurations for our simulation can be found in Table 6.

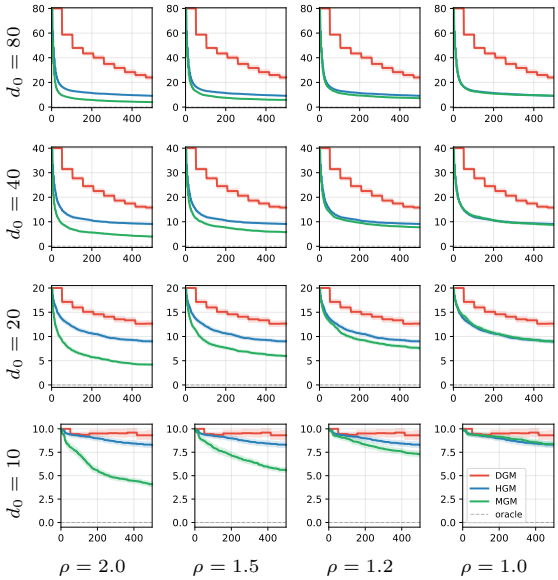


Figure 5: **Parameter sensitivity of simulated evolution.** Rows vary the initial edit distance d_0; columns vary the fix-probability advantage ratio ρ. Each panel shows the best edit distance evolved with cumulative cost. MGM matches HGM at ρ=1.0 and consistently outperforms others as ρ grows, and the trend is more pronounced at small d_0, confirming the benefit of higher-quality diagnostic signals.

Interpretation. Figure 3 tracks the best edit distance over time, confirming that MGM outperforms other baselines in terms of both performance and convergence speed. Figure 4 shows the distribution of final performance across seeds, which MGM achieves the lowest mean and tightest spread, while other baselines' broader distribution reflects wastes and unstableness in their budget allocation.

The parameter sensitive results in Figure 5 and Figure 13 in Appendix tests the robustness of MGM across a broad range of initial difficulties d_0 and operator-quality ρ settings. MGM consistently outperforms other baselines across all

d_0 and $\rho > 1.0$ settings, demonstrating reliable gains no matter the are becomes easier or harder. When $\rho = 1$, where richer operators provide no fix-quality advantage, MGM matches HGM exactly, confirming that their difference under this setting is driven solely by their difference under this setting is driven solely by MGM's better diagnostic quality. As ρ increases, MGM's advantage grows steadily, and this trend is especially pronounced at smaller d_0, where less but more precise self-improvement is required and the higher-quality diagnostic signals of MGM's operators translate into clearer performance gains.

5 Experiments

We evaluate MGM on challenging software engineering benchmarks to answer these four research questions: **(1)** Does MGM exhibit better performance under same budget? **(2)** Does MGM improve efficiency for reaching a similar level of performance? **(3)** Can agents evolved by MGM generalize to other challenging tasks? **(4)** 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), and Polyglot (Gauthier, 2024). In all settings, agents are not given access to private test cases or test results during evolution. To ensure fair comparisons and control computational cost, unless otherwise specified, all results reported in this section are evaluated on two 60-task subsets of SWE-bench Verified and Polyglot, which are respectively derived from the first two stages of DGM's progressive evaluation following Wang et al. (2026); Zhang et al. (2026a).

5.1 Performance

To assess whether MGM exhibits greater self-improvement potential than HGM, we evaluate both methods on 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, which obtains 50.8% accuracy on Polyglot-60. We report the accuracy of the best-belief agent evolved by each method.

Table 1 shows that, despite starting from the same initial performance, MGM improves to 93.2% accuracy, whereas HGM reaches 77.9% under

Metric	Polyglot	
	HGM	**MGM**
Initial accuracy	50.8	50.8
Evolved accuracy	$77.9^{+27.1\%}$	$\mathbf{93.2}^{+42.4\%}$
Relative gain	↑ 53.35%	↑ **83.46%**
# Actions	$200\varphi + 24\Phi$	$200\varphi + 24\Phi$
Time	44.20 h	**40.14 h**

Table 1: **Self-improving performance on Polyglot.** One hour reported in the table corresponds to one hour of CPU wall-clock time with four H100 GPUs.

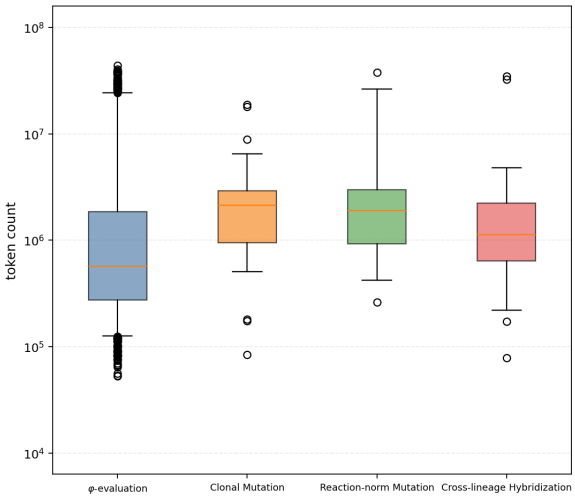


Figure 6: **Token costs of each action for HGM and MGM evolved on Polyglot.** The distribution lies in the same order of magnitude, confirming that all φ and Φ actions incur comparable average token costs.

the same evaluation budget. This result suggests that MGM has a higher performance ceiling and stronger long-term potential for self-improvement. Moreover, MGM attains this improvement with slightly lower training time cost, controlling that its advantage is not a consequence of additional computation. We further analyzed the token consumption of each step in the experiments. Figure 6 shows that the three evolutionary strategies have comparable average token costs, indicating that the observed performance gains are not achieved by increasing token expenditure. Evaluation on the full 225-task Polyglot benchmark (Figure 2 and Appendix D.2) confirms the same results.

5.2 Efficiency

To demonstrate the efficiency of MGM, we conducted experiments on SWE-bench verified to compare the computational budget required by each method to achieve a similar level of performance improvement. We adopt Qwen3-Coder-Next (Cao et al., 2026) as the backbone LLMs. For all methods, we start from the same initial ancestor, which

Metric	SWE-bench Verified	
	HGM	**MGM**
Initial accuracy	33.3	33.3
Evolved accuracy	$40.0^{+6.7\%}$	$\mathbf{41.7}^{+8.4\%}$
Relative gain	↑ 20.12%	↑ **25.23%**
# Actions	$160\varphi + 21\Phi$	$\mathbf{117\varphi + 17\Phi}$
Time	117.69 h	**86.05 h**

Table 2: **Self-improving performance on SWE-bench Verified.** One hour reported in the table corresponds to one hour of CPU wall-clock time with eight H100 GPUs.

achieves 33.3% on the SWE-bench Verified-60. We compare the training budget used and the hours when they achieve roughly the same performance.

The results are shown in Table 2. MGM achieves an evolved accuracy of 41.7%, slightly surpassing HGM's 40.0%. Meanwhile, MGM utilizes only 73.1% φ-evaluation, 81.0% Φ-expansion, and 73.1% computation time in total compared to HGM. This demonstrates that MGM attains comparable performance while being substantially more computationally efficient, highlighting the practical advantage of its improved diagnostic and search dynamics.

5.3 Generalization

A key question for self-improving agents is whether the evolved improvements are genuinely reusable, or merely overfit to the benchmark used during evolution. To test this, we evaluate the best MGM-evolved agent obtained from Polyglot on SWE-bench Pro (Deng et al., 2025). Importantly, the agent is evolved only on Polyglot and has never seen any SWE-bench Pro task during the self-improvement process. Therefore, SWE-bench Pro serves as a strictly held-out benchmark for measuring cross-benchmark generalization.

We choose SWE-bench Pro for two reasons. First, it contains challenging long-horizon software engineering tasks that are substantially different from the multilingual coding problems in Polyglot. Second, due to its license and data release setting, SWE-bench Pro provides a cleaner evaluation protocol with minimal risk of benchmark contamination or training-data leakage. This makes it particularly suitable for testing whether the capabilities evolved on Polyglot transfer to genuinely unseen software-engineering problems, rather than reflecting memorization of benchmark-specific instances. The SWE-bench Pro task list is reported in Appendix F.

The results are shown in Table 3. The initial

Agent	SWE-bench Pro Acc.	Relative Gain
Initial	16.7	–
MGM-Evolved	**26.7** +10%	↑ **60%**

Table 3: **Generalization performance on SWE-bench Pro from Polyglot.**

agent achieves 16.7% accuracy on SWE-bench Pro. After being evolved by MGM on Polyglot, the resulting agent achieves 26.7% accuracy on SWE-bench Pro, corresponding to a +10.0 absolute percentage-point improvement. Relative to the initial agent, this is a 60% gain. This result provides evidence that MGM does not simply memorize or overfit to the tasks encountered during evolution. Since the MGM-evolved agent is trained only on Polyglot but evaluated on unseen SWE-bench Pro tasks under a contamination-resistant setting, the improvement is more likely to come from more general agent-level capabilities rather than benchmark-specific shortcuts.

The fact that MGM transfers from Polyglot to SWE-bench Pro confirms that self-improving agent with MGM can discover reusable workflow-level improvements rather than narrow task-specific heuristics. Together with the strong in-domain improvement on Polyglot, this result supports the central claim that MGM produces agents with substantially better generalization ability through comparative self-improvement. Appendix A provides qualitative evidence that these gains correspond to reusable workflow-level skills rather than task-specific patches.

5.4 Ablation Study

To understand the contribution of each component in MGM, we conduct an ablation study on Polyglot-60. All variants are evaluated under the same computational budget of 200 evaluations, with two parallel workers enabled during evolution. We use Qwen3.6-35B-A3B (Qwen Team, 2026) as the backbone LLM for all settings. Each variant starts from the same initial ancestor agent, which achieves 50.8% accuracy on Polyglot-60. We compare the full MGM with two ablated variants: MGM without Reaction-norm Mutation (RM) and MGM without Cross-lineage Hybridization (CH). We report the evolved accuracy of the best-belief agent, the relative gain over the initial ancestor, and the total training cost. Appendix B visualizes the corresponding evolution trees for the full model, HGM, and ablations. As shown in Table 4, the full MGM achieves the best performance,

Metric	MGM		
	Full	**w/o RM**	**w/o CH**
Initial acc.	50.8	50.8	50.8
Evolved acc.	**93.2** + 42.4%	79.7 + 28.9%	74.6 + 23.8%
Relative gain	↑ **83.46%**	↑ 56.89%	↑ 46.85%
# Actions	$200\varphi + 24\Phi$	$200\varphi + 24\Phi$	$200\varphi + 24\Phi$
Time	**40.14 h**	43.01 h	39.95 h

Table 4: Ablation study of MGM on Polyglot.

improving from 50.8% to 93.2% accuracy. Removing RM leads to a clear performance drop, with the evolved accuracy decreasing to 79.7%. Removing CH causes an even larger degradation, reducing the evolved accuracy to 74.6%. This suggests that RM plays an important role in guiding the self-improvement process toward more promising agents, and 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 difference in computational cost.

6 Related Work

LLM agent systems and automated agent-design methods study how prompts, tools, control flow, and workflows can be manually engineered or automatically searched as executable scaffolds(Yao et al., 2023; Schick et al., 2023; Wu et al., 2023; Yang et al., 2024a; 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 searchable program space with 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., 2024; Xia et al., 2025). Robeyns et al. (2025) shows that a coding agent can evaluate itself, edit its own codebase, and improve over iterations. Zhang et al. (2026a) extends this process into open-ended evolution by maintaining an archive of self-modified agents, while Wang et al. (2026) improves archive expansion by estimating the future self-improvement potential of clades. MGM follows this archive-based self-improvement setting, but targets a complementary bottleneck of improving what evidence is used for the expansion itself. By reusing existing trajectories across tasks and lineages, MGM makes each self-modification step more informative without requiring additional task evaluations.

Software-engineering benchmarks such as SWE-bench (Jimenez et al., 2024), SWE-bench Pro (Deng et al., 2025), and Polyglot (Gauthier, 2024) evaluate repository-level repair, long-horizon reasoning, and cross-language coding ability. These benchmarks are utilized to test whether coding agents discovers reusable workflow-level improvements.

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

7 Conclusion

We introduced the Mendel Gödel Machine (MGM), a self-improving coding-agent framework that makes self-modification more informative by fully utilizing existing evaluation trajectories. MGM augments archive-based agent evolution with three evidence-driven operators: clonal mutation, reaction-norm mutation and cross-lineage hybridization. These operators reuse existing trajectories without additional task evaluations. Controlled simulations and experiments on SWE-bench Verified and Polyglot show that MGM improves both efficiency and final performance over HGM. Ablations further confirm that both comparative operators contribute to the gains, while held-out evaluation on SWE-bench Pro suggests that the evolved improvements transfer across different tasks.

Limitations

Self-improving coding-agent experiments remain expensive. Even though MGM reuses existing trajectories and does not require additional task executions for its comparative operators, evolving and evaluating coding agents over repository-level tasks still requires substantial wall-clock and GPU resources. This limits the number of independent evolution seeds and the breadth of hyperparameter sweeps we can report.

MGM is history-dependent. Its main advantage comes from comparative evidence accumulated in the archive: reaction-norm mutation requires multiple trajectories from the same agent, and cross-lineage hybridization requires overlapping tasks across different lineages. When the archive is still

small, task overlap is sparse, or informative failures have not yet been observed, MGM has limited comparative evidence and may behave similarly to single-trajectory self-modification baselines. The failed-task pool partially mitigates this issue by increasing diagnostic overlap.

Ethics Consideration

MGM advances the capability of self-improving coding agents, which raises considerations specific to autonomous systems that can modify their own source code. Self-evolving agents present a qualitatively different risk profile from static models. In our experiments, all self-modification and evaluation steps are executed within isolated Docker containers with no network access and read-only mounts of the host file system, following the safety protocol established by prior work. Any deployment of MGM-style agents beyond this sandboxed coding environment requires careful re-evaluation of the execution boundary. In particular, granting a self-modifying agent access to production systems, external APIs, or persistent storage outside of controlled containers could allow unintended or harmful modifications to propagate in ways that are difficult to audit or reverse. Practitioners who build on MGM should retain the isolated execution model and subject any extension to a broader action space to explicit safety review before deployment.

The benchmarks and backbone models used in our experiments may carry harmful contents. SWE-bench, SWE-bench Pro, and Polyglot are drawn from public open-source repositories, and the backbone LLMs are openly released models that have been pre-trained on large-scale web corpora. Both the training data of these models and the benchmark repositories may contain codes potentially harmful and may reflect demographic or geographic biases present in open-source software communities, including overrepresentation of certain languages and problem domains. We did not conduct dedicated bias audits of the evolved agents, and potential fairness or safety issues inherited from the underlying models may persist. In addition, our work is scoped strictly to studying self-evolution of coding agents, and the evolved agents are not equipped with tools to act outside a software repository. Appropriate task-specific evaluations is necessary before deploying evolved agents in contexts beyond general-purpose coding assistance.

References

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. 2026. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Jeff Clune. 2020. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *Preprint*, arXiv:1905.10985.

Antoine Cully. 2021. 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, page 84–92. ACM.

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, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

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, and 3 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *Preprint*, arXiv:2509.16941.

Jordan S. Ellenberg, Cristofero S. Fraser-Taliente, Thomas R. Harvey, Karan Srivastava, and Andrew V. Sutherland. 2025. Generative modeling for mathematical discovery. *Preprint*, arXiv: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, and 8 others. 2026. 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*. Survey Certification.

Paul Gauthier. 2024. o1 tops aider's new polyglot leaderboard. `https://aider.chat/2024/12/21/polyglot.html`. 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. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

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

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

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. 2022. Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *Preprint*, arXiv:2205.00445.

Bojie Li. 2026. Incompressible knowledge probes: Estimating black-box llm parameter counts via factual capacity. *Preprint*, arXiv:2604.24827.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

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, and 109 others. 2025. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *Preprint*, arXiv:2506.13585.

Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *Preprint*, arXiv: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. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *Preprint*, arXiv:2506.13131.

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

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

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. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *Preprint*, arXiv:2505.20286.

Qwen Team. 2026. Qwen3.6-35B-A3B: Agentic coding power, now open to all.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. 2025. A self-improving coding agent. In *Scaling Self-Improving Foundation Models without Human Supervision*.

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

Jurgen Schmidhuber. 1987. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May.

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

Jürgen Schmidhuber. 2007. Gödel machines: Fully self-referential optimal universal self-improvers. In Ben Goertzel and Cassio Pennachin, editors, *Artificial General Intelligence*, pages 199–226. Springer, Berlin, Heidelberg.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

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, and 467 others. 2026. Openai gpt-5 system card. *Preprint*, arXiv: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, and 307 others. 2026. Kimi k2.5: Visual agentic intelligence. *Preprint*, arXiv:2602.02276.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. 2026. 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*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Richard Woltereck. 1909. Weitere experimentelle Untersuchungen über Artveränderung, speziell über das Wesen quantitativer Artunterschiede bei Daphniden. *Verhandlungen der deutschen zoologischen Gesellschaft*, 19:110–173.

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. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *Preprint*, arXiv:2308.08155.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

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

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *Preprint*, arXiv:2410.03859.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. 2026a. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*.

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

Appendix Contents

A Discussion **12**
- A.1 When Does Hybridization Help? . 12
- A.2 Are skills evolved from MGM really more general? 12

B Visualizations of Evolving Trees. **13**
- B.1 Evolving Tree of MGM 13
- B.2 Evolving Tree of HGM 13
- B.3 Evolving Tree of Ablation Studies 13

C Additional Related Work **14**
- C.1 From Agent Design to Inherited Self-Modification 14
- C.2 Runtime Adaptation versus Training-Time Self-Evolution . . . 15
- C.3 Open-Ended Search and Algorithm Discovery 16
- C.4 Benchmark Coverage and Evaluation Motivation 16

D Additional Results **17**
- D.1 Additional Simulation Results . . 17
- D.2 Full Evaluation on Polyglot 17

E Experiment Details **17**
- E.1 Hyper-parameter Settings 17
- E.2 Theoretical Simulation Parameters 17
- E.3 Primary-Lineage Selection in Cross-Lineage Hybridization . . . 18
- E.4 Prompt Design 19
- E.5 Pseudocode of Mendel Gödel Machine 25

F Benchmark Details **26**
- F.1 SWE-bench Pro Tasks 26

G Best Discovered-Agents **27**
- G.1 MGM on Polyglot 27
- G.2 MGM on SWE-bench Verified . . 28

A Discussion

A.1 When Does Hybridization Help?

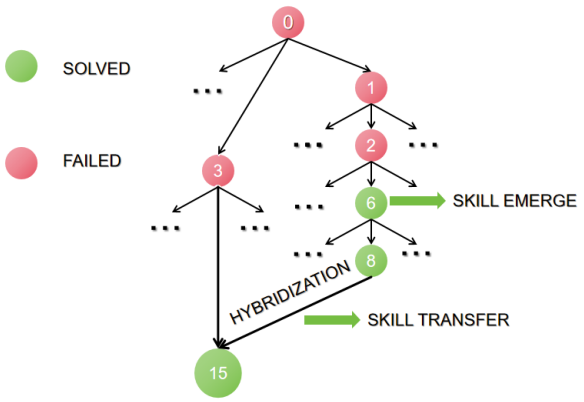


Figure 7: Illustration of **Cross-lineage Hybridization**. Nodes are colored by their outcome on the shared diagnostic task `javascript__queen-attack`: red nodes fail and green nodes solve it.

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 Are skills evolved from MGM really more general?

The visualization suggests that MGM does not merely produce more changes. It produces changes that concentrate around a reusable workflow-level capability. Most MGM points lie in a

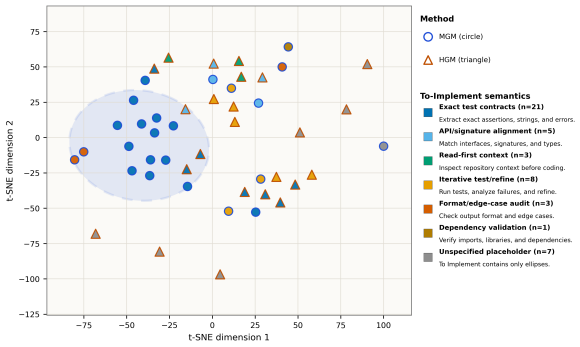


Figure 8: This Figure provides a content-level view of the evolved "To Implement" descriptions from MGM and HGM. Each point corresponds to one evolved skill or workflow change, embedded by the semantics of its To Implement text. The marker shape indicates the evolution method, while color indicates the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

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 describe a general procedure: read tests and existing interfaces, extract the expected contract, implement against that contract, and verify exact agreement before finalizing the patch. This kind of skill 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 "generality" 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.

B Visualizations of Evolving 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 ($\Phi_{\text{CM}}/\Phi_{\text{RM}}/\Phi_{\text{CH}}$). Starred nodes mark the representative agent chosen for each experimental setting.

B.1 Evolving Tree of MGM

The Figure 9 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. Edge colors indicate which operator generated each child (red, orange, and purple, respectively). Cross-lineage Hybridization nodes display the context donor as ctx #id. Node fill color encodes evaluation accuracy. The tree contains multiple parallel lineages, and purple edges highlight cross-archive transfer of capabilities. The starred node #16 is the best-belief agent (utility ≈ 0.92).

B.2 Evolving Tree of HGM

Figure 10 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. Consequently, the tree contains no Reaction-norm Mutation or Cross-lineage Hybridization edges, and no context-donor annotations appear. Compared with full MGM, HGM still explores multiple branches, but it lacks the additional comparative evidence channels represented by the orange and purple edges. Although node #24 appears greener in the visualization, it has only been evaluated once, so its color reflects a high-variance estimate rather than a reliable final selection. After a full-test comparison, the starred node #18 is selected as the final best-belief agent.

B.3 Evolving Tree of Ablation Studies

The Figure 11a shows the ablation that removes Reaction-norm Mutation. Clonal Mutation and Cross-lineage Hybridization remain enabled, so the

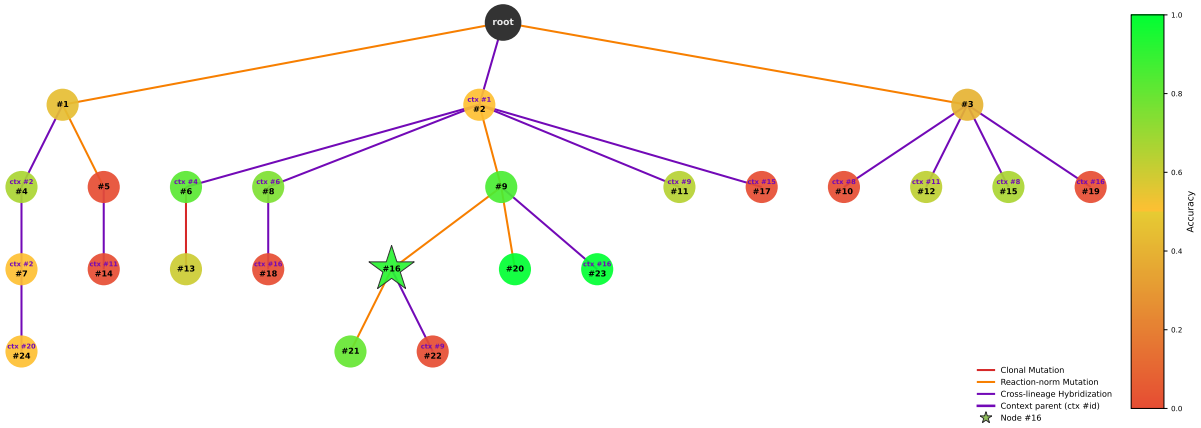


Figure 9: **MGM evolution tree** under the full operator mixture (24 nodes, 200 task evals). Nodes are colored by evaluation accuracy (red→green). Edges indicate the operator that produced the child.

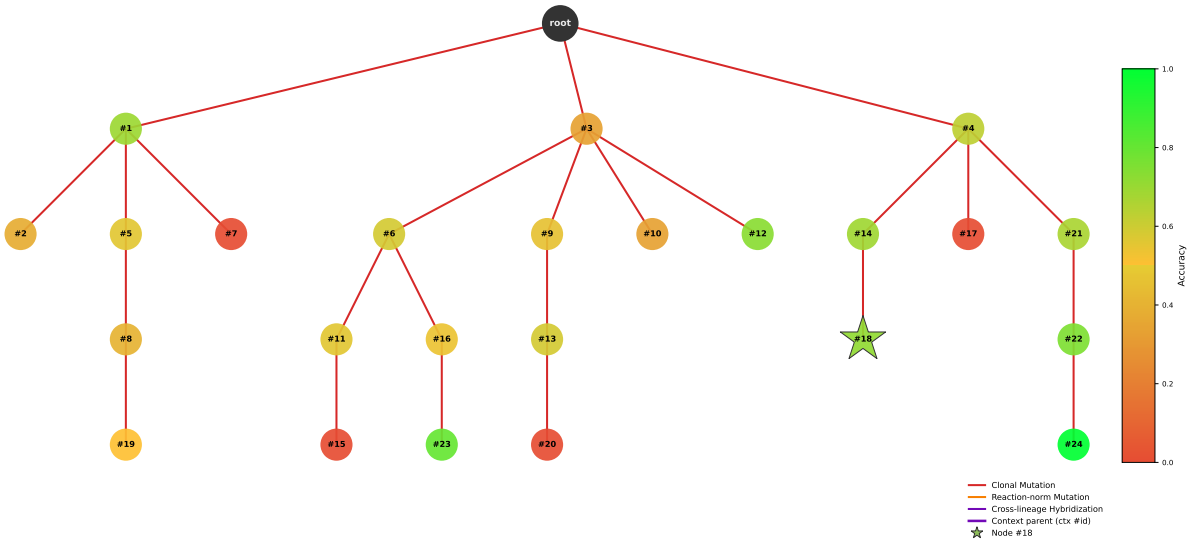


Figure 10: **HGM evolution tree** with Clonal Mutation only (24 nodes, 200 task evals).

tree contains only red and purple edges. Hybridization nodes still annotate context donors. The run uses 24 nodes and a 200-evaluation budget. The starred node (#14) completed full evaluation on all 59 tasks (47/59 resolved, utility ≈ 0.84).

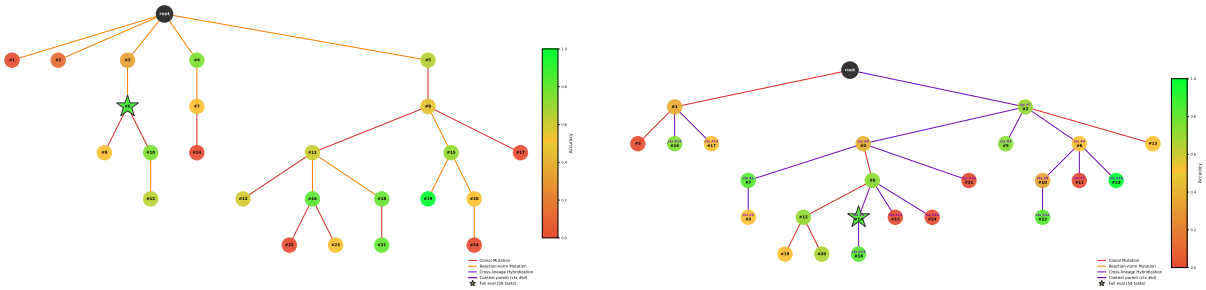
The Figure 11b shows the ablation that removes Cross-lineage Hybridization. Clonal Mutation and Reaction-norm Mutation remain active, so the tree contains only red and orange edges, with no ctx labels. All improvement is confined to within-lineage evolution. The starred node is #6,

C Additional Related Work

As foundation models continue to advance, including DeepSeek-AI et al. (2025); **?**); **?**); Team et al. (2026); 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).

C.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).



(a) **Remove-RM ablation** (Clonal Mutation and Cross-lineage Hybridization only; 24 nodes, 200 task evals).

(b) **Remove-CH ablation** (Clonal Mutation and Reaction-norm Mutation only; 24 nodes, 200 task evals).

Figure 11: Ablation evolution trees for MGM variants.

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., 2024a). HyperAgent instead explores a multi-agent decomposition of software engineering work, assigning specialized roles such as planning, navigation, editing, and execution to different agents (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.

C.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 evo-

lution 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.

C.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 and 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.

C.4 Benchmark Coverage and Evaluation Motivation

SWE-bench is the canonical benchmark for repository-level issue resolution. Unlike function-level benchmarks such as HumanEval or MBPP, 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 (Yang et al., 2024b).

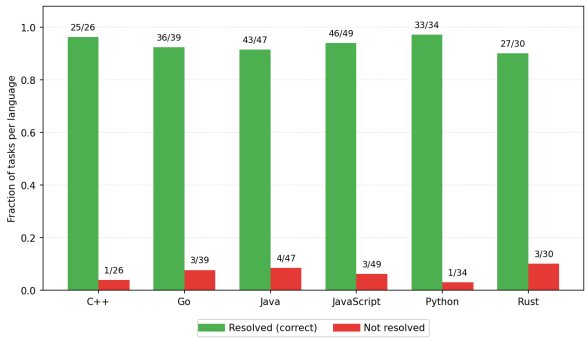


Figure 12: Full Polyglot evaluation of the best MGM-discovered agent. The agent is evaluated after evolution without further self-modification and solves 210 out of 225 tasks overall. Bars report the fraction of resolved and unresolved tasks within each language. Labels show counts per language. MGM 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.

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.

D Additional Results

D.1 Additional Simulation Results

Figure 13 reports tests the robustness of MGM. MGM consistently outperforms other baselines across all d_0 and $\rho > 1.0$ settings, and matches HGM exactly when $\rho = 1$ when there's no fix-quality advantage for richer operators, confirming that the gain from budget allocation drives the gap to DGM, and MGM's better diagnostic quality mainly drives its gap to HGM.

D.2 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.

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

Table 5: Hyper-parameter settings used in our experiments.

As shown in Figure 12, 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.

E Experiment Details

E.1 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 5.

E.2 Theoretical Simulation Parameters

Table 6 lists all parameters used in the Monte Carlo simulation of Section 4. All three edit operators are assigned equal cost ($c_{\text{CM}} = c_{\text{RM}} = c_{\text{CH}} = c_\tau$), so that any performance gap between HGM and MGM is attributable solely to edit-diagnostic quality. The fix-probability advantage ratio is $\rho = p_f^{\text{RM}}/p_f^{\text{CM}} = p_f^{\text{CH}}/p_f^{\text{CM}} = 2.0$ in the main simulation; the parameter sweep varies both ρ and d_0 to test robustness

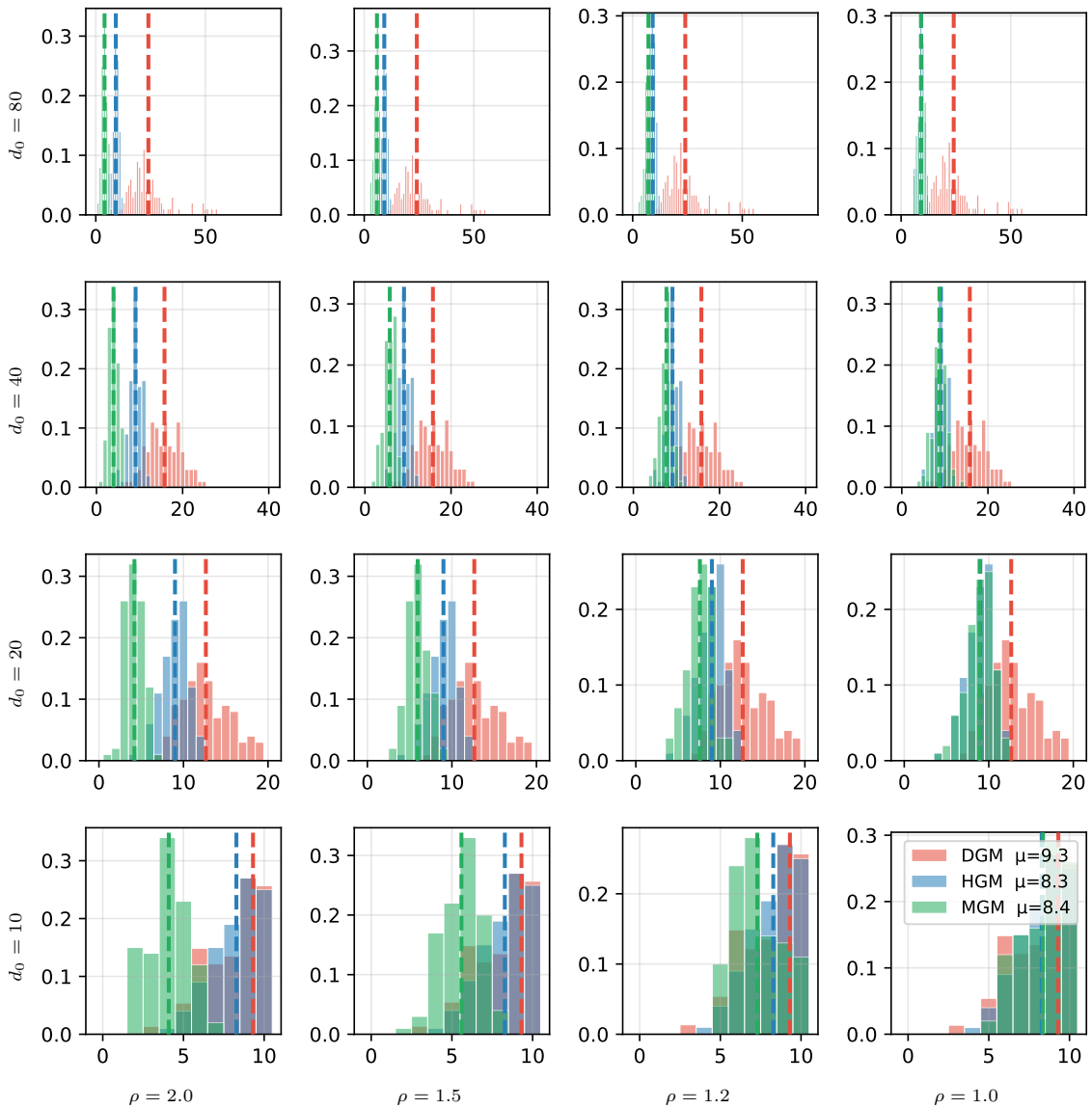


Figure 13: **Parameter sensitivity of final performance distribution.** Rows vary the initial edit distance d_0; columns vary the fix-probability advantage ratio ρ. Each panel shows the distribution of the simulated performance of evolution. MGM matches HGM at ρ=1.0 and consistently outperforms others as ρ grows, and the trend is more pronounced at small d_0, confirming the benefit of higher-quality diagnostic signals.

of the ordering.

E.3 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.

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

$$E_C(i, j, \tau^\star) = \{(\phi(a_i, \tau^\star), r(a_i, \tau^\star)), (\phi(a_j, \tau^\star), r(a_j, \tau^\star))\}. \quad (15)$$

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:

$$a' \leftarrow \Phi_C(a_p, E_C),\ a' \in \text{children}(a_p). \quad (16)$$

This direction is intentional: the goal is not to further edit the already successful lineage, but to let

Group	Parameter	Symbol	Value
DNA model	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 6: Theoretical simulation parameter settings (Section 4). 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.

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.

E.4 Prompt Design

In this section, we present the details of **Self-Improvement** prompt we used in the algorithm.

Reaction-norm Mutation & SWE-bench:

```
You are given TWO runs of the coding agent on different GitHub issues using the
same agent codebase. At least one of the runs failed. Your goal is to identify
a systematic weakness in the agent that manifests across different tasks,
rather than a fix for one specific issue.

Analyze both scenarios carefully:
- Look for common patterns: Do both runs share similar failure modes, e.g.,
  poor tool usage, wrong debugging strategy, incomplete understanding of the
  codebase?
- Look for contrasting signals: If one run succeeded and the other failed, what
  capability was present in the successful run but missing in the failed one?
- Focus on the agent's general approach and decision-making, not on
  repository-specific knowledge.

CRITICAL -- evolve GENERAL capabilities only:
The whole point of looking at TWO different tasks is to factor out
task-specific accidents. Any improvement you propose must apply broadly across
unseen issues and repositories. Do NOT propose changes that hard-code keywords,
paths, function/class names, framework idioms, or example bugs from these two
tasks. If a candidate improvement only "works" because of details visible in
these logs, discard it and search for the underlying general skill the agent was
lacking.
```

Figure 14: Prompt for reaction-norm mutation on SWE-bench.

Reaction-norm Mutation & Polyglot:

```
"""You are given TWO runs of the coding agent on **different** programming tasks using the **
    same agent codebase**. At least one of the runs failed. Your goal is to identify a **
    systematic weakness** in the agent that manifests across different tasks and languages.

Analyze both scenarios:
- Look for **common patterns** in how the agent approaches problems across languages.
- If one run succeeded and the other failed, identify what capability made the difference.
- Focus on **general coding strategies** that work across C++, Go, Java, JavaScript, Python,
    and Rust.

**CRITICAL  evolve GENERAL capabilities only:**
The improvement must be a transferable, language-agnostic, task-agnostic skill. Reject
    anything tied to a specific algorithm, problem statement, language, file name, or test
    case from these two tasks. The right question is: "what general coding/debugging/planning
     skill would have helped the agent solve *both* tasks AND hundreds of unseen tasks?"

"""
```

Figure 15: Prompt for reaction-norm mutation on Polyglot.

Reaction-norm Json Block:

```
    """
Respond precisely in the following format including the JSON start and end markers:

```json
<JSON>
```

In <JSON>, provide a JSON response with the following fields:
- "task_1_analysis": Analyze how the agent approached Task 1. Summarize the tools used, the
     problem-solving strategy, and the outcome (success or failure). If it failed, identify
     the specific failure mode.
- "task_2_analysis": Analyze how the agent approached Task 2. Same structure as above.
- "common_weakness": Identify the **systematic weakness** that connects the two scenarios.
     This MUST be a *general* agent limitation (e.g., poor error recovery, lack of incremental
      testing, weak codebase navigation, fragile planning, missing self-verification) NOT a
     task-specific issue. If one task succeeded and the other failed, explain why the
     successful approach did not transfer, again at the level of a general skill.
- "potential_improvements": List potential improvements that address the identified common
     weakness. **Every item must be a general capability that would help the agent on
     completely unseen issues, repositories, and languages.** Reject any candidate that
     depends on the specific tasks, files, frameworks, or error messages shown.
- "improvement_proposal": Choose ONE high-impact *general* improvement from the list.
     Describe it in detail. Sanity check: if you removed all references to Task 1 and Task 2,
     would the proposal still make sense and be implementable? It must.
- "implementation_suggestion": Describe what feature or tool to add or change in the agent.
     Prefer extending existing tools over inventing redundant ones. Do NOT hard-code task-,
     repo-, or language-specific strings, paths, or rules.
- "problem_description": Phrase the improvement proposal and implementation suggestion as a
     GitHub issue description so an engineer can implement it as a *reusable, general-purpose*
      capability not a patch for these two tasks.

Your response will be automatically parsed, so ensure that the string response is precisely
     in the correct format. Do NOT include the `<JSON>` tag in your output."""
```

Figure 16: Reaction-norm Json Block

Cross-lineage Hybridization & SWE-bench & ✗✗:

```
    You are given **one** GitHub issue attempted by **two different versions** of the coding
      agent. **Both versions failed** to solve this issue.

Logs are labeled:
- **Primary** (commit: `{primary_commit}`, overall accuracy: {primary_utility:.1\%}) -- the
     version you should optimize. The new improved child agent will be attached to this
     version.
- **Context** (commit: `{context_commit}`, overall accuracy: {context_utility:.1\%}) -- for
     comparison only.

Your goal is a **differential analysis**: compare how the two versions approached the same
     problem, identify which got closer to a solution and why, and propose one improvement
     for the Primary version.

**CRITICAL  propose only GENERAL capabilities:**
Even though both versions failed on this *one* task, your proposal must NOT be a fix for
     this specific task. The shared task is just a probe. Use it to discover a *general*
     missing skill (e.g., better hypothesis testing, better reproduction-script discipline,
     better failure-mode triage, better context navigation, better self-verification) that
     would help the agent on hundreds of unseen issues. Anything tied to this specific repo/
     file/function/error string must be discarded and re-abstracted.

"""
```

Figure 17: Cross-lineage Hybridization & SWE-bench & ✗✗:

Cross-lineage Hybridization & SWE-bench & ✗✓:

```
    """You are given **one** GitHub issue attempted by **two different versions** of the
     coding agent. One version **succeeded** and the other **failed**.

By convention in this self-improvement loop:
- **Primary** (commit: `{primary_commit}`, overall accuracy: {primary_utility:.1\%}) is the
     version that **FAILED** on this shared task. Primary is the version you should optimize,
      and the new improved child agent will be attached as Primary's descendant.
- **Context** (commit: `{context_commit}`, overall accuracy: {context_utility:.1\%}) is the
     version that **SUCCEEDED** on this shared task. It is shown only as a reference donor.

**The objective is capability transfer**: figure out which *general* skill or behavior
     allowed Context to succeed where Primary failed, and propose how to instill that skill
     into Primary so it generalizes.

This is a powerful diagnostic signal: by comparing a successful and failed attempt on the **
     same** issue, you can pinpoint exactly what capability made the difference. But your
     proposal must be a **transferable, general lesson**  not a task-specific fix, not a copy
      of Context's exact actions, and not anything keyed to this repo/file/function/test.

**CRITICAL  evolve GENERAL capabilities only:**
- Do NOT propose memorizing Context's exact tool sequence, exact diff, or exact reasoning
     steps for this issue.
- Do NOT add hard-coded keywords, paths, framework names, or example bugs from this task
     into the agent.
- DO propose a *general* skill (better planning, better test-writing discipline, better self
     -critique, better tool design, better error-recovery loop, better information-gathering
      strategy, etc.) such that Primary, once equipped with it, would solve this task **and**
      many other unrelated tasks.
- Sanity check: if you removed every mention of this specific issue from your proposal,
     would it still be a coherent, implementable improvement? It must be.

"""
```

Figure 18: Cross-lineage Hybridization & SWE-bench & ✗✓

Cross-lineage Hybridization & Polyglot & ✗✗:

```
 """You are given **one** programming task attempted by **two different versions** of the
     coding agent. **Both versions failed**.

Logs are labeled:
- **Primary** (commit: `{primary_commit}`, overall accuracy: {primary_utility:.1%}) --
     optimize this version. The new improved child agent will be attached here.
- **Context** (commit: `{context_commit}`, overall accuracy: {context_utility:.1%}) -- for
     comparison.

Compare how each version approached the same task. Identify which got closer and why, then
     propose one improvement for the Primary version.

**CRITICAL  propose only GENERAL, language-agnostic capabilities:**
This task is just a probe; the improvement must transfer across all of C++, Go, Java,
     JavaScript, Python, and Rust, and across unseen problems. Reject anything tied to this
     specific algorithm, language, file, or test fixture. Search for the *underlying general
      skill* (planning, debugging discipline, test-writing, self-verification, tool
     ergonomics) that was missing.

"""
```

Figure 19: Cross-lineage Hybridization & Polyglot & ✗✗

Cross-lineage Hybridization & Polyglot & ✗✓:

```
    """You are given **one** programming task attempted by **two different versions** of the
     coding agent. One version **succeeded** and the other **failed**.

By convention in this self-improvement loop:
- **Primary** (commit: `{primary_commit}`, overall accuracy: {primary_utility:.1\%}) is the
    version that **FAILED** on this shared task. Primary is the version to optimize, and
    the new improved child agent will be attached as Primary's descendant.
- **Context** (commit: `{context_commit}`, overall accuracy: {context_utility:.1\%}) is the
    version that **SUCCEEDED**. It is a reference donor only.

**The objective is capability transfer**: extract the *general* skill that allowed Context
    to succeed and design how to instill it into Primary so the new skill helps on many
    unseen tasks.

**CRITICAL  evolve GENERAL capabilities only:**
- Do NOT copy Context's specific tool calls, diff, or chain-of-thought for this task into
    Primary.
- Do NOT hard-code anything language-, framework-, file-, or test-specific from this task.
- DO propose a *general*, language-agnostic skill (planning, decomposition, reproduction,
    incremental testing, self-verification, error-recovery, tool design) such that Primary,
     once equipped, would solve this task **and** hundreds of unrelated tasks across all six
     supported languages.
- Sanity check: if you removed every mention of this specific task from your proposal, would
     it still stand on its own as an implementable improvement? It must.

"""
```

Figure 20: Cross-lineage Hybridization & Polyglot & ✗✓

Cross-lineage Hybridization Json Block:

```
    """
Respond precisely in the following format including the JSON start and end markers:

```json
<JSON>
```

In <JSON>, provide a JSON response with the following fields:
- "primary_approach": Analyze how the **Primary** (the version we are optimizing the one
     that *failed* on this task when one side succeeded) approached the task. Summarize tools
     used, problem-solving strategy, and outcome.
- "context_approach": Analyze how the **Context** (the donor / comparison version the one
     that *succeeded* on this task when one side succeeded) approached the same task. Same
     structure.
- "differential_insight": Compare the two approaches side-by-side. What specific differences
     in behavior, tool usage, or strategy led to the different outcomes? If both failed, which
      got closer and why?
- "transferable_lesson": Extract a **generalizable** lesson from this comparison that applies
      far beyond this specific task. Phrase it as a general capability or strategy that the
     Primary agent should adopt never as "do exactly what Context did on this issue". The
     lesson must be reusable on unseen issues, repositories, and (for polyglot) other
     languages.
- "potential_improvements": Based on the differential analysis, list potential improvements
     for the Primary agent. **Every item must be a general capability** (planning,
     decomposition, reproduction, debugging discipline, test-writing, self-verification, error
     -recovery, tool design, context management, etc.). Reject any candidate that is tied to
     this specific issue/file/function/framework/test/language. If your only ideas feel task-
     specific, climb the abstraction ladder and re-propose them as general skills.
- "improvement_proposal": Choose ONE high-impact *general* improvement. Describe it in detail
      as a unified plan to enhance the Primary agent. Sanity check: if you scrubbed every
     reference to this specific task, would the proposal still be a coherent, implementable
     upgrade? It must.
- "implementation_suggestion": Referring to the coding agent's summary and implementation,
     describe what feature or tool to add or change in Primary so the proposed *general*
     capability is realized. Prefer extending existing tools over inventing redundant ones. Do
      NOT hard-code task-, repo-, framework-, or language-specific strings, paths, regexes, or
      branch logic the change must be useful on tasks the agent has never seen.
- "problem_description": Phrase the improvement proposal and implementation suggestion as a
     GitHub issue description so an engineer can implement it as a *reusable, general-purpose*
      capability that ships with the Primary agent's lineage.

Reminder: the new improved child agent produced from this analysis will be attached as a
     descendant of **Primary**. So the proposal is literally what Primary's offspring will
     inherit make sure it is a *general* upgrade that compounds across many future tasks, not
      a one-off fix for the shared task.

Your response will be automatically parsed, so ensure that the string response is precisely
     in the correct format. Do NOT include the `<JSON>` tag in your output."""
```

Figure 21: Cross-lineage Hybridization Json Block

E.5 Pseudocode of Mendel Gödel Machine

In this section, we present pseudocode of MGM, which helps understand the full training(evolving) 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.

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`

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

    $\tau \leftarrow \text{SelectTask}(a, \mathcal{B}, \mathcal{P}, \beta_{fail})$ `// 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 $\mathcal{A}$

Figure 22: Pseudocode of MGM

F Benchmark Details

F.1 SWE-bench Pro Tasks

To control evaluation cost, instead of evaluating MGM on the full SWEbench-Pro, we utilize ChatGPT randomly choose 60 tasks, which is required to be representative and involve all the six Programming Languages, `JavaScript`, `Python`, `Java`, `C++`, `TypeScript`, `Go`. In this section, we report the 60 tasks used for evaluation on SWE-bench Pro.

- `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-3fa6904377c0064971699 45428e8197158667910
- 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

G Best Discovered-Agents

G.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* → #2 → #9 → #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.

G.2 MGM on SWE-bench Verified

Listing below shows an example self-modification discovered by MGM on SWE-bench Verified. The modification does not hard-code a solution for any particular repository or issue. Instead, it improves the agent's general validation workflow. In the original scaffold, the agent was only instructed to rerun tests after editing and iterate if failures remained. The evolved version makes this process more explicit by introducing a structured before/after validation protocol: the agent must first confirm that the relevant test fails on the current buggy codebase, then apply the fix, and finally verify that the same test passes after the change.

This modification also extends the test-running tool with a dedicated `validate_test_fix` interface that returns structured JSON output about the test file, target function, expected behavior, current status, and recommendations. As a result, the agent receives a clearer signal about whether a test truly captures the intended bug and whether the proposed patch resolves it. This is representative of the type of improvement MGM aims to discover: a reusable debugging and verification habit that can transfer across SWE-bench tasks, rather than a narrow patch tied to a single benchmark instance.

Figure 17: Code modification on Polyglot

```diff
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
```

```diff
+- 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
```

```diff
-        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,
+                )
+
```

```diff
+                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 11041a7..c1d1219 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -212,13 +212,28 @@ Based on the test analysis and baseline failures, you MUST:
 2. Reference the test expectations when making changes
 3. Ensure your changes are minimal and focused

-## Step 4: Validate Changes
-After making code changes, you MUST:
-1. Use the `run_tests` tool again to verify that tests now pass
-2. If tests still fail, analyze the new output
-3. Repeat Steps 3-4 until all relevant tests pass
+## Step 4: Test Validation (Before/After Fix)
+After making code changes, you MUST validate tests using the following workflow:
+
+### Before Fix Validation (Current State)
+1. Use the `validate_test_fix` tool from `run_tests` to validate tests BEFORE the fix
+2. Run the validation with: `test_file`, `test_function`, and `expected_behavior` parameters
+3. Ensure tests FAIL with the current codebase (this confirms they properly detect the issue)
+4. Verify that the test output indicates test failures for the expected behavior
+
+### After Fix Validation (Proposed State)
+1. Use the `validate_test_fix` tool again to validate tests AFTER the fix
+2. Run the validation with the same parameters
+3. Ensure tests PASS with the fixed codebase (this confirms the fix works)
+4. Verify that the test output indicates test success for the expected behavior
+
+### Validation Summary
+- **Before fix**: Tests MUST fail (return code != 0) - this proves they detect the bug
+- **After fix**: Tests MUST pass (return code == 0) - this proves the fix resolves the issue
+- **Structured output**: Always use the structured JSON output from `validate_test_fix` for
    clear validation results

 ## Step 5: Final Verification
+
 Once tests pass, you MUST:
 1. Run tests one final time to confirm stability
 2. Document what was changed and why
@@ -228,11 +243,15 @@ Remember: Do not skip any of these steps. Each step builds on the previous
    one.

 Available tools you MUST use:
 - `test_analysis`: To analyze test files and understand requirements BEFORE coding
-- `run_tests`: To run tests and validate changes AFTER coding
+- `run_tests` (tool_function): To run tests and get status output
+- `run_tests` (validate_test_fix): To validate test fixes with structured before/after analysis
 - `editor`: To view and edit files
 - `bash`: To run commands if needed

+IMPORTANT: For test validation workflows, use `validate_test_fix` with test_file, test_function,
     and expected_behavior parameters to get structured output indicating whether tests pass/
    fail and what changes are needed.
+
 Follow this workflow rigorously to ensure reliable and validated fixes.
+USE validate_test_fix FOR STRUCTURED BEFORE/AFTER FIX VALIDATION BEFORE SUBMITTING CHANGES.
 """
         chat_history, n_llm_calls_used = chat_with_agent(
             instruction,
diff --git a/tools/test_runner.py b/tools/test_runner.py
index 285822a..df82b22 100644
--- a/tools/test_runner.py
+++ b/tools/test_runner.py
@@ -1,6 +1,7 @@
 # This file is adapted from https://github.com/jennyzzt/dgm.

 import sys
+import json
```

```
 from utils.test_runner import run_tests, discover_tests, format_test_output_for_agent


@@ -14,7 +15,10 @@ Use this tool to:
 3. Identify which tests are failing and why

 The tool returns a success status and detailed test output. Use this information to
-understand the current state of the code and validate that your changes fix the issues.""",
+understand the current state of the code and validate that your changes fix the issues.
+
+For validation workflows (before/after fix validation), also use the validate_test_fix
+function which provides structured output indicating whether tests pass/fail and what changes
     are needed.""",
         "input_schema": {
             "type": "object",
             "properties": {
@@ -34,6 +38,263 @@ understand the current state of the code and validate that your changes fix
     the
     }


+def validate_test_fix(test_file, test_function=None, expected_behavior=None, timeout=300):
+    """
+    Validate a test or set of tests for before/after fix validation workflow.
+
+    This function helps ensure tests properly validate fixes by:
+    1. Running the test on the current codebase to establish baseline
+    2. Analyzing whether the test properly captures the expected behavior
+    3. Providing structured output about test status and required changes
+
+    Args:
+        test_file: Path to the test file to validate (required)
+        test_function: Optional specific test function name to validate
+        expected_behavior: Expected behavior or requirements the test should validate
+        timeout: Timeout for the test run in seconds
+
+    Returns:
+        str: JSON-formatted string with structured validation results including:
+            - test_file: Path to the test file
+            - test_function: Name of the test function (if specified)
+            - expected_behavior: Expected behavior description
+            - current_status: Whether test currently passes or fails
+            - analysis: Analysis of the test's ability to validate the fix
+            - recommendations: What changes are needed to the test or code
+    """
+    import os
+    import ast
+    from pathlib import Path
+
+    result = {
+        'test_file': test_file,
+        'test_function': test_function,
+        'expected_behavior': expected_behavior,
+        'current_status': 'unknown',
+        'test_exists': False,
+        'analysis': {},
+        'recommendations': []
+    }
+
+    # Check if test file exists
+    if not os.path.exists(test_file):
+        result['error'] = f"Test file not found: {test_file}"
+        result['recommendations'].append(f"Create test file at: {test_file}")
+        return json.dumps(result, indent=2)
+
+    result['test_exists'] = True
+
+    # Try to run the test
+    test_files_to_run = [test_file]
+    if test_function:
```

```
+        # For specific test function, we'll need to run with pytest -k
+        # For now, just run the file and we'll analyze the results
+        pass
+
+    try:
+        # Run the tests
+        success, output = run_tests(test_files=test_files_to_run, timeout=timeout)
+
+        result['current_status'] = 'pass' if success else 'fail'
+        result['test_output'] = output
+
+        # Analyze the test file to understand what it's testing
+        test_analysis = analyze_test_for_validation(test_file, test_function, expected_behavior)
+        result['analysis'] = test_analysis
+
+        # Generate recommendations based on analysis
+        recommendations = generate_validation_recommendations(
+            test_file, test_function, expected_behavior, test_analysis, success
+        )
+        result['recommendations'] = recommendations
+
+    except Exception as e:
+        result['error'] = str(e)
+        result['recommendations'].append(f"Error during test validation: {str(e)}")
+
+    return json.dumps(result, indent=2)
+
+
+def analyze_test_for_validation(test_file, test_function=None, expected_behavior=None):
+    """
+    Analyze a test file to understand what it's testing and how well it validates behavior.
+
+    Args:
+        test_file: Path to the test file
+        test_function: Optional specific test function name
+        expected_behavior: Expected behavior description
+
+    Returns:
+        dict: Analysis of the test's validation capabilities
+    """
+    analysis = {
+        'has_assertions': False,
+        'assertion_count': 0,
+        'captures_expected_behavior': False,
+        'coverage_gaps': [],
+        'strengths': [],
+        'weaknesses': []
+    }
+
+    try:
+        with open(test_file, 'r') as f:
+            content = f.read()
+
+        tree = ast.parse(content)
+
+        # Find the specific test function if requested
+        test_nodes = []
+        for node in ast.walk(tree):
+            if isinstance(node, ast.FunctionDef):
+                if test_function and node.name == test_function:
+                    test_nodes.append(node)
+                    break
+                elif node.name.startswith('test_') and not test_function:
+                    test_nodes.append(node)
+
+        if not test_nodes:
+            analysis['warning'] = "No test functions found matching criteria"
+            return analysis
+
+        # Analyze each test function
```

```
+        for test_node in test_nodes:
+            # Count assertions
+            assertion_count = 0
+            for child in ast.walk(test_node):
+                if isinstance(child, (ast.Assert, ast.Call)):
+                    if isinstance(child, ast.Call):
+                        if isinstance(child.func, ast.Attribute):
+                            if 'assert' in child.func.attr.lower():
+                                assertion_count += 1
+                    elif isinstance(child, ast.Assert):
+                        assertion_count += 1
+
+            analysis['assertion_count'] += assertion_count
+
+            if assertion_count > 0:
+                analysis['has_assertions'] = True
+
+            # Check if test has clear expected behavior
+            docstring = ast.get_docstring(test_node)
+            if docstring:
+                if expected_behavior and expected_behavior.lower() in docstring.lower():
+                    analysis['captures_expected_behavior'] = True
+                    analysis['strengths'].append("Test docstring captures expected behavior")
+                else:
+                    analysis['coverage_gaps'].append("Test docstring doesn't match expected behavior")
+
+            # Check for validation patterns
+            test_source = ast.unparse(test_node) if hasattr(ast, 'unparse') else content
+            if 'assert' in test_source:
+                analysis['strengths'].append("Test includes assertions")
+            if 'assertEqual' in test_source or 'assertEqual' in test_source:
+                analysis['strengths'].append("Test uses specific assertion methods")
+            if 'expected' in test_source.lower() or 'expected' in test_source.lower():
+                analysis['strengths'].append("Test references expected values")
+
+            # Check for weaknesses
+            if assertion_count == 0:
+                analysis['weaknesses'].append("No assertions found in test")
+            if not docstring:
+                analysis['weaknesses'].append("Test lacks docstring explaining expected behavior")
+
+            # Check if test has setup/teardown
+            if 'setup' in test_source.lower() or 'teardown' in test_source.lower():
+                analysis['strengths'].append("Test has setup/teardown")
+            else:
+                analysis['coverage_gaps'].append("Test lacks setup/teardown methods")
+
+        # General analysis
+        if analysis['assertion_count'] == 0:
+            analysis['coverage_gaps'].append("Test has no assertions - cannot validate behavior")
+
+        if expected_behavior and not analysis['captures_expected_behavior']:
+            analysis['coverage_gaps'].append(f"Expected behavior '{expected_behavior}' not explicitly validated")
+
+    except Exception as e:
+        analysis['error'] = str(e)
+
+    return analysis
+
+
+def generate_validation_recommendations(test_file, test_function, expected_behavior, analysis, success):
+    """
+    Generate recommendations for test validation based on analysis.
+
+    Args:
```

```
+        test_file: Path to test file
+        test_function: Test function name
+        expected_behavior: Expected behavior
+        analysis: Analysis results from analyze_test_for_validation
+        success: Whether test currently passes
+
+    Returns:
+        list: List of recommendations
+    """
+    recommendations = []
+
+    # Check for critical issues
+    if not analysis.get('has_assertions', False):
+        recommendations.append({
+            'priority': 'high',
+            'issue': 'No assertions found',
+            'action': 'Add at least one assertion to validate behavior'
+        })
+
+    if analysis.get('assertion_count', 0) == 0:
+        recommendations.append({
+            'priority': 'high',
+            'issue': 'Test has no assertions',
+            'action': 'Add assertions to validate the expected behavior'
+        })
+
+    if expected_behavior and not analysis.get('captures_expected_behavior', False):
+        recommendations.append({
+            'priority': 'medium',
+            'issue': 'Expected behavior not validated',
+            'action': f'Ensure test validates: {expected_behavior}'
+        })
+
+    # Check current status
+    if success:
+        if analysis.get('weaknesses'):
+            recommendations.append({
+                'priority': 'low',
+                'issue': 'Test passes but has weaknesses',
+                'action': 'Review and address the following weaknesses: ' + ', '.join(analysis['
    weaknesses'])
+            })
+        else:
+            recommendations.append({
+                'priority': 'info',
+                'issue': 'Test validates correctly',
+                'action': 'Test is properly validating the expected behavior'
+            })
+    else:
+        # Test is failing - check if it should fail
+        if analysis.get('coverage_gaps'):
+            recommendations.append({
+                'priority': 'medium',
+                'issue': 'Test fails due to gaps',
+                'action': 'Address the following gaps: ' + ', '.join(analysis['coverage_gaps'])
+            })
+        else:
+            recommendations.append({
+                'priority': 'info',
+                'issue': 'Test fails as expected',
+                'action': 'Fix the implementation to make test pass'
+            })
+
+    # Specific recommendations based on test function
+    if test_function:
+        recommendations.append({
+            'priority': 'info',
+            'issue': 'Focused test',
+            'action': f'Test function "{test_function}" should reproduce the exact failure from
    the issue'
```

```
+        })
+
+    return recommendations
+
+
diff --git a/coding_agent.py b/coding_agent.py
index 11041a7..19fc8c6 100644
--- a/coding_agent.py
+++ b/coding_agent.py
@@ -75,6 +75,59 @@ def safe_log(message, level=logging.INFO):
         print(f"Warning: No logger found for thread {threading.get_ident()}")


+def _is_non_submission_path(path):
+    """
+    Return True for files that should not count as final SWE-bench submissions.
+
+    The report pipeline strips test patches before evaluation. Keep this local
+    check in the agent too so a test-only diff is caught before submission.
+    """
+    normalized = path.replace("\\", "/")
+    if normalized.startswith("a/") or normalized.startswith("b/"):
+        normalized = normalized[2:]
+    return (
+        normalized == "tox.ini"
+        or normalized.endswith("/tox.ini")
+        or normalized.startswith("test/")
+        or normalized.startswith("tests/")
+        or normalized.startswith("testing/")
+        or "/test/" in normalized
+        or "/tests/" in normalized
+        or "/testing/" in normalized
+        or normalized.startswith("test_")
+        or "/test_" in normalized
+        or normalized.endswith(".md")
+        or normalized.endswith(".bak")
+        or normalized.endswith(".backup")
+    )
+
+
+def filter_submission_patch(model_patch):
+    """
+    Remove test and auxiliary file hunks from a patch.
+
+    This mirrors the evaluation-side test filtering and adds a small auxiliary
+    file guard for markdown summaries and backup files that agents often create
+    while debugging.
+    """
+    filtered_lines = []
+    skip_file = False
+    for line in model_patch.splitlines(keepends=True):
+        if line.startswith("diff --git a/"):
+            pieces = line.split()
+            target = pieces[-1] if pieces else ""
+            skip_file = target.startswith("b/") and _is_non_submission_path(target)
+        if not skip_file:
+            filtered_lines.append(line)
+    return "".join(filtered_lines)
+
+
+def truncate_patch_for_prompt(model_patch, limit=12000):
+    if len(model_patch) <= limit:
+        return model_patch
+    return model_patch[:limit] + "\n\n[patch truncated]\n"
+
+
 class AgenticSystem:
     def __init__(
         self,
@@ -190,6 +243,12 @@ Your task is to run the regression tests in the {self.git_tempdir} directory
```

```
     to

 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.

+Final patch requirements:
+- The final submitted patch MUST include at least one non-test production source-code change.
+- You may create temporary reproduction tests or scripts while debugging, but they must not be
     the only changes in the final diff.
+- Do not submit changes that only touch tests, test fixtures, markdown summaries, backup files,
     demo scripts, or tox.ini.
+- Before finishing, inspect the final diff and make sure it changes the implementation that
     fixes the issue.
+
 # IMPORTANT: Test-Driven Development Workflow

 You MUST follow a structured test-driven development workflow to ensure your changes are
     validated:
@@ -302,8 +361,42 @@ def main():
     # Run the agentic system to try to solve the problem
     agentic_system.forward(args.timeout)

-    # Get code diff and save to model_patch.diff
-    model_patch = diff_versus_commit(args.git_dir, args.base_commit)
+    # Get code diff and save to model_patch.diff. The evaluator strips tests
+    # before scoring, so retry once if the agent only produced test/auxiliary
+    # changes that would become an empty submitted patch.
+    raw_model_patch = diff_versus_commit(args.git_dir, args.base_commit)
+    model_patch = filter_submission_patch(raw_model_patch)
+    if not model_patch.strip():
+        reason = (
+            "Your current diff only changes tests or auxiliary files, which are "
+            "removed before SWE-bench scoring."
+            if raw_model_patch.strip()
+            else "Your current diff is empty."
+        )
+        corrective_instruction = f"""I have uploaded a Python code repository in the directory {
     args.git_dir}.
+
+The previous attempt cannot be submitted: {reason}
+
+You must now fix the production implementation for the original problem. The final submitted
     patch must include at least one non-test source-code change. Temporary tests or reproduction
      scripts are allowed while reasoning, but they must not be the only final changes.
+
+<problem_description>
+{args.problem_statement}
+</problem_description>
+
+<current_diff>
+{truncate_patch_for_prompt(raw_model_patch)}
+</current_diff>
+"""
+        chat_with_agent(
+            corrective_instruction,
+            model=agentic_system.code_model,
+            msg_history=[],
+            logging=safe_log,
+            timeout=max(60, min(600, args.timeout // 3)),
+        )
+        raw_model_patch = diff_versus_commit(args.git_dir, args.base_commit)
+        model_patch = filter_submission_patch(raw_model_patch)
+
     model_patch_outfile = (
         os.path.join(args.outdir, "model_patch.diff")
         if args.outdir
@@ -312,6 +405,15 @@ def main():
     with open(model_patch_outfile, "w") as f:
        f.write(model_patch)
```

```
+    if raw_model_patch != model_patch:
+        raw_patch_outfile = (
+            os.path.join(args.outdir, "model_patch.raw.diff")
+            if args.outdir
+            else "model_patch.raw.diff"
+        )
+        with open(raw_patch_outfile, "w") as f:
+            f.write(raw_model_patch)
+

 if __name__ == "__main__":
     main()
```