HUXLEY-GÖDEL MACHINE: HUMAN-LEVEL CODING AGENT DEVELOPMENT BY AN APPROXIMATION OF THE OPTIMAL SELF-IMPROVING MACHINE

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent studies operationalize self-improvement through coding agents that edit their own codebases, grow a tree of self-modifications through expansion strategies that favor higher software engineering benchmark performance, considering that this implies more promising subsequent self-modifications. However, we identify a mismatch between the agent's self-improvement potential (metaproductivity) and its coding benchmark performance, namely the *Metaproductivity-Performance Mismatch*. Inspired by Huxley's concept of clade, we propose a metric (CMP) that aggregates the benchmark performances of the *descendants* of an agent as an indicator of its potential for self-improvement. We show that the Gödel Machine, the optimal self-improving machine, is achieved with access to true CMP. We introduce the Huxley-Gödel Machine (HGM), which, by estimating CMP and using it as guidance, searches the tree of self-modifications. On SWE-bench Verified and Polyglot, HGM outperforms prior self-improving coding agent search methods while using less wall-clock time. Moreover, the agent optimized by HGM on SWE-bench Verified outperforms SWE-agent, a leading human-engineered open source coding agent on SWE-bench Lite, where SWE-agent ranks the best on the official leaderboard, when both use the GPT-5-mini backbone, demonstrating that HGM self-improvement indeed enhances genuine coding capability.

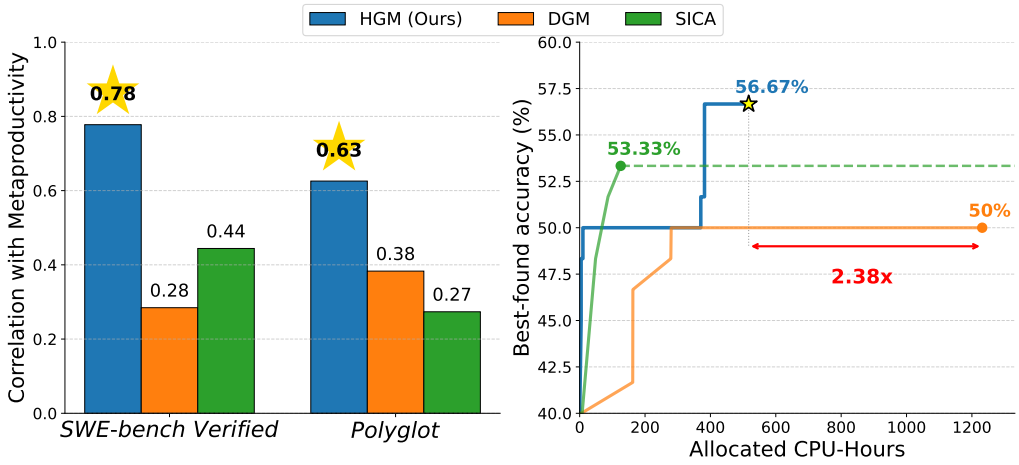

Figure 1: **(Left)** Weak correlation between the guidance metrics of other methods (based on benchmark performance) and long-term self improvement; HGM mitigates this mismatch by leveraging clade-level metaproductivity. **(Right)** On *SWE-bench Verified*, HGM achieves **higher** accuracy with 2.38 time less allocated CPU-hours. Together, the results indicate the practical advantage of approximating Gödel Machines with long-term self-improvement estimates. SICA encountered repeated errors after consuming 45% of its budget, preventing any further self-modifications.

1 INTRODUCTION

Self-improvement, the ability of agents to iteratively revise their own code, underlies the growth of complex systems, from biological evolution to human culture and scientific progress (Good, 1966; Schmidhuber, 1987). Bringing this principle into artificial intelligence, self-improving agents (Hutter, 2005; Schmidhuber, 2003; 2006; Hall, 2007) mark a pivotal step toward open-ended AI. Unlike static systems constrained by fixed architectures, such agents can recursively modify their own mechanisms and strategies, reusing newly gained abilities to fuel subsequent improvements. This recursive capacity fosters continual adaptation, reduces reliance on human intervention, and enables problem-solving capabilities that cannot be fully anticipated at design time.

A central challenge, however, is how to decide which self-modifications to accept. The Gödel machine (Schmidhuber, 2003) offers a theoretically optimal answer: accept only modifications that provably increase the expected long-term utility. While this provides a sound blueprint, its reliance on formal proofs makes it impractical. Current implementations instead rely on coding agents that edit their own codebases and favor self-modifications from agents with higher benchmark performance (Robeyns et al., 2025; Zhang et al., 2025a). Yet, as illustrated in Figure 1 (left), this heuristic can be misleading: a high-scoring agent may produce unproductive descendants, while a lower-scoring one seeds lineages that achieve greater long-term gains. We term this phenomenon the *Metaproductivity–Performance Mismatch*.

To address this mismatch, we introduce *clade-level metaproductivity (CMP)*, inspired by Huxley's notion of clades as lineages of common ancestry (Huxley, 1957). CMP quantifies the productivity of a lineage by aggregating the success of an agent's descendants rather than relying only on its immediate benchmark score. Furthermore, we show in Theorem 1 that *access to the true CMP is sufficient to reproduce the Gödel Machine's acceptance rule*. This insight motivates our proposed algorithm, the **Huxley–Gödel Machine (HGM)**, which approximates Gödel-style self-improvement by estimating CMP from clade-aggregated descendant outcomes, and selecting nodes to expand via Thompson sampling. Furthermore, due to a more reliable estimate, we adaptively decouple expansion from evaluation, leading to asynchronous execution for efficient parallelism.

Empirically, HGM better aligns with long-run agent productivity than benchmark-driven baselines, as shown in Figure 1 (left). On SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) and Polyglot (Gauthier, 2024), HGM consistently outperforms Darwin Gödel Machine (DGM) (Zhang et al., 2025a) and Self-Improving Coding Agent (SICA) (Robeyns et al., 2025). Remarkably, one agent found by HGM surpasses SWE-agent (Yang et al., 2024), the highest-scored open-source human-engineered coding agent with officially checked results, on SWE-bench Lite (Jimenez et al., 2024), when both use the GPT-5-mini backbone under matched budgets.

To summarize, our contributions are as follows:

- We analytically define the Clade-Metaproductivity (CMP) function as a measure of agents' self-improving ability and show that access to a CMP oracle suffices to reproduce the Gödel Machine's acceptance mechanism (Theorem 1).
- We empirically observe that immediate benchmark performance is an unreliable predictor of CMP and show that our CMP estimator better predicts it.
- Using our CMP estimator, we propose the **Huxley–Gödel Machine (HGM)**, which operationalizes Gödel's approximately from partial evaluations and guides the expansion via Thompson sampling with adaptive scheduling.
- We empirically validate HGM on SWE-bench Verified and Polyglot, demonstrating higher-quality optimized agents compared to previous self-improving methods, even though they were discovered within substantially smaller allocated CPU-hours. Furthermore, HGM **achieves human-level coding agent design** on SWE-bench Lite by optimizing on SWE-bench Verified.

2 SELF-IMPROVING AS TREE-SEARCH

Both the Darwin Gödel Machine (DGM) and the Self-Improving Coding Agent (SICA) belong to the class of self-referential AI (Schmidhuber, 1987; 2006; 2007), wherein a single agent modifies itself to generate new agents, each empirically validated on downstream tasks. In this paper, we formalize this self-improvement process as an iterative tree-search problem, where the goal is to discover an

agent that maximizes performance across multiple downstream tasks. Concretely, starting from an initial agent as the root, a tree-search policy incrementally grows the tree of self-modified agents. At each iteration, the policy either selects an agent (a node in the tree) to expand by producing a child agent (a self-modified version of the selected agent) or selects an agent to undergo additional evaluation on downstream tasks.

Formally, let $\mathcal{T}_t$ denote the archive of our agents at iteration t. In this paper, the archive is always represented as a tree of evolved agents, and we use the terms archives and trees interchangeably. $T_0 = \{a_0\}$ is initialized as a single-node tree with a fixed initial agent. At iteration t, the policy selects actions $a_{t+1} \sim \pi(\cdot \mid \mathcal{T}_t)$, where π is a policy over actions $\mathcal{A}_t = \mathcal{M}_t \cup V_t$, where $\mathcal{M}_t = \{m_a : a \in \mathcal{T}_t\}$ are agent modifications and $V_t = \{v_a : a \in \mathcal{T}_t\}$ are evaluations. Here, m_a instructs agent a to produce a self-modification that is added as a child to the tree, while v_a selects an agent from the tree for an additional evaluation on one more downstream task. After exhausting the computational budget, the policy selects a final agent ($a_{\text{final}} = \arg\max_{a \in \mathcal{T}} Score(a) \in \mathcal{T}_T$ where T is the termination iteration and $Score$ is part of the policy) from the final tree as the returned agent. The objective is to optimize $J(\pi) = \mathbb{E}[U(a_{\text{final}})]$, where U is a utility function that measures downstream tasks performance. In this work, we define U as the average of binary success indicators across all downstream tasks. π denotes an algorithm, with DGM, SICA, and our proposed HGM representing concrete instances.

Compound Policy. At each step of self-improvement, the system faces a compound decision: whether to expand the tree by generating new agents or to evaluate existing ones. This decision naturally decomposes into three sub-policies: (i) a *selection policy* that chooses between expansion and evaluation, (ii) an *expansion policy* that determines which parent to modify, and (iii) an *evaluation policy* that selects which agent to test. Prior approaches, such as SICA and DGM, conflate these choices. They always expand a parent, create a child, and immediately evaluate that child on multiple tasks. This fixed sequence restricts flexibility: once a new agent is generated, it monopolizes evaluations, even if older agents appear more promising. For instance, an agent that fails nine tasks in a row continues to consume evaluations, while an older agent with partial successes is ignored.

HGM breaks this rigidity by *decoupling* expansion from evaluation. At each step, it adaptively decides whether to generate a new agent or to further probe an existing one, and evaluations are always at the granularity of a *single agent–task pair*. This finer control enables early stopping on unpromising agents. Table 4 summarizes how SICA, DGM, and HGM instantiate these sub-policies.

3 Huxley-Gödel Machine

In this section, we introduce the Huxley–Gödel Machine (HGM), a practical search policy for self-improving coding agents that elevates lineage-level evidence over short-term node performance. At the core of HGM lies the notion of clade-level metaproductivity (CMP), which is sufficient to reproduce the Gödel Machine when its true value is accessible(Theorem. 1). HGM, by estimating CMP from partially observed evaluations, approximates Gödel Machine, guiding expansion through Thompson sampling and adaptively balancing modification and evaluation. Our formulation naturally supports asynchronous execution, enabling efficient parallel utilization of compute resources.

3.1 Metaproductivity and Clade-Metaproductivity

Given a policy π, to quantify the quality of how an agent's self-modification influences the performance of the system, we define the notion of global metaproductivity (GMP):

$$\text{GMP}_\pi(\mathcal{T}, a) \;=\; \mathbb{E}_{\mathcal{T}_B \sim p_\pi(\cdot \mid \mathcal{T}, a)} \left[U(\text{argmax}_{a' \in \mathcal{T}_B} Score_\pi(a')) \right],$$

where $\mathcal{T}$ is a tree of agents and $a \in \mathcal{T}$. $Score_\pi$ is the function that scores the agents for the final selection. The policy π unrolls the trajectory until the end of the episode with policy π and produces a final archive of agents $\mathcal{T}_B$. The distribution of the trajectory is given by p_π .

GMP directly corresponds to the Q-value function in reinforcement learning, with state phrased as the archive of agents, and action being the selected agent to expand. The GMP value of a node measures how good (on average) the final agent obtained from the search process will perform. GMP measures the long-term potential of self-improvements, which also includes modifications that improve self-improvement itself and so on. An algorithm might, at the beginning, focus on improving

the ability to self-improve while neglecting direct benchmark abilities, only to later focus on them. This is a principal meta-learning behavior that is captured in the original Gödel Machine (Schmidhuber, 2003). The objective of designing a policy for self-improvement (Section 2) is equivalent to optimizing $\mathrm{GMP}(\{a_0\}, a_0)$.

While GMP captures the full long-term potential of a policy, its scope is overly broad for practical conceptualization. Notably, the Gödel Machine considers only the outcomes of the current agent and its descendants when deciding whether to accept a modification. Motivated by this observation, we define a localized variant of GMP that focuses on the subtree rooted at a given agent, i.e., its *clade*. We refer to this quantity as Clade-Metaproductivity (CMP):

$$\begin{aligned}\mathrm{CMP}_\pi(\mathcal{T}, a) &= \mathbb{E}_{\mathcal{T}_B \sim p_\pi(\cdot|\mathcal{T},a)}\left[U(\text{argmax}_{a' \in \mathcal{C}(\mathcal{T}_\mathcal{B},a)} Score(a'))\right] \\ &= \mathbb{E}_{\mathcal{T}_B \sim p_\pi(\cdot|\mathcal{T},a)}\left[\text{max}_{a' \in \mathcal{C}(\mathcal{T}_\mathcal{B},a)} U(a')\right] \qquad \text{(if Sel = U)},\end{aligned}$$

where $\mathcal{C}(\mathcal{T}_\mathcal{B}, a)$ is the clade (i.e., the subtree with a as the root) of the node a in the Tree $\mathcal{T}_\mathcal{B}$ and $Score$ is the final agent selection metric.

CMP contains the non-greedy information about the future evolution of self-improving agents, therefore guiding good strategies for self-improvement aimed also at the improvement of the self-improvement itself. Furthermore, we show the crucial relation of CMP to the Gödel Machine.

Theorem 1. *Access to a* CMP *oracle is sufficient to implement a Gödel Machine.*

The proof is available in the App. A. This observation motivates us to introduce the estimate of CMP and use this as guidance in our algorithm. An algorithm with a perfect estimate of CMP would be able to produce the Gödel Machine. HGM by estimating CMP approximates the original Gödel Machine. We describe our algorithm fully in the next section.

3.2 Algorithm

Existing methods use benchmark performance on coding tasks as a guidance metric, treating task success as an indicator of self-improvement potential. This assumption is overly greedy: it evaluates only the immediate utility of a modification while ignoring its downstream consequences for future self-modifications. We refer to this gap as the *Metaproductivity-Performance Mismatch*: the divergence between short-term task performance and the long-term capacity for self-improvement as measured by CMP. Empirical evidence shows that this mismatch happens in practice (see Section 4.1.) We aim to model long-term, global dependencies by deriving our estimator of CMP. Specifically, we define HGM by stating its three subpolicies.

Expansion Policy. The core of the HGM algorithm is its selection criterion for expansion. HGM aims to estimate Clade-Metaproductivity with the motivation that the true CMP as the criterion would produce the Gödel-Machine due to Theorem 1. In this sense, HGM approximates Gödel-Machine, the **optimal** self-improving machine. This is in contrast to the currently used greedy selection criteria based on performance metrics, which ignore the potential of the model to improve self-improving abilities.

We estimate CMP with the **weighted** average of agents' empirical performance in the clade. (See below for how our evaluation policy promotes more accurate estimation of CMP.) Formally, let us assume a fixed archive of agents $\mathcal{T}_t$, $n_{\text{success}}(a)$ be the number of passed tests of a, and $n_{\text{failure}}(a)$ be the number of failed tests of a. Then

$$n^C_{\text{success}}(a) = \sum_{a' \in C(a)} n_{\text{success}}(a') \quad \text{and} \quad n^C_{\text{failure}}(a) = \sum_{a' \in C(a)} n_{\text{failure}}(a').$$

Where $C(a)$ is the clade of a in $\mathcal{T}_t$. We define our Clade-Metaproductivity estimator as

$$\widehat{\mathrm{CMP}}(a) = \frac{n^C_{\text{success}}(a)}{n^C_{\text{success}}(a) + n^C_{\text{failure}}(a)},$$

Evaluating productivity at the level of entire clades rather than individual agents offers several key advantages. It aligns better with the goal of self-improvement, as a modest ancestor can still be highly valuable if its descendants consistently advance, while stagnant lineages are deprioritized. At the same time, aggregating evidence across a clade yields more statistically robust estimates than single-node outcomes by using information from more samples. This is particularly important when evaluations are costly and benchmarks are only partially observed.

$\widehat{\mathrm{CMP}}(a)$ can be viewed as a weighted sum over empirical means of agents in $C(a)$, with the weight for an agent being the number of task evaluations it has. Furthermore, we design our evaluation selection in such a way that it selects **highly performing agents**, which creates a selection of a soft maximum in the clade. We discuss how $\widehat{\mathrm{CMP}}$ estimates CMP in detail in App. D.

After calculating the CMP estimates, the HGM probabilistically approximates the selection of the highest scoring agent with Thompson Sampling - a standard method in the bandit literature for smoothly maximizing the decision criterion (Agrawal & Goyal, 2012; Chapelle & Li, 2011; Lattimore et al., 2020). We will refer to $a \sim TS(\{n_s, n_f | n \in \mathcal{T}_t\})$ as the agent sampled from the Thompson-Sampling process with parameters n_s (number of successes) and n_f (number of failures). Given the fact that the search problem has a known budget, Our algorithm introduces an exploration-exploitation scheduler τ which is monotonically increasing with respect to the current time t, encouraging exploration in the early stage and polarization of the sampling distribution when approaching the end. Formally, we select the agent to expand a^* as

$$a^* \sim TS(\{\tau(1 + n^C_{\text{success}}(a)), \tau(1 + n^C_{\text{failure}}(a)) | a \in \mathcal{T}_t\}).$$

Evaluation Policy As stated in the expansion policy, we design our evaluation policy to prioritize **agents** with a higher evaluation score to induce the selection of the maximum over the clade. Formally, the agent to evaluate a^* is sampled from the Thompson Sampling process with

$$a^* \sim TS(\tau(1 + n_{\text{success}}(a)), \tau(1 + n_{\text{failure}}(a)).$$

Selection Policy. Finally, our agent has to choose between expansion and evaluation. At each iteration, the algorithm first selects whether to evaluate or expand. Previous methods have evaluated newly created agents directly after their creation. Our novel estimation of agent self-improving quality has an additional benefit of collecting more samples faster (because it has samples from the entire clade). This enables a more fine-grained control over when to evaluate and when to create a new agent for better efficacy. Therefore, we decouple evaluation from expansion and treat them as separate steps.

To decide how and when to evaluate or expand agents, we draw inspiration from the infinite-armed bandit literature. Infinite-armed bandit problems capture the tension between repeatedly sampling known options to reduce uncertainty about promising arms and exploring new options that have the potential to perform better. This perspective provides a natural lens for our setting, where evaluations correspond to sampling existing arms and expansions correspond to introducing new ones. In this work, we follow the strategy of UCB-Air (Wang et al., 2008), which adds arms when the number of evaluations $N^\alpha \geq m$ for some $\alpha \in [0, 1]$, where m is the number of existing arms. In our case, arms correspond to the agents; hence, we decide to expand at time t if $N^\alpha_t \geq |\mathcal{T}_t|$.

Final Agent Selection Strategy HGM iteratively executes the structured policy defined by our selection policy, expansion policy, and evaluation policy. When the computational budget exceeds, it returns the agent with the highest ϵ percentile of the utility posterior in the final tree for some hyperparameter ϵ, namely the **best-belief agent**. Formally, a best-belief agent is defined as

$$\text{argmax}_{a \in \mathcal{T}_B} I_\epsilon(1 + n_{\text{success}(a)}, 1 + n_{\text{failure}}(a)),$$

where I is the regularized incomplete beta function. See Algorithm1 in Appendix B for the detailed procedure of HGM.

Asynchronous Implementation As an additional benefit of decoupling the policy, we introduce asynchronous execution of evaluation and expansion. Since the execution of coding agents generally requires querying large language models multiple times, the computation time can be lengthy. To boost our algorithm, we propose the asynchronous HGM algorithm (HGM Async), which utilizes

Table 1: **Clade-Metaproductivity: Empirical vs. Estimation Correlation**. We report the Pearson correlations between the empirical $\mathrm{CMP}s$ and the estimates from DGM, SICA, and HGM on SWE-Verified-60 and Polyglot. For the weighted correlations, each prediction is weighted by its accessed number of evaluations.

Estimates	SWE-Verified-60		Polyglot	
	Weighted	Un-weighted	Weighted	Un-weighted
SICA	0.444	0.444	0.274	0.274
DGM	0.285	0.406	0.383	0.357
HGM (Ours)	**0.778**	**0.512**	**0.626**	**0.873**

all possible computational power until the computational budget is exceeded. HGM Async synonymously executes one iteration process on each available CPU. Once one iteration finishes, a new iteration immediately starts. It uses the most recent data with one exception and updates the data once it finishes. The exception is that one needs to take all running expansions and explorations into consideration when executing the selection strategy. See experimental results 2 for run time comparison with DGM and SICA.

4 Experimental Results

We evaluate HGM on challenging software engineering tasks to assess three core aspects: **1)** the fidelity of HGM's CMP estimation (Sec. 4.1), **2)** its capability for self-improvement with HGM compared with DGM and SICA (Sec. 4.2), and **3)** the effectiveness in automatic agent design through evolutionary processes, benchmarked against a leading human design up to date[1] (Sec. 4.3). We conducted our experiments on the SWE-bench Verified (SWE-Verified) and SWE-bench Lite (SWE-Lite) variants, and the Polyglot problems, both consisting of coding challenges and are widely used for coding agent evaluation (Xia et al., 2025; Zhang et al., 2024; 2025b). For budget considerations, in addition to the full datasets, we use 60-task subsets (SWE-Verified-60), derived from the first two stages of DGM's progressive evaluation. In all experiments, we employ HGM with an exploration-exploitation scheduler $\frac{B}{b}$, where b is the remaining budget, $\epsilon = 1$, and $\alpha = 0.6$. All experiments involving HGM use the HGM-Async algorithm. We apply an identical initial agent when compared to DGM and SICA, which is adopted from the official implementation of DGM. See Appendix C.1 for a detailed description of the initial agents used in different experiments.

4.1 Metaproductivity-Performance Mismatch

The experiments in this section are designed to serve two purposes: (i) to provide evidence of the Metaproductivity-Performance Misalignment (MPM) issue; and (ii) to assess whether the $\widehat{\mathrm{CMP}}$ of HGM is a more reliable CMP estimator than the utility measures adopted by DGM and SICA. To reveal the misalignment inherent in such reliance, we compute the correlation between their predictions and empirical CMP. To obtain empirical $\mathrm{CMP}s$, we analyze the expanded search tree after each method has completed its run. For every node in the tree, we define its empirical CMP as the maximum empirical mean of the task performance achieved within its clade with the root of this clade excluded. This construction ensures that empirical CMP captures the self-improvement ability of a node. We exclude the root of a clade to avoid circular use of the target in the estimators. For HGM, the $\widehat{\mathrm{CMP}}$ is defined as a function over the clade of a node; a critical adjustment is required to avoid target leakage. Specifically, we exclude the evaluations that are most directly related to the target: the root of the clade (an ancestor of the target) and the subtree rooted at a direct child of the evaluated node that contains the empirical maximum, thereby ensuring a fair comparison (see Appendix D for detailed computation). We report both the correlation coefficient weighted by the number of evaluations used in prediction, as well as the unweighted correlation. We conducted our experiments on the SWE-Veified-60 and Polyglot datasets.

[1]The leading SWE-agents on `https://www.swebench.com` (Lite) as of 22 September 2025.

Table 2: **Self-Improving Capability Comparison**. We report the task performance (in accuracy) of each method's best-belief agent and the allocated CPU-hours time required for 800 evaluations. Super-scripted accuracies with "+" indicate performance gains over their respective initial agents.

Best-belief Agent of	SWE-Verified-60		Polyglot	
	Acc. (%)↑	Time (hours)↓	Acc. (%)↑	Time (hours)↓
SICA	50.0^{+10}	infinite loop	$25.4^{+5.1}$	572
DGM	$53.3^{+13.3}$	1231	$27.1^{+6.8}$	2385
HGM (Ours)	$\mathbf{56.7}^{+16.7}$	**517**	$\mathbf{30.5}^{+10.2}$	**347**

Results &. Discussion Table 1 summarizes the correlations between the three estimators and the empirical $\mathrm{CMP}s$ (the targets). We first observe that the SICA and DGM estimators achieve positive Pearson correlation coefficients: 0.444 and 0.285 on SWE-Verified-60, and 0.274 and 0.383 on Polyglot, respectively, suggesting weak alignments, i.e., MPM. In comparison, HGM's estimator, $\widehat{\mathrm{CMP}}$, achieves substantially stronger weighted correlations of 0.778 and 0.626 on SWE-Verified-60 and Polyglot, respectively, as well as 0.512 and 0.8783 unweighted correlation, surpassing SICA and DGM by wide margins. These results provide strong indications that HGM, equipped with $\widehat{\mathrm{CMP}}$, offers a more reliable estimate of metaproductivity and effectively mitigates the MPM issues inherent to SICA and DGM.

4.2 Evaluating HGM Self-Improving Capability

To validate our hypothesis that our CMP estimator better predicts future self-improvement and hence leads to more effective self-modifications, we evaluate HGM against two state-of-the-art self-improving coding agent methods: DGM and SICA. We conduct controlled experiments under the same setup as DGM, with a budget-friendly modification applied consistently to both HGM and all baselines to ensure fairness, i.e., we adopt more cost-efficient backbone LLMs (GPT-5 for expansion and GPT-5-mini for evaluation for SWE-Verified; Qwen3-Coder-480B-A35B-Instruct for expansion and Qwen3-Coder-30B-A3B-Instruct for evaluation for Polyglot). For all methods, we start with the same initial ancestor, which achieves 40% and 20.3% on the SWE-Verified-60 and Polyglot, respectively. We compare the task performance of their best-belief final agents after a maximum allowance of 800 benchmark task evaluations, selected in DGM and SICA using empirical means. In addition, our asynchronous parallelization of expansion and evaluation should enable the self-improvement to consume fewer allocated CPU-hours than DGM and SICA (see Sec. 3.2). To verify this, we also report the allocated CPU-hours required for 800 evaluations.

Results &. Discussion We summarize the comparison results in Table 2. Across both SWE-Verified-60 and Polyglot, all three methods successfully perform agent discovery by optimizing the initial agent through self-improvement. However, HGM's best-belief agent demonstrates not only the highest task performance—56.7% on SWE-Verified-60 and 30.5% on Polyglot—but also the best efficiency, requiring the fewest allocated CPU-hours for 800 evaluations: 6.86× faster than DGM and 1.65× faster than SICA on Polyglot, and 2.38× faster than DGM on SWE-Verified-60. Notably, on SWE-Verified-60, SICA repeatedly encounters "query length out-of-LLM-context-window" during self-improvement processes after 360 evaluations. Despite this, the Polyglot results validate our hypothesis on HGM's runtime advantage over the baselines. In conclusion, HGM, equipped with a better utility estimator and asynchronous expansion–evaluation iterations, establishes itself as a more effective self-improving mechanism compared to DGM and SICA.

4.3 HGM vs. Humans: on Coding Agents Design

To gain a better understanding of its potential, we extend our evaluation of HGM by benchmarking it against the best human performance in coding agent design on SWE-Lite. We consider two settings: 1) optimization on full SWE-Verified and 2) generalization to SWE-Lite.

4.3.1 Optimization on Full SWE-Bench Verified

In this experiment, rather than using the SWE-Verified-60, we scale HGM evaluation to the full SWE-Bench Verified benchmark (500 coding challenges) with an increased number of HGM itera-

Table 3: **Generalization on SWE-Lite: HGM's Best-belief SWE-Verified Agent.** We report the accuracy of HGM's best-belief SWE-Verified agent on SWE-Lite under two settings: filtered (completely unseen) and standard (the leaderboard setting used for evaluating human-designed agents).

Coding Agents	SWE-Lite Filtered (%)	SWE-Lite Standard (%)
HGM Initial Ancestor	34.8	44.0
SWE-agent+GPT-5-mini	39.6	47.6
HGM's Best-belief SWE-Verified Agent	**40.1**	**49.0**

tions (8000 evaluations). Under this setup, the initial GPT-5-mini agent achieves 53.2% accuracy. Notably, this stronger starting point underscores the difficulty of further improvement: as task complexity grows and the search space expands, naive strategies tend to plateau.

Results &. Discussion After 8000 evaluations, HGM discovers an optimized agent that solves 61.4% of tasks, surpassing the best human-designed agent built on GPT-5-mini on the SWE-Verified leaderboard. This establishes our discovered agent as the *top-scoring* GPT-5-mini–based system, and positions it among the *top-10* agents over all checked submissions, even when compared to systems built on stronger backbone models that can cost $5\times$ more (e.g., Claude-3.7). While higher scores on the leaderboard do not necessarily indicate superior general coding ability—since both human- and machine-designed agents may overfit to the benchmark—these results demonstrate a promising potential of HGM for competing with established human-designed baselines under identical model constraints.

4.3.2 Generalization To SWE-Bench Lite

To validate that HGM's self-evolution produces agents with stronger general coding ability—rather than merely overfitting to SWE-Verified—we evaluate the top agent discovered on SWE-Verified against unseen tasks. Specifically, we compare this agent with its initial ancestor (which achieved 53.2% on SWE-Verified) using SWE-Lite, a benchmark of 300 coding tasks, 93 of which overlap with SWE-Verified. For rigor and comparability, we report two settings: (i) a filtered setting where the 93 overlapping tasks are excluded, leaving only completely unseen tasks, and (ii) the full 300-task benchmark, identical to the standard evaluation used for human designs on the leaderboard. As of the time of writing, no checked submission using GPT-5-mini appears on the SWE-Lite leaderboard. To control for backbone differences and isolate agent design, we adapt the leading system (with checked submissions) (SWE-agent + Claude 4 Sonnet) by replacing its backbone with GPT-5-mini, yielding SWE-agent + GPT-5-mini, as an additional baseline for comparison.

Results &. Discussion We show the generalization results of HGM's best-belief SWE-Verified agent on SWE-Lite benchmark in Table 3. The best-belief HGM agent found on SWE-Verified achieves 40.1% under the filtered (completely unseen) setting and 49.0% under the standard setting. Compared to its initial ancestor (34.8% and 44.0%, respectively), these gains substantiate the effectiveness of HGM's self-evolution in improving general coding ability—rather than overfitting to the optimization set. Notably, the superior performance of our HGM agent achieved on the standard SWE-Lite places it firmly *in the second place* on the SWE-Lite leaderboard among all checked submissions. Moreover, based on our local execution result of SWE-agent using the SWE-agent + Claude 4 Sonnet submission version with the same configuration, the agent optimized by HGM outperforms SWE-agent + GPT-5-mini, which achieves 39.6% (vs. 40.1% for us) on the filtered and 47.6% (vs. 49.0% for us) on the standard. This demonstrates the edge arises not from the GPT-5-mini backbone but from the genuine design improvements introduced by HGM evolution.

5 Related Works

Early imagination and formal ideals of self-improvement span learning programs and self-modifying systems (Friedberg, 1958; Samuel, 1959), with Good (1966) arguing that recursive self-enhancement could accelerate capability; early evolutionary formulations likewise proposed *self-referential learning* via variation and selection over "offspring" programs (Schmidhuber, 1987). The Gödel Machine formalizes a fully self-referential agent that rewrites its own code when it can *prove* higher expected utility (Schmidhuber, 2003; 2006); AIXI (Hutter, 2005) defines an incomputable Bayesian RL agent that is Pareto-optimal under a Solomonoff prior; and HSEARCH (Hutter, 2002) proposes proof-

based universal search with compute allocated to candidates having provable bounds. These ideals motivate the pragmatic, execution-grounded view we adopt, where agents alter their own code and validate changes empirically.

Before LLMs, pragmatic self-improvement progressed via interaction and agentic RL: The Success-Story Algorithm forces Self-Modifying Policies to come up with better and better self-modification algorithms that continually improve reward intake per time (Schmidhuber & Zhao, 1996; Schmidhuber et al., 1997); self-referential learning dynamics reduced outer-loop design via Fitness-Monotonic Execution favoring execution of models with higher ancestors' performance (Kirsch & Schmidhuber, 2022a;b); and meta-discovered update rules optimized optimizers and black-box search (Metz et al., 2021; Lange et al., 2023). Our approach follows this empirical tradition but specializes the exploration loop to an agent's *own* implementation.

The success of contemporary large language models (LLMs) opens the opportunity to automate the process of software engineering. The concept of coding agents (Yang et al., 2024; Qian et al., 2023; Wang et al., 2024; Xia et al., 2024; Hong et al., 2024) introduces an interface enabling LLMs to take control of conventional computers, offering a practical foundation for building self-improving coding agents. Two pioneer attempts of self-referential self-improving coding systems in the era of large language models initiate the route of instructing LLMs to improve the way that they are queried (Zelikman et al., 2024; Yin et al., 2024). Go further by implementing self-modifying machines as full software-engineering projects: they self-reference and self-modify by editing their own repositories and validating changes on execution-grounded software-engineering tasks (Zhang et al., 2025a; Robeyns et al., 2025). DGM and SICA, explicitly or implicitly, equate higher software-benchmark scores with greater self-improvement capacity. By contrast, HGM introduces a qualitative measure of self-improvement consistent with the theoretical Gödel Machine and directs self-modifications using estimates of this measure.

The identified tree-search problem spans fixed-budget best-arm identification (BAI), Monte Carlo Tree Search, and infinite-armed bandits, and introduces a distinct decision: explicit expansion actions that create new candidate leaves alongside ordinary evaluations. Fixed-budget BAI and Bayesian value-of-information methods assume a finite known set of arms and offer guarantees for static candidates, thus not modeling the discovery of unknown arms (Audibert & Bubeck, 2010; Karnin et al., 2013; Frazier et al., 2008). Monte-Carlo Tree Search and its UCT variants (Coulom, 2006; Kocsis & Szepesvári, 2006) alternate selection, expansion, and simulation, while their backup and selection rules typically target cumulative reward rather than fixed-budget final-choice objectives under noisy, low-signal feedback, with limited guarantees for pure exploration of leaf quality (Kaufmann & Koolen, 2017). Infinite-armed bandit formulations capture explore-discover tradeoff but typically model discoveries as i.i.d. draws from a reservoir, missing tree structure and hierarchical dependencies (Wang et al., 2008; Bubeck et al., 2011; Carpentier & Valko, 2015).

6 Conclusion

Prior works treat software-benchmark performance as a proxy for a coding agent's self-improvement ability. We identify a significant gap between benchmark scores and true self-improvement, which we term the phenomenon of metaproductivity–performance mismatch (MPM). To address it, we define the clade-metaproductivity (CMP) function as an analytic measure of an agent's self-improving ability, and prove that any policy that follows its CMP is a theoretically optimal Gödel Machine. Building on this, we introduce the Huxley–Gödel Machine (HGM), which (i) estimates CMP by aggregating clade-level information, (ii) optimizes downstream task performance, and (iii) approximates a Gödel Machine. This clade-based view cleanly decouples expansion from evaluation, enabling adaptive allocation that uses limited computational budgets more efficiently.

We validate empirically that benchmark performance is a weak approximation of CMP, and our estimator mitigates this mismatch. We demonstrated that HGM outperforms SICA and DGM on both SWE-bench Verified and Polyglot. Its gains transfer beyond the optimization set: on SWE-bench Lite, HGM surpasses a leading human-engineered coding agent (ranked number one on the official leaderboard over checked submissions) when both use GPT-5-mini. By directly optimizing metaproductivity, HGM provides a principled path toward self-modifying systems that improve reliably over time, advancing the goal of open-ended intelligence.

REPRODUCIBILITY STATEMENT

The codebase to produce our experimental results is based on the officially released GitHub Repository of Darwin Gödel Machine Zhang et al. (2025a). Adjustments have been made as detailed in Appendix C.1. We provide our source code to reproduce the results as reported in Section 4 in the supplementary material.

LLM USAGE

Large language models (LLMs) are used to help discover relevant works in the literature. In particular, they have been used for suggesting papers in upper-bound based tree search algorithms. LLMs are also used to polish the writing and fix grammatical errors.

REFERENCES

Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem, 2012. URL https://arxiv.org/abs/1111.1797.

Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *COLT-23th Conference on learning theory-2010*, pp. 13–p, 2010.

Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, 412(19):1832–1852, 2011.

Alexandra Carpentier and Michal Valko. Simple regret for infinitely many armed bandits. In *International Conference on Machine Learning*, pp. 1133–1141. PMLR, 2015.

Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pp. 72–83. Springer, 2006.

Peter I Frazier, Warren B Powell, and Savas Dayanik. A knowledge-gradient policy for sequential information collection. *SIAM Journal on Control and Optimization*, 47(5):2410–2439, 2008.

R. M. Friedberg. A learning machine: Part i. *IBM Journal of Research and Development*, 2(1):2–13, 1958. doi: 10.1147/rd.21.0002.

Paul Gauthier. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, dec 2024. Accessed: 2025-09-22.

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

John Storrs Hall. Self-improving ai: An analysis. *Minds and Machines*, 17(3):249–259, 2007.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR, 2024.

Marcus Hutter. The fastest and shortest algorithm for all well-defined problems, 2002. URL https://arxiv.org/abs/cs/0206022.

Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2005.

Julian Huxley. The three types of evolutionary process. *Nature*, 180(4584):454–455, 1957.

Intel. Autoround. https://github.com/intel/auto-round, 2025. GitHub repository. Accessed 2025-09-25.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International conference on machine learning*, pp. 1238–1246. PMLR, 2013.

Emilie Kaufmann and Wouter M Koolen. Monte-carlo tree search by best arm identification. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Louis Kirsch and Jürgen Schmidhuber. Self-referential meta learning. In *First Conference on Automated Machine Learning (Late-Breaking Workshop)*, 2022a.

Louis Kirsch and Jürgen Schmidhuber. Eliminating meta optimization through self-referential meta learning, 2022b. URL https://arxiv.org/abs/2212.14392.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Robert Lange, Tom Schaul, Yutian Chen, Tom Zahavy, Valentin Dalibard, Chris Lu, Satinder Singh, and Sebastian Flennerhag. Discovering evolution strategies via meta-black-box optimization. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, pp. 29–30, 2023.

Tor Lattimore, Csaba Szepesvari, and Gellert Weisz. Learning with good feature representations in bandits and in rl with a generative model, 2020. URL https://arxiv.org/abs/1911.07676.

Luke Metz, C Daniel Freeman, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Training learned optimizers with randomly initialized learned optimizers. *arXiv preprint arXiv:2101.07367*, 2021.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent, 2025. URL https://arxiv.org/abs/2504.15228.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. doi: 10.1147/rd.33.0210.

J. Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048, IDSIA, Manno-Lugano, Switzerland, 2003. Revised 2006.

J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin (eds.), *Artificial General Intelligence*, pp. 199–226. Springer Verlag, 2006. Variant available as arXiv:cs.LO/0309048.

Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-. hook. 1987. URL https://api.semanticscholar.org/CorpusID:264351059.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pp. 199–226. Springer, 2007.

Jürgen Schmidhuber and Jieyu Zhao. Multi-agent learning with the success-story algorithm. In *Workshop on Learning in Distributed Artificial Intelligence Systems*, pp. 82–93. Springer, 1996.

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

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Yizao Wang, Jean-yves Audibert, and Rémi Munos. Algorithms for infinitely many-armed bandits. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou (eds.), *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2008.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

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

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

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. G\" odel agent: A self-referential agent framework for recursive self-improvement. *arXiv preprint arXiv:2410.04444*, 2024.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation, August 2024. URL http://arxiv.org/abs/2310.02304.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025a. URL https://arxiv.org/abs/2505.22954.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025b.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

A GÖDEL MACHINE WITH CMP ORACLE

The proof of Gödel Machine being simulated with a CMP oracle is a simple observation that in the Gödel Machine setup CMP is an actual Q_π-Value function, as the agents above the clades are not reachable according to the design of Gödel Machines. In order to state that, we need to formalize precisely what we mean by each term.

The problem setup in the main paper is slightly different from the setup in the original Gödel Machine. Therefore, we start by introducing the Gödel Machine in the notation that aligns well with the notation used in the main text. We show the relation between the two setups. Then we will define what specifically we mean by simulating the Gödel Machine in our setup.

Proof. We define Gödel Machine Proof search as an optimal policy on the POMDP (let's call it Gödel POMDP) defined as:

We assume full access to the utility function of each agent; hence, we skip the evaluation actions from the action space.

State space Gödel POMDP uses an Extended state space, which, apart from the tree of agents $\mathcal{T}$, consists of two special agents $a_{\text{parent}}, a_{\text{child}}$ from the tree $\mathcal{T}$. For a full formal description, let's also add a budget descriptor determining the number of transitions left before the budget b is exceeded.

Observation space Observation is limited to the agents $(a_{\text{parent}}, a_{\text{child}})$ and the remaining budget b.

Action space For most of the states, the action space consists of two actions - accept and reject. As in the original Gödel Machine, the action selects the new parent in the next iteration. Intuitively, accept selects a_{child} as the new parent, and reject lets the parent remain unchanged. For simplicity of the notation, we will refer to the newly selected agent as if it were the actions, so for policy π, $a^{\text{new}}_{\text{parent}} = \pi((a_{\text{parent}}, a_{\text{child}}))$ corresponds to the action reject.

Each policy also consists of a $Score$ function that scores all possible elements in the observation. Formally, $Score : Obs \to \mathcal{T} \to \mathbb{R}$ is a function from observation into a function of agents that is applied when the budget b is equal to 0, from which the agent with the highest score is selected as the final agent.

The policy selects the highest scoring agent scored by the $Score(o)$ function. Furthermore, we restrict the score functions such that they always produce indicator functions with support as a subset of the set of observed agents, i.e., one observed agent has a score of 1 and all other agents get a score of zero.

We introduce the $Score$ function to match the notation from the original problem. It represents the selected agent at the end of the utility evaluation. For Gödel Machines, the selection function will always select the current parent as the selected agent for utility evaluation.

Transitions Transitions are generating new self-modifications from the newly selected next parent, i.e., the action output. At each transition, a self-modification of $a^{\text{new}}_{\text{parent}}$ is created and assigned as a new node a_{child}. $a^{\text{new}}_{\text{parent}}$ is becomes the a_{parent} for the next time step. Formally, if $a(a)$ is the self-modification of a, and Tr is the transition function, then $Tr((\mathcal{T}, a_{\text{parent}}, a_{\text{child}}), a, b) = (\mathcal{T} \cup \{a(a)\}, a, a(a), b-1)$ Finally, in the Gödel POMDP, the final node is implicitly selected as the last a_{parent}. That said, the final reward in Gödel POMDP is $U(\arg\max_{a \in \mathcal{T}_B} Score(a_{\text{parent}}, a_{\text{child}})(a))$

We also adapt the original problem slightly. Specifically, we match the state space with the extended space of the Gödel POMDP. Let's call it Extended MDP. Similarly to Gödel POMDP, its state is a tree $\mathcal{T}$ and two special agents $a_{\text{parent}}, a_{\text{child}}$, and the budget b. The action space is the same as in our problem and consists of a single agent $a \in \mathcal{T}$ when there is still some budget left. In other cases, the policy uses its $Score : \mathcal{T} \to \mathcal{T} \to \mathbb{R}$ function that scores created agents in the trees for the selection of the final agent. The transition function adds the self-modification to the tree and sets a as the new a_{parent} and $a(a)$ as the new a_{child}.

We have extended the state space of our original problem. The new problem is also an MDP (call it an extended MDP). The new state consists of the tree and two special agents $(a1, a2)$ from the tree.

At each step, the policy selects a single agent a to self-modify. The tree is expanded, the selected agent becomes a new special agent $a1$, and the modification $a(a)$ becomes the special agent $a2$.

Every policy from Gödel POMDP can be mapped to a policy in the extended MDP. Specifically, let π_G be the policy in the Gödel POMDP, then we define the corresponding policy in the extended MDP π_E as $\pi_E((\mathcal{T}, a_{\text{parent}}, a_{\text{child}})) = \pi_G(a_{\text{parent}}, a_{\text{child}})$. At each node, the policy chooses between the two special nodes, ignoring the rest of the tree.

Now we adapt CMP to Gödel POMDP. CMP adapted to the Gödel case is still a function of a state in the extended state space (not only observation) and agent.

$$
\begin{aligned}
&\mathrm{CMP}_{\pi,Score}((\mathcal{T}, a_p, a_c, b), a) \\
=&\mathbb{E}_{(\mathcal{T}_B, a_{Bp}, a_{Bc}, 0)\sim p_\pi(\cdot|(\mathcal{T}, a_p, a_c, b), a)} \left[U\Big(\operatorname*{arg\,max}_{a' \in \mathcal{C}(\mathcal{T}_B, a)} Score(a_{Bp}, a_{Bc})(a') \Big) \right] \\
=&\mathbb{E}_{(\mathcal{T}_B, a_{Bp}, a_{Bc}, 0)\sim p_\pi(\cdot|(\mathcal{T}, a_p, a_c, b), a)} \left[U\Big(\operatorname*{arg\,max}_{a' \in \{a_{Bp}, a_{Bc}\}} Score(a_{Bp}, a_{Bc})(a') \Big) \right] \\
=&Q_{\pi,Score}((\mathcal{T}, a_p, a_c, b), a).
\end{aligned}
$$

The second equality comes from the fact that the support of the $Score$ function is a set with an element from the current observation. The third equality comes from directly unrolling the Q-value function.

The Gödel Machine is defined by a prover that produces a proof whether accepting or selecting a given node (or rejecting). Here, the algorithm by having access to CMP has access to the true Q value function in the Gödel POMDP. This serves as a proof of better utility of either parent or the child. Additionally, we state that we break the tie by selecting the parent node as in the original Gödel Machine. Hence, the algorithm that follows it is a Gödel Machine.

Independently, as this proof directly shows that Gödel Machine selects an action that maximizes its own Q-value function, it is optimal due to the Bellman Optimality Equation. With the procedure shown above, we can adapt it to the extended MDP.

□

B ALGORITHM

Algorithm 1 presents the procedure of HGM.

In the standard HGM, when an expansion ends with a new a added to the tree, the next following evaluation actions have the choice of choosing a. At the beginning of the search, the first actions have the most dense ratio of expansion actions, and the following evaluation actions would diverge more from the standard HGM since most of their choices are not generated yet, concentrating on the early finished expansions. Therefore, to avoid this bias, we initialize by expanding the initial agent 5 times with each of the processes in parallel.

The asynchronization also introduces another bias that favors agents with fewer evaluated results to be more often selected for evaluation than in the standard HGM. This is due to the fact that the easy tasks usually stop earlier than the difficult ones. This yields agents with fewer evaluations having a higher empirical mean. During our experiments, we observed that many agents were successful on the first ten evaluated tasks; however, this accuracy drops quickly as the number of evaluations grows. After having more than 50 evaluations, this bias is barely observed.

Algorithm 1 Huxley–Gödel Machine (HGM)

Input: the initial agent a_0, widening parameter α, and the percentile ϵ for final selection
Initialize a tree $\mathcal{T}$ with root a_0
Initialize counters $n^C_{\text{success}}(a), n^C_{\text{failure}}(a), n_{\text{success}}(a), n_{\text{failure}}(a)$ for all $a \in \mathcal{T}$
while Computational Budget not Exceeded **do**
  if $|\mathcal{T}| \leq n^\alpha$ and expandable parents exist **then**
    Expand:
    for each node $a \in A$ **do**
      Sample $S_C(a) \sim \text{Beta}(\tau(1 + n^C_{\text{success}}(a)),\ \tau(1 + n^C_{\text{failure}}(a)))$
    end for
    Select node $a^\star = \arg\max_a S_C(a)$
    Create child c by self-modification of $a^\star$
    Add c to $\mathcal{T}$
  else
    Evaluate:
    for each agent $a \in A$ with remaining tasks **do**
      Sample $S(a) \sim \text{Beta}(\tau(1 + n_{\text{success}}(a)),\ \tau(1 + n_{\text{failure}}(a)))$
    end for
    Select agent $a^\star = \arg\max_a S(a)$
    Allocate a benchmark task to $a^\star$
    update $n_{\text{success}}, n_{\text{failure}}$ for $a^\star$
    update $n^C_{\text{success}}, n^C_{\text{failure}}$ for $a^\star$ and ancestors
  end if
end while
Return $\text{argmax}_{a \in \mathcal{T}_B} I_\epsilon(1 + n_{\text{success}(a)}, 1 + n_{\text{failure}}(a))$

C Experimental Details

C.1 Initial Agents

Our initial agents applied in Section 4.2 are adopted from the official implementation of DGM with minor changes, including modifying API support, setting up a timeout option, and adding a length of LLM interaction restriction. The initial agent is essentially a single loop of LLM queries with two tool options, i.e., file editing and bash command execution. We set a time limit of one hour for each agent execution.

The initial agents used in SWE-bench experiments and Polyglot experiments differ in that the Polyglot initial agent includes test commands with different programming language support. There are two additional functions in the SWE-bench initial agent that serve to summarize existing tests and execute the tests with a report generated, respectively.

The initial agent employed in Section 4.3 is further adjusted by removing the file-editing tool, leaving only the bash tool, to minimize initial inductive bias. The time limit is extended to five hours for both self-modification and task evaluation, reducing the risk of prematurely eliminating stronger agents due to time constraints.

C.2 Other Details

For the Polyglot experiments presented in Section 4.2, the exact large language model used for self-modification is an int4 and int8 mixed quantized version of Qwen3-Coder-480B-A35B-Instruct generated by AutoRound (Intel, 2025). It is impossible to report the detailed expansions of each method we tested in the experiments since there is no way to force the evolving agents to log them. A self-improving agent can always self-modify to remove the log function. During our experiments, we do observe such self-modification happening. Overall, we spent approximately $5000 USD to produce all experimental results.

D EMPIRICAL CMP AND ITS ESTIMATION

In this section, we provide the exact formula to compute the empirical CMP and the variant of our CMP estimator being used in Section 4.1 for correlation analysis. The empirical CMP of an agent a as a node in a tree is defined as

$$\max_{a' \in C(a) \setminus \{a\}} \frac{n_{\text{success}}(a')}{n_{\text{success}}(a') + n_{\text{failure}}(a')}.$$

The prediction of our CMP estimator is defined as

$$\frac{n^C_{\text{success}}(a) - n_{\text{success}}(a) - n^C_{\text{success}}(b^*)}{n^C_{\text{failure}}(a) - n_{\text{failure}}(a) - n^C_{\text{failure}}(b^*) + n^C_{\text{success}}(a) - n_{\text{success}}(a) - n^C_{\text{success}}(b^*)},$$

where b^* is a child of a such that

$$\left(\text{argmax}_{n \in C(a)} \frac{n_{\text{success}}(n)}{n_{\text{failure}}(n)} \right) \cap C(b^*) \neq \emptyset.$$

For both SICA and DGM, we consider the benchmark performance of an agent as their estimator of the agent's CMP.

E BASELINES

Table 2 summarizes the three subpolicies of SICA, DGM, and HGM, which define solutions to the iterative tree search problem defined in 2.

Subpolicy	**SICA**	**DGM**	**HGM (Ours)**
Mod vs. Eval	Alternates between modification and evaluation.	Alternates between modification and evaluation.	Adaptive choice between modification and evaluation.
Modification Strategy	Greedily selects the agent with the best performance up to this point and modifies it with the entire history accessible to the agent.	Selects the node probabilistically based on the evaluation metric and the number of children of the agents.	Selects the node based on the statistics of the *clade* stemming from a given node.
Evaluation Strategy	Evaluates the most recently created agent on the entire evaluation dataset.	Progressively evaluates the last created agent on subsets of the dataset, expanding if results are promising.	Selects the agent based on the statistics and evaluates it on a single task.

Table 4: Comparison of structured policies across self-improving methods. Each method is described by three subpolicies: Mod vs. Eval, Modification Strategy, and Evaluation Strategy.

F DISCOVERED AGENTS

We present interesting findings about HGM-discovered agents during our experiments. By manually inspecting the diff patch files that were generated by self-modification, we make interesting observations.

One engaging self-modification we found implements iterative refinement to make incremental improvements. In the self-improving context, it means that during one expansion (self-modification) step, the agent is accurately performing multiple self-modifications. More interestingly, this phenomenon can be viewed as the agent being self-motivated to perform self-modification, instead of being instructed to do so by humans. This self-motivated repetitive self-modification automatically creates arbitrary levels of meta-improvement in the sense that each self-modification ideally needs to consider all its subsequent self-modifications, which the subsequent self-modifications also need to consider future self-modifications. We show in List 1 the part of the patch file that includes the implementation of this self-modification.

Another interesting phenomenon we observe is the file structures that self-modifying agents create. Many agents tend to write the chunk they self-modify into a diff patch file, usually with a constant file name. When multiple consecutive attempts of these self-modifying actions apply, the structure of these git diff files becomes abstruse. Imagine that there is a diff file that includes certain edits. Once there are changes applied to this file, it then generates the diff patch file. In this new patch file, a nested format of editing record appears. At the beginning of some lines, there are multiple + and - signs that correspond to nested file edits. This is a natural format to present information that inspires long-sighted self-improvement since it shows how self-modifications indeed influence subsequent self-modifications. The authors find it mind-bending to understand such a file structure and parse the semantics manually. For readers interested in further investigation, we provide one instance of such a file structure in List 2.

Listing 1: The self-modifying agent that executes multiple self-modifications per instruction.

```
+def attempt_error_resolution(git_dir, test_output, test_error,
    ↪ language):
+    """
+    Attempt to automatically diagnose and resolve errors.
+    Returns a tuple of (resolved, message) where resolved indicates if
    ↪ errors were fixed.
+    """
+    safe_log("Attempting automated error diagnosis and resolution...")
+
+    # Diagnose errors using our enhanced bash tool function
+    diagnosis = diagnose_errors(test_output, test_error, "")
+
+    if not diagnosis["has_errors"]:
+        return False, "No errors detected to resolve."
+
+    resolution_messages = []
+
+    # Try to apply automated fixes for each diagnosed error
+    for error in diagnosis["errors"]:
+        safe_log(f"Processing error: {error['type']} -
    ↪ {error['description']}")
+
+        # Simple resolution strategies based on error type
+        if error["type"] == "python_module_not_found":
+            # For Python module not found errors, we might install the
    ↪ module
+            match = re.search(r"No module named '([^']+)'",
    ↪ error["description"])
+            if match:
+                module = match.group(1)
+                resolution_messages.append(f"Would attempt to install
    ↪ Python module: {module}")
+                # In practice, we would run: pip install {module}
```

```
+                # But we'll skip actual installation to avoid side
    ↪ effects
+
+        elif error["type"] == "python_syntax_error" and "file" in error:
+            # For syntax errors, we could potentially apply fixes
+            file_path = os.path.join(git_dir, error["file"])
+            if os.path.exists(file_path):
+                resolution_messages.append(f"Would attempt to fix
    ↪ syntax error in {file_path} at line {error.get('line',
    ↪ 'unknown')}")
+                # In practice, we would use the editor tool's apply_fix
    ↪ command
+                # This is just a demonstration of what could be done
+
+        elif error["type"] == "test_failure":
+            # For test failures, we might suggest reviewing the
    ↪ implementation
+            resolution_messages.append("Would analyze test failures and
    ↪ suggest implementation improvements")
+
+    if resolution_messages:
+        return True, "Automated resolution attempted:\n" +
    ↪ "\n".join(resolution_messages)
+    else:
+        return False, "No automated resolutions available for detected
    ↪ errors."
+
class AgenticSystem:
     def __init__(
                self,
@@ -243,6 +293,16 @@ Your task is to make changes to the files in the
    ↪ {self.git_dir} directory to add

                safe_log(f"Attempt {attempt + 1} test results: {'PASSED' if
    ↪ test_success else 'FAILED'}")

+            # If tests failed, attempt automated error resolution
+            if not test_success:
+                resolved, resolution_message = attempt_error_resolution(
+                    self.git_dir, test_output, test_error, self.language
+                )
+                safe_log(f"Error resolution: {resolution_message}")
+
+                # Even if we couldn't automatically resolve, we still
    ↪ provide feedback
+                # In a more advanced implementation, we might actually
    ↪ apply fixes here
+
                # If this is the first attempt or tests passed and we
    ↪ didn't have a successful attempt yet, update best patch
                if attempt == 0 or (test_success and (best_patch is None or
    ↪ not best_test_results)):
                    best_patch = current_patch
@@ -278,37 +338,31 @@ Please revise your code to fix these issues and
    ↪ try again.
         # Log final summary
         safe_log(f"\n{'='*20} FINAL SUMMARY {'='*20}")
         safe_log(f"Best solution found on attempt:
    ↪ {best_test_results['attempt'] if best_test_results else 'None'}")
-        safe_log(f"Tests passed: {best_test_results['test_success'] if
    ↪ best_test_results else 'Unknown'}")
+        safe_log(f"Final test result: {'PASSED' if best_test_results
    ↪ and best_test_results['test_success'] else 'FAILED'}")
+
+        if best_test_results:
```

```
+            safe_log(f"Final test
    ↪ output:\n{best_test_results['test_output']}")
+            if best_test_results['test_error']:
+                safe_log(f"Final test
    ↪ errors:\n{best_test_results['test_error']}")

-        # Save attempt history to a file
-        history_file =
    ↪ os.path.join(os.path.dirname(self.chat_history_file),
    ↪ 'attempt_history.md')
-        with open(history_file, 'w') as f:
-            f.write("# Attempt History\n\n")
-            for result in self.attempt_history:
-                f.write(f"## Attempt {result['attempt']}\n")
-                f.write(f"**Tests Passed**: {result['test_success']}\n")
-                f.write(f"**LLM Calls Used**: {result['llm_calls']}\n")
-                f.write(f"**Test
    ↪ Output**:\n```\n{result['test_output']}\n```\n")
-                f.write(f"**Test
    ↪ Error**:\n```\n{result['test_error']}\n```\n")
-                f.write(f"**Patch**:\n```\n{result['patch']}\n```\n\n")
+        return bool(best_test_results and
    ↪ best_test_results['test_success'])
```

Listing 2: An instance of the nested diff patch format.

```
diff --git a/attempt_history.md b/attempt_history.md
new file mode 100644
index 0000000..b132b1a
--- /dev/null
+++ b/attempt_history.md
@@ -0,0 +1,727 @@
+# Attempt History
+
+## Attempt 1
+**Tests Passed**: True
+**LLM Calls Used**: 18
+**Test Output**:
+```
+============================= test session starts
    ↪ ==============================
+platform linux -- Python 3.10.18, pytest-8.4.2, pluggy-1.6.0 --
    ↪ /usr/local/bin/python3.10
+cachedir: .pytest_cache
+rootdir: /dgm
+configfile: pytest.ini
+testpaths: tests
+plugins: asyncio-1.1.0, anyio-4.10.0
+asyncio: mode=strict, asyncio_default_fixture_loop_scope=None,
    ↪ asyncio_default_test_loop_scope=function
+collecting ... collected 29 items
+
+tests/test_bash_tool.py::TestBashTool::test_simple_command PASSED
    ↪  [  3%]
+tests/test_bash_tool.py::TestBashTool::test_multiple_commands PASSED
    ↪  [  6%]
+tests/test_bash_tool.py::TestBashTool::test_command_with_error PASSED
    ↪  [ 10%]
+tests/test_bash_tool.py::TestBashTool::test_environment_variables
    ↪ PASSED [ 13%]
+tests/test_bash_tool.py::TestBashTool::test_command_output_processing
    ↪ PASSED [ 17%]
+tests/test_bash_tool.py::TestBashTool::test_long_running_command PASSED
    ↪  [ 20%]
```

```
+tests/test_bash_tool.py::TestBashTool::test_invalid_commands[invalid_command_name]
    ↪ PASSED [ 24%]
+tests/test_bash_tool.py::TestBashTool::test_invalid_commands[cd
    ↪ /nonexistent/path] PASSED [ 27%]
+tests/test_bash_tool.py::TestBashTool::test_invalid_commands[/bin/nonexistent]
    ↪ PASSED [ 31%]
+tests/test_bash_tool.py::TestBashTool::test_command_with_special_chars
    ↪ PASSED [ 34%]
+tests/test_bash_tool.py::TestBashTool::test_multiple_line_output PASSED
    ↪  [ 37%]
+tests/test_bash_tool.py::TestBashTool::test_large_output_handling
    ↪ PASSED [ 41%]
+tests/test_edit_tool.py::TestEditorTool::test_view_file PASSED
    ↪  [ 44%]
+tests/test_edit_tool.py::TestEditorTool::test_create_file PASSED
    ↪  [ 48%]
+tests/test_edit_tool.py::TestEditorTool::test_create_existing_file
    ↪ PASSED [ 51%]
+tests/test_edit_tool.py::TestEditorTool::test_edit_file PASSED
    ↪  [ 55%]
+tests/test_edit_tool.py::TestEditorTool::test_edit_nonexistent_file
    ↪ PASSED [ 58%]
+tests/test_edit_tool.py::TestEditorTool::test_view_directory PASSED
    ↪  [ 62%]
+tests/test_edit_tool.py::TestEditorTool::test_invalid_path PASSED
    ↪  [ 65%]
+tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[unknown_command]
    ↪ PASSED [ 68%]
+tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[] PASSED
    ↪  [ 72%]
+tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[None]
    ↪ PASSED [ 75%]
+tests/test_error_diagnosis.py::TestErrorDiagnosis::test_python_syntax_error_diagnosis
    ↪ PASSED [ 79%]
+tests/test_error_diagnosis.py::TestErrorDiagnosis::test_python_module_not_found_diagnosis
    ↪ PASSED [ 82%]
+tests/test_error_diagnosis.py::TestErrorDiagnosis::test_no_error_diagnosis
    ↪ PASSED [ 86%]
+tests/test_error_diagnosis.py::TestErrorDiagnosis::test_format_diagnosis_with_errors
    ↪ PASSED [ 89%]
+tests/test_error_diagnosis.py::TestErrorDiagnosis::test_format_diagnosis_without_errors
    ↪ PASSED [ 93%]
+tests/test_error_diagnosis.py::TestAutomatedFixes::test_apply_missing_import_fix
    ↪ PASSED [ 96%]
+tests/test_error_diagnosis.py::TestAutomatedFixes::test_apply_syntax_error_fix
    ↪ PASSED [100%]
+
+================================== PASSES
    ↪ ==================================
+========================== short test summary info
    ↪ ============================
+PASSED tests/test_bash_tool.py::TestBashTool::test_simple_command
+PASSED tests/test_bash_tool.py::TestBashTool::test_multiple_commands
+PASSED tests/test_bash_tool.py::TestBashTool::test_command_with_error
+PASSED tests/test_bash_tool.py::TestBashTool::test_environment_variables
+PASSED
    ↪ tests/test_bash_tool.py::TestBashTool::test_command_output_processing
+PASSED tests/test_bash_tool.py::TestBashTool::test_long_running_command
+PASSED
    ↪ tests/test_bash_tool.py::TestBashTool::test_invalid_commands[invalid_command_name]
+PASSED tests/test_bash_tool.py::TestBashTool::test_invalid_commands[cd
    ↪ /nonexistent/path]
+PASSED
    ↪ tests/test_bash_tool.py::TestBashTool::test_invalid_commands[/bin/nonexistent]
```

```
+PASSED
    ↪ tests/test_bash_tool.py::TestBashTool::test_command_with_special_chars
+PASSED tests/test_bash_tool.py::TestBashTool::test_multiple_line_output
+PASSED tests/test_bash_tool.py::TestBashTool::test_large_output_handling
+PASSED tests/test_edit_tool.py::TestEditorTool::test_view_file
+PASSED tests/test_edit_tool.py::TestEditorTool::test_create_file
+PASSED
    ↪ tests/test_edit_tool.py::TestEditorTool::test_create_existing_file
+PASSED tests/test_edit_tool.py::TestEditorTool::test_edit_file
+PASSED
    ↪ tests/test_edit_tool.py::TestEditorTool::test_edit_nonexistent_file
+PASSED tests/test_edit_tool.py::TestEditorTool::test_view_directory
+PASSED tests/test_edit_tool.py::TestEditorTool::test_invalid_path
+PASSED
    ↪ tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[unknown_command]
+PASSED tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[]
+PASSED
    ↪ tests/test_edit_tool.py::TestEditorTool::test_invalid_commands[None]
+PASSED
    ↪ tests/test_error_diagnosis.py::TestErrorDiagnosis::test_python_syntax_error_diagnosis
+PASSED
    ↪ tests/test_error_diagnosis.py::TestErrorDiagnosis::test_python_module_not_found_diagnosi
+PASSED
    ↪ tests/test_error_diagnosis.py::TestErrorDiagnosis::test_no_error_diagnosis
+PASSED
    ↪ tests/test_error_diagnosis.py::TestErrorDiagnosis::test_format_diagnosis_with_errors
+PASSED
    ↪ tests/test_error_diagnosis.py::TestErrorDiagnosis::test_format_diagnosis_without_errors
+PASSED
    ↪ tests/test_error_diagnosis.py::TestAutomatedFixes::test_apply_missing_import_fix
+PASSED
    ↪ tests/test_error_diagnosis.py::TestAutomatedFixes::test_apply_syntax_error_fix
+============================== 29 passed in 3.58s
    ↪ ==============================
+
+```
+**Test Error**:
+```
+
+```
+**Patch**:
+```
+diff --git a/coding_agent.py b/coding_agent.py
+index 78e8ad4..77e5097 100644
+--- a/coding_agent.py
++++ b/coding_agent.py
+@@ -5,9 +5,13 @@ from logging.handlers import RotatingFileHandler
+ import os
+ import threading
+ import time
++import json
++import re
+
+ from llm_withtools import CLAUDE_MODEL, OPENAI_MODEL, chat_with_agent
+ from utils.git_utils import diff_versus_commit, reset_to_commit,
    ↪ apply_patch
++from tools.bash import diagnose_errors
++from tools.edit import apply_automated_fix, read_file, write_file
+
+ # reset_to_commit(git_dname, commit)
+ # apply_patch(git_dname, patch_str)
+@@ -136,6 +140,52 @@ def run_tests(git_dir, language):
+         # Always change back to original directory
+         os.chdir(original_cwd)
+
```

```
++def attempt_error_resolution(git_dir, test_output, test_error,
    ↪ language):
++    """
++    Attempt to automatically diagnose and resolve errors.
++    Returns a tuple of (resolved, message) where resolved indicates if
    ↪ errors were fixed.
++    """
```