HyperAgents

Jenny Zhang[1,2†], **Bingchen Zhao**[3†], **Wannan Yang**[4†], **Jakob Foerster**[6]
Jeff Clune[1,2,5], **Minqi Jiang**[‡], **Sam Devlin**[7], **Tatiana Shavrina**[6]

[1]University of British Columbia, [2]Vector Institute, [3]University of Edinburgh
[4]New York University, [5]Canada CIFAR AI Chair, [6]FAIR at Meta, [7]Meta Superintelligence Labs
[†]Work done during internship at Meta, [‡]Work done at Meta

Self-improving AI systems aim to reduce reliance on human engineering by learning to improve their own learning and problem-solving processes. Existing approaches to recursive self-improvement typically rely on fixed, handcrafted meta-level mechanisms, which fundamentally limit how fast such systems can improve. The Darwin Gödel Machine (DGM) (Zhang et al., 2025b) demonstrates that open-ended self-improvement is achievable in coding. Starting from a single coding agent, the DGM repeatedly generates and evaluates self-modified variants, forming a growing archive of stepping stones for future improvement. Because both evaluation and self-modification are coding tasks, gains in coding ability can translate into gains in self-improvement ability. However, this alignment does not generally hold beyond coding domains. We introduce **hyperagents**, self-referential agents that integrate a task agent (which solves the target task) and a meta agent (which modifies itself and the task agent) into a single editable program. Crucially, the meta-level modification procedure is itself editable, enabling metacognitive self-modification, improving not only task-solving behavior, but also the mechanism that generates future improvements. We instantiate this framework by extending DGM to create DGM-Hyperagents (DGM-H). By allowing the improvement procedure to evolve, the DGM-H eliminates the assumption of domain-specific alignment between task performance and self-modification skill, and can potentially support self-accelerating progress on any computable task. Across diverse domains (coding, paper review, robotics reward design, and Olympiad-level math-solution grading), the DGM-H improves performance over time and outperforms baselines without self-improvement or open-ended exploration, as well as prior self-improving systems like DGM. We further show that the DGM-H improves the process by which it generates new agents (e.g., persistent memory, performance tracking), and that these meta-level improvements transfer across domains and accumulate across runs. All experiments were conducted with safety precautions (e.g., sandboxing, human oversight). We discuss what safety entails in this setting and the broader implications of self-improving systems. DGM-Hyperagents offer a glimpse of open-ended AI systems that do not merely search for better solutions, but continually improve their search for how to improve.

Date: March 23, 2026
Correspondence: Jenny Zhang at jennyzzt@cs.ubc.ca, Tatiana Shavrina at rybolos@meta.com
Code: https://github.com/facebookresearch/Hyperagents

Meta

1 Introduction

With appropriate safety considerations, AI systems that can improve themselves could transform scientific progress from a human-paced process into an autonomously accelerating one, thereby allowing society to realize the benefits of technological advances much earlier. Such self-improving AI seeks to continually improve its own learning and task-solving abilities. However, most existing self-improvement architectures rely on a fixed meta agent (i.e., a higher-level system that modifies a base system). This creates a limitation since the base system can only be improved within the boundaries defined by the meta agent's design. Adding a meta-meta system to improve the meta agent does not solve this problem, it merely shifts the issue upward and ultimately leads to an infinite regress of meta-levels. To overcome this limitation and allow a system to modify any part of itself without being constrained by its initial implementation, the system must be

self-referential, that is, able to analyze, modify, and evaluate itself (Kirsch and Schmidhuber, 2022; Zhang et al., 2025b). When the mechanism of improvement is itself subject to improvement, progress can become self-accelerating and potentially unbounded (Lu et al., 2023).

The Darwin Gödel Machine (DGM) (Zhang et al., 2025b) demonstrates that open-ended self-improvement is achievable in coding. In the DGM, agents generate and evaluate modifications to their own code, and successful variants are retained in an archive as stepping stones for further improvement. However, the DGM relies on a handcrafted, fixed mechanism to produce self-improvement instructions (Appendix B). This mechanism analyzes past evaluation results and the agent's current codebase to generate an instruction directing where the agent should self-improve. This mechanism is not modifiable. Hence, the DGM's capacity for self-improvement is bottlenecked by this fixed instruction-generation step. Despite this handcrafted step, the DGM can still improve at self-improving. Because both evaluation and self-modification are coding tasks, improvements in evaluation performance directly reflects the agent's capacity to generate effective self-modifications. To improve at self-improving, the DGM relies on a limiting assumption: that the skills required to solve the evaluation tasks are the same as those required for effective self-reflection and self-modification. This assumption is unlikely to hold outside coding domains, where task-solving skills may differ substantially from the skills needed to analyze failures, propose effective self-improvements, and implement them.

This work introduces *hyperagents*, self-referential agents that can in principle self-improve for any computable task. Here, an *agent* is any computable program, optionally including calls to foundation models (FMs), external tools, or learned components. A *task agent* solves a given task. A *meta agent* modifies agents and generates new ones. A hyperagent combines the task agent and the meta agent into a single self-referential, modifiable program, such that the mechanism responsible for generating improvements is itself subject to modification. As a result, a hyperagent can improve not only how it solves tasks (i.e., the task agent), but also how it generates and applies future modifications (i.e., the meta agent). Because its self-improvement mechanism is itself modifiable, we call this *metacognitive self-modification*. We extend the DGM with hyperagents, creating DGM-Hyperagents (DGM-H). The DGM-H retains the open-ended exploration structure of the DGM and extends the DGM with metacognitive self-modification. As with DGM, to support sustained progress and avoid premature convergence, the DGM-H grows an archive of hyperagents by branching from selected candidates, allowing them to self-modify, evaluating the resulting hyperagents, and adding them back to the archive. Because a hyperagent can modify its self-modification process, the DGM-H is not constrained by its initial implementation and can potentially self-improve for any computable task.

Across our experiments, the DGM-H demonstrates substantial and generalizable improvements in both task performance and self-improvement ability. On the Polyglot coding benchmark (Gauthier, 2024), the DGM-H achieves gains comparable to the most established prior self-improving algorithm (the Darwin Gödel Machine, Zhang et al., 2025b), despite not being handcrafted for coding. Beyond coding, the DGM-H substantially improves performance on paper review (Zhao et al., 2026) and robotics reward design (Genesis, 2024), with gains transferring to held-out test tasks and significantly outperforming prior self-improving algorithms, which struggle outside coding unless customized. Ablations without self-improvement or without open-ended exploration show little to no progress, highlighting the necessity of each component (Section 5.1). Crucially, the DGM-H learns transferable mechanisms on how to self-improve (e.g., persistent memory, performance tracking) that systematically improve its ability to generate better task or meta agents over time. As a result, meta-level improvements learned by the DGM-H transfer across domains. Specifically, hyperagents optimized in one setting (i.e., paper review and robotics tasks) remain significantly effective at generating improved task agents in a different domain (i.e., Olympiad-level math grading) (Section 5.2). We further show that self-improvements learned by the DGM-H in one setting can compound with continued self-improvement in another setting (Section 5.3). This suggests that, given appropriate tasks, the DGM-H has the potential to achieve unbounded open-ended self-improvement over time. We discuss the safety implications of such open-ended self-improving systems and outline practical considerations for responsible deployment in Section 6. Overall, hyperagents open up the possibility of improving their ability to improve while improving their ability to perform any computable task.

2 Related Work

Open-Endedness. Open-endedness refers to the ability of a system to continually invent new, interesting, and increasingly complex artifacts, extending its own frontier of discovery without a fixed objective or predefined end (Stanley et al., 2017; Hughes et al., 2024). Recent work has leveraged FMs as proxies for human interestingness and as versatile engines for generating and evaluating novel behaviors across diverse domains (Zhang et al., 2024; Faldor et al., 2025). Building on these advances, recent progress in open-ended learning (Hu et al., 2025; Zoph and Le, 2017; Colas et al., 2023; Lehman et al., 2023) and quality-diversity algorithms (Lehman and Stanley, 2011; Mouret and Clune, 2015; Bradley et al., 2023; Samvelyan et al., 2024; Ding et al., 2024; Pourcel et al., 2023; Coiffard et al., 2025; Dharna et al., 2025; Yuan et al., 2026) has shown that sustained exploration can produce diverse and increasingly capable artifacts across domains ranging from game-playing agents (Klissarov et al., 2023, 2025; Wang et al., 2024) to scientific discovery (Lu et al., 2024a,b; Romera-Paredes et al., 2024; Novikov et al., 2025; Audran-Reiss et al., 2025) and robotic control (Cully et al., 2015; Li et al., 2024; Grillotti et al., 2025). Recent progress has shown that open-ended AI systems capable of continuously generating diverse and increasingly complex artifacts are possible (Zhang et al., 2024; Faldor et al., 2025; Hu et al., 2025). An important next step is to explore how such systems can achieve compounding improvement. In human scientific and technological progress, advances often build on prior advances not only by producing better artifacts, but also by improving the tools and processes that generate future discoveries, leading to accelerating innovation (Good, 1966; Kwa et al., 2025). Inspired by this pattern, we focus on open-ended systems that can improve not only the artifacts they generate, but also the mechanisms by which novelty and progress are produced (Clune, 2019; Jiang et al., 2023).

Self-improving AI. Early theoretical work on self-improving AI dates back to formal models of self-modifying agents (Hutter, 2003). One prominent example is the Gödel Machine (Schmidhuber, 2003), which proposes agents that rewrite themselves when provably beneficial, though such approaches remain impractical in real-world settings. Subsequent research explored self-improvement through adaptive neural systems, in which agents modify their own weights or learning dynamics via meta-learning (Schmidhuber, 1993; Miconi et al., 2018; Javed and White, 2019; Beaulieu et al., 2020; Miconi et al., 2020; Irie et al., 2022; Chalvidal et al., 2022; Oh et al., 2025), evolution (Stanley and Miikkulainen, 2002; Lange et al., 2023; Qiu et al., 2025; Zhao et al., 2025), or self-play (Silver et al., 2016, 2017; Xia et al., 2025b, 2026). Notably, Silver et al. (2017) use self-play to iteratively improve neural network agents, achieving superhuman performance in domains such as Go and chess, although the underlying learning algorithms themselves remain fixed and human-designed. More recently, FMs have enabled self-improvement through iterative refinement of prompts (Fernando et al., 2023; Wang et al., 2025a; Zhang et al., 2025c,a; Ye et al., 2026), reasoning traces (Zelikman et al., 2022; Yin et al., 2025; Havrilla et al., 2024; Zhuge et al., 2024), and entire code repositories (Zhang et al., 2025b; Wang et al., 2025b; Xia et al., 2025a), as well as through systems that update model weights using self-generated data or interaction (Wu et al., 2024; Zweiger et al., 2025; Wen et al., 2025; Wei et al., 2025b). Among these, the Darwin Gödel Machine (DGM) (Zhang et al., 2025b) stands out as a practical instantiation of recursive self-improvement in coding domains. However, despite their effectiveness, most existing approaches (including the DGM and its derivatives) rely on fixed, handcrafted meta-level mechanisms (Appendix B) that constrain how self-improvement can compound over time and generalize across domains.

Self-referential Meta-learning. Self-referential meta-learning studies systems that learn to improve the mechanisms by which learning occurs. Prior work has explored this idea in neural networks (Kirsch and Schmidhuber, 2022; Jackson et al., 2024) and evolutionary methods (Lu et al., 2023). More recently, several works have explored self-referential improvement using FM-based agents (Zelikman et al., 2024; Robeyns et al., 2025; Yin et al., 2025; Zhang et al., 2025b). The Darwin Gödel Machine (DGM) (Zhang et al., 2025b) and its successors (Wang et al., 2025b; Xia et al., 2025a; Weng et al., 2026) instantiate recursive self-improvement through self-modification, primarily in coding domains. However, these approaches improve at improving primarily within coding tasks only. In the DGM and related systems, a coding agent is tasked with improving itself, and the resulting improved coding agent is then used in subsequent self-improvement steps to generate an even better version of itself. Because both the evaluation task and the self-modification process involve coding, improving the coding agent also enhances the system's ability to carry out future self-improvements. However, this property only holds when the evaluation task and the self-modification task are closely aligned. For example, if the evaluation task were instead poetry writing, improving an agent's poetry-writing ability would

not necessarily improve its ability to modify its own code. Prior work therefore relies on an *alignment* between the evaluation task and the skills required for self-improvement. In contrast, hyperagents do not assume such alignment, because the self-modification mechanism is fully modifiable and not tied to any particular task domain. Hence, hyperagents can improve both task performance and the process of improvement itself across any computable task.

3 Methods

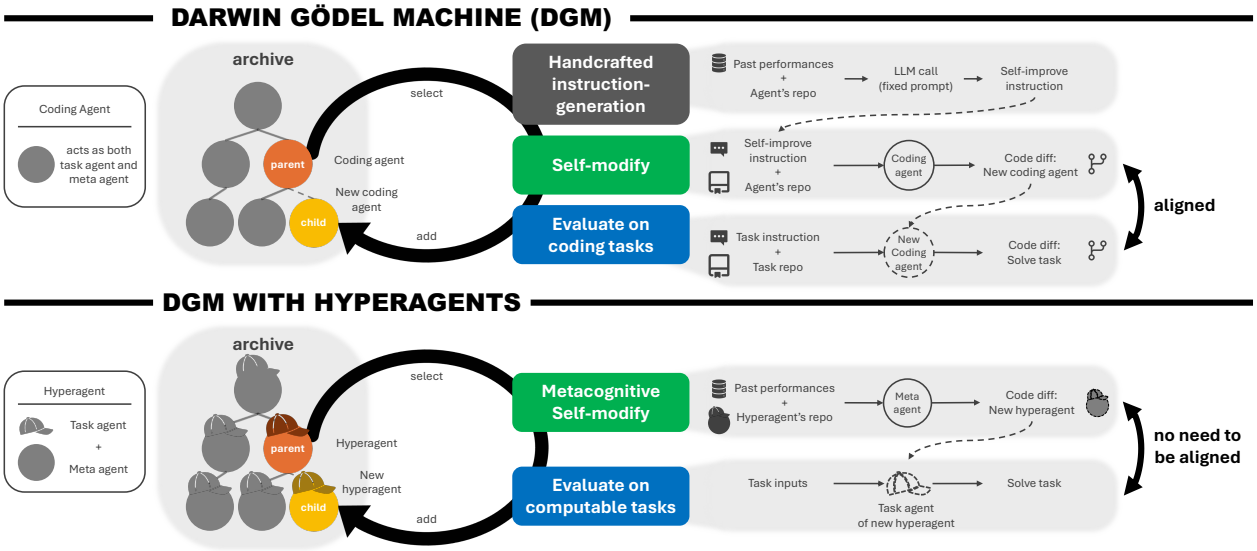

Figure 1 The Darwin Gödel Machine with Hyperagents. The DGM-Hyperagents (DGM-H) extends the Darwin Gödel Machine (DGM) (Zhang et al., 2025b) beyond coding tasks, enabling agents to improve not only their task performance but also their ability to improve themselves, across any computable task. (Top) In the DGM, a coding agent evolves through open-ended exploration by generating and evaluating self-modified variants, which are stored in an archive of stepping stones. The same coding agent acts as both the task agent (to be evaluated) and the meta agent (to generate modifications). While this design enables compounding gains in coding, the instruction-generation mechanism that drives self-improvement is fixed and handcrafted. Consequently, recursive improvement depends on alignment between coding performance and self-modification ability. (Bottom) In the DGM-H, the task agent and meta agent are combined into a single modifiable program called a hyperagent. This design allows the meta agent itself to be autonomously improved. The system retains the open-ended exploration structure of the DGM while making the meta-level improvement mechanism editable. This enables metacognitive self-modification and supports self-referential improvement across any computable task.

We introduce hyperagents, self-referential agents that unify task execution and agent generation into a single modifiable program. A hyperagent can improve not only how it solves tasks but also how it generates future improvements. To enable sustained and accumulating progress, we instantiate hyperagents by building directly on the Darwin Gödel Machine (DGM) to form DGM-Hyperagents (DGM-H). The DGM provides an open-ended, population-based exploration process that maintains an archive of progressively improving agents, allowing successful variants to serve as stepping stones for future gains. DGM-H retains this open-ended evolutionary structure and extends it by making the entire meta-level modification mechanism editable (Figure 1). By allowing agents to modify not only how they solve tasks but also how they improve themselves, the DGM-H has the potential to open-endedly self-improve on any computable task.

Agents. This paper defines an *agent* as any computable program, optionally including calls to FMs, external tools, or learned components. Agents are not restricted to a particular representation (e.g., neural networks or prompts) and may include arbitrary algorithmic logic, memory, and control flow. A *task agent* is an agent instantiated to solve a set of tasks. Examples include generating code edits for a software repository (Gauthier, 2024; Jimenez et al., 2024), predicting acceptance decisions for research papers (Couto et al., 2024), and designing reward functions for robotics environments (Ma et al., 2024). Task agents are evaluated empirically

on the given task. A *meta agent* is an agent whose only task is to modify existing agents and generate new ones. Given access to the entire archive of previous agents and evaluations, a meta agent proposes changes intended to improve future performance (including potentially many generations later). Importantly, these changes may target not only task-solving logic but also the meta agent itself, enabling improvements to the procedures by which future modifications are generated.

Hyperagents. A hyperagent is a self-referential agent that integrates a task agent and a meta agent within a single editable program, enabling it to modify not only how it performs tasks but also how it generates future self-modifications. Unlike hierarchical systems with fixed meta-levels, in hyperagents the meta agent is part of the same editable program and can rewrite itself. As a result, a hyperagent can improve both (1) how it solves tasks and (2) how it generates future self-improvements. We use Python, which is Turing-complete (Turing et al., 1936), and since a hyperagent can edit any code, it has the potential to build any computable machine.

Metacognitive self-modification. In hyperagents, the agent's self-improvement mechanism is itself subject to modification. In addition to improving its performance on a given task, the agent can simultaneously modify the procedures by which it proposes and applies further self-improvements. We refer to this process as *metacognitive self-modification*, in which the hyperagent improves not only the task-performing agent responsible for solving the given task, but also the meta agent that determines how subsequent hyperagents are generated. This characteristic addresses a central limitation of prior self-improving systems (Zhang et al., 2025b; Wang et al., 2025b) by directly enabling improvements to the self-improvement process itself (Section 2). Examples of such metacognitive self-modifications are presented in Section 5.2 and Appendix E.3.

Darwin Gödel Machine with Hyperagents. Augmenting the original DGM (Zhang et al., 2025b) with hyperagents, we create DGM-Hyperagents (DGM-H). DGM-H employs the open-ended exploration process in the DGM to mitigate premature convergence and avoid getting trapped in local optima. This process maintains an archive of generated hyperagents, initialized with a single hyperagent and expanded over time by continuously accumulating generated variants. The process alternates between two phases: metacognitive self-modification and evaluation. During the metacognitive self-modification phase, selected parent hyperagents from the archive generate modified versions of themselves. Parent selection is probabilistic and proportional to a hyperagent's performance, and inversely proportional to the number of children that successfully compiled, biasing sampling toward hyperagents that perform well and generate strong descendants while preserving exploration (Appendix A.2). During the evaluation phase, each modified hyperagent is empirically evaluated and subsequently added to the archive. In principle, a fully self-referential algorithm should allow modification of every part of itself (including the parent selection and evaluation mechanisms). While we present preliminary results exploring the possibility of automatically improving the parent selection mechanism in Appendix E.5, the experiments in the main text use a handcrafted parent selection mechanism that is not subject to modification in order to isolate the effects of hyperagent self-modification. Overall, DGM-H consists of two interacting components: (1) an open-ended exploration process inherited from the DGM, and (2) an initial hyperagent, which evolves over time through self-generated variants (Figure 1, Appendix A). By extending the DGM to make the meta-level mechanism itself modifiable, the DGM-H generalizes recursive self-improvement beyond coding and enables self-referential improvement for any computable task.

4 Experiment Setup

The DGM-H is initialized with a single hyperagent built around a frozen FM (Brown et al., 2020) with external tool-use capabilities (Schick et al., 2023). The FM autonomously invokes tools as needed at each step. This initial hyperagent is equipped with two tools: a bash tool for executing shell commands, and a specialized tool for inspecting and modifying files. In the initial hyperagent, the task agent receives the task inputs and outputs the response from a single FM call. The meta agent receives the location of the agent's repository, the location of previous evaluation results, and the number of remaining experiment iterations (the experiment budget is a hyperparameter), and is tasked with modifying any part of the given codebase (Appendix A.1). We deliberately keep the implementation of the initial hyperagent (i.e., both the task agent and the meta agent) simple, allowing for a broad search space and avoiding prematurely constraining the range of possible improvements. Appendix D.1 details the FMs and the hyperparameters used in each experiment, while Appendix D.2 reports the corresponding cost estimates.

4.1 Baselines

We compare the Darwin Gödel Machine with Hyperagents (**DGM-H**) against a set of baselines designed to isolate the contributions of self-improving meta agents, open-ended exploration, and metacognitive self-modification (i.e., hyperagents) (Appendix B). DGM-H without self-improving meta agents (**DGM-H w/o self-improve**) fixes the meta agent modifying other hyperagents to the initial meta agent for the entire run. While the system can still generate and evaluate new hyperagents, the procedures used to propose and apply modifications are not themselves modifiable. This baseline replicates the approach of (ADAS, Hu et al., 2025) in this setting. It tests whether performance gains arise solely from task-level optimization under a fixed meta-level mechanism, or from improvements in the self-improvement process itself. DGM-H without open-ended exploration (**DGM-H w/o open-ended exploration**) removes the archive of previously generated hyperagents. Instead of retaining previous variants as stepping stones for future exploration, each newly generated hyperagent replaces its predecessor. After generating a new hyperagent, it is evaluated and automatically becomes the next selected parent. This baseline isolates the role of open-ended, population-based search. Comparing against this baseline tests whether sustained improvement requires accumulating and reusing diverse intermediate solutions.

We also compare against the original implementation of the Darwin Gödel Machine (**DGM**) (Zhang et al., 2025b). The DGM relies on a handcrafted, fixed mechanism to generate self-improvement instructions. Because this instruction-generation mechanism is designed specifically for coding benchmarks and is not modifiable, the DGM's self-improvement capacity in other domains is limited (Section 2). To provide a stronger comparison, we manually customize the DGM's instruction-generation mechanism for the target domains (**DGM-custom**) (Appendix B). This baseline measures how much the DGM relies on human engineering to remain competitive across domains. Comparing the DGM-H against this baseline tests whether automated metacognitive self-modification can outperform human-designed self-improvement mechanisms. Additionally, we compare against static solutions that have been handcrafted for each domain in prior work.

4.2 Domains

We evaluate our method and baselines across diverse domains (i.e., coding, paper review, robotics reward design, and Olympiad-level math grading) (Appendix C). To reduce computational cost, for each domain we first evaluate agents on a small subset of the training tasks to estimate overall effectiveness. Only agents that demonstrate sufficient performance are subsequently evaluated on the remaining training tasks. Agents that do not are treated as having zero performance on unevaluated tasks. Domain-specific evaluation protocols are described in detail in the subsequent paragraphs. For domains where we create AI judges to reflect human data (i.e., paper review and Olympiad-level math grading), we construct a validation subset because the AI judges are more likely to overfit to the training data. When a validation subset is defined for a domain, the performance component used in parent selection is measured on the validation set. Otherwise, it is measured on the training set. Each domain includes separate held-out test tasks that are used only for final evaluation.

Coding. We choose Polyglot (Gauthier, 2024) as a computationally cost-efficient coding benchmark for direct comparison with prior work (Zhang et al., 2025b). In this benchmark, the agent is given a code repository and a natural language instruction describing a desired change, and must modify the repository accordingly. We follow the experimental setup used in the DGM (Zhang et al., 2025b), including the same training and test splits, no validation set, and the same staged evaluation protocol (i.e., first evaluating each agent on 10 tasks to estimate effectiveness before expanding to 50 additional tasks) (Appendix C.1).

Paper review. This domain evaluates agents on a simulated conference peer review task. For each task, the agent is given the full text of an AI research paper and must predict a binary accept/reject decision. We include paper review to evaluate the DGM-H in a hard-to-verify setting where there is no objective ground truth. Peer review is subjective, and reviewer decisions can vary due to differing priorities and perspectives. We do not aim to change the peer review system, but rather, we study whether hyperagents can automatically learn decision procedures that align with observed human judgments. The agent outputs a single acceptance decision, and performance is measured by comparing predictions against observed acceptance outcomes. The dataset is drawn from Zhao et al. (2026), which constructs a large-scale benchmark from publicly available submissions and acceptance decisions from recent top-tier machine learning conferences. The representative static baseline for this domain is the reviewer agent from the AI-Scientist-v2 (Yamada

et al., 2025). Appendix C.2 provides full details on the dataset splits (train, validation, and test), the staged evaluation protocol (i.e., first evaluating each agent on a 10-task subset to estimate effectiveness before expanding evaluation to a total of 100 tasks), and the representative baselines for this domain.

Robotics reward design. This domain evaluates an agent's ability to design reward functions for robotic tasks. We include this domain to move beyond language-only tasks and show that hyperagents can leverage external simulators (e.g., physics engines) and training algorithms (e.g., reinforcement learning (RL)) to produce effective solutions. Given a natural language description of a robotics task, an agent must generate a suitable reward function. This reward function is then used to train a quadruped robot in simulation using RL (Genesis, 2024). The quality of the agent's solution is measured by the performance of the resulting policy: after training with the generated reward function, we evaluate how well the robot achieves the desired behavior (Ma et al., 2024). We use separate training and test tasks. During training, agents are required to generate reward functions that enable the robot to walk forward. For held-out testing, agents must zero-shot generate new reward functions that maximize the robot's torso height. Because reward functions that successfully enable a robot to walk forward do not induce jumping behaviors (the more optimal behavior for maximizing the robot's torso height), this setup evaluates whether a single agent can design suitable reward functions for different robotics tasks. This domain does not have a separate validation task. Appendix C.3 provides full details on the staged evaluation protocol (i.e., first evaluating each agent on 3 repetitions of the training task to estimate effectiveness before expanding evaluation to a total of 6 repetitions), and the representative baselines for this domain.

Olympiad-level math grading. This domain evaluates an agent's ability to grade solutions to Olympiad-level math problems. This domain is reserved as a held-out meta-evaluation to test whether DGM-H's improvements to its self-improvement process transfer across domains and continue to compound over time. We use IMO-GradingBench (Luong et al., 2025), which consists of International Mathematical Olympiad (IMO)-level problems paired with candidate solutions and expert human grades. For each task, the agent is given an IMO-level problem, a candidate solution, reference solutions, and grading guidelines to predict a discrete score. Performance is measured by the accuracy of the agent's grades with respect to expert human grades. The representative static baseline for this domain is the ProofAutoGrader from IMO-GradingBench. Appendix C.4 provides full details on the score labels, dataset splits (train, validation, and test), the staged evaluation protocol (i.e., first evaluating the agent on a 10-task subset to estimate effectiveness before expanding evaluation to a total of 100 tasks), and the baselines for this domain.

5 Results

For each experiment, we run each method 5 times. We report medians with 95% bootstrap confidence intervals computed from 1,000 resamples, using the notation *median* (CI: *lower* – *upper*). In line plots, lines show median performance and shaded regions indicate the confidence intervals (Figures 2 to 4). Bar plots report median performance on held-out test sets, with error bars indicating confidence intervals (Figures 2 to 4). Statistical significance is assessed using the Wilcoxon signed-rank test. Overall, the DGM-H exhibits general self-improvement at both the task and meta levels. Improvements to the task agent transfer to held-out test tasks within each domain, exceeding open-sourced static baselines (Section 5.1). Meta-level improvements transfer across domains, enabling hyperagents to significantly improve their ability to generate better task agents in previously unseen domains (Section 5.2). Self-improvements learned in one DGM-H run can potentially accelerate learning in subsequent runs and continue to compound as further self-modifications are applied (Section 5.3). All experiment logs are open-sourced in our codebase.

5.1 Improving Task Performance

The DGM-H can achieve self-improvement in coding comparable to prior self-improving algorithms. On the Polyglot coding benchmark, we use the same experimental settings as in the DGM (e.g., identical FM parameters, same number of 80 iterations) to enable a direct comparison. Across 5 runs, the DGM-H improves its training performance on the 50-task Polyglot subset from 0.140 (the initial agent) to 0.340 (CI: 0.300 – 0.380). When evaluated on the full Polyglot benchmark, which consists largely of tasks unseen during training, performance increases from 0.084 (the initial agent) to 0.267 (CI: 0.231 – 0.280). These improvements are

comparable to those reported for the original DGM, which improves from 0.140 to 0.380 on the training subset and from 0.142 to 0.307 on the full benchmark (Zhang et al., 2025b). Overall, these results show that the DGM-H can effectively self-improve in the coding domain and achieve a similar level of improvement to the original DGM, despite not being handcrafted specifically for coding tasks.

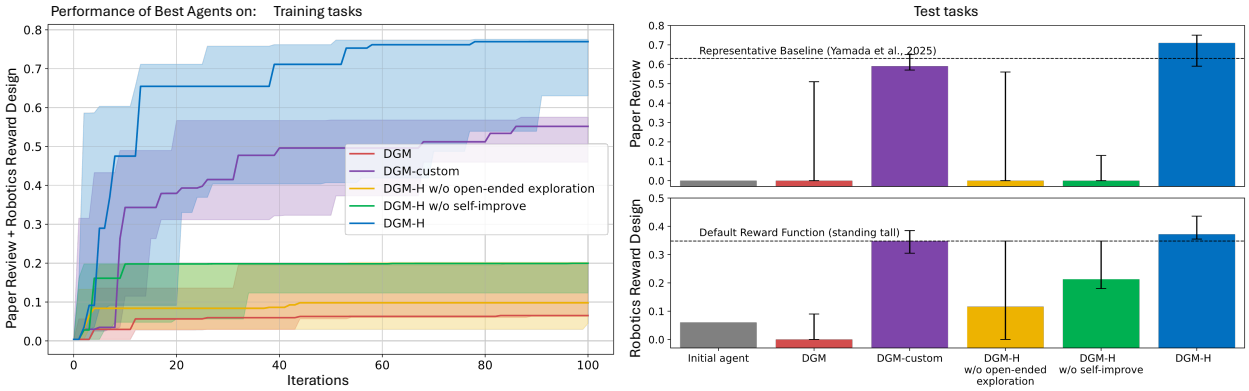

Figure 2 Metacognitive self-modification and open-ended exploration enable the DGM-H to continue making progress and improve its performance. (Left) The DGM-H can optimize for diverse tasks within the same run and automatically discovers increasingly better task agents. (Right) The best discovered task agents, selected based on validation or training scores, are evaluated on test tasks in (Top-Right) paper review and (Bottom-Right) robotics reward design. The DGM-H outperforms baselines that lack metacognition (i.e., the original DGM), self-improvement, or open-ended exploration, showing that all components are essential for continual self-improvement.

Beyond coding, we evaluate the DGM-H on paper review and robotics reward design to assess its ability to self-improve across diverse tasks within a single run (Appendix A.4). If an agent fails the staged evaluation in any domain, it is not evaluated on the full training set for any domain, and a score of zero is assigned for all remaining tasks. For example, when jointly optimizing the paper review and robotics reward design domains within the same experiment run, failure in the staged evaluation of either domain (e.g., correctly predicting none of the paper reviews in a smaller training subset, or failing to generate any compilable reward function) results in the agent not being evaluated on the full training set for either domain. Only the best agents, selected via validation scores (or training scores when validation tasks do not exist), are evaluated on the test set. All runs start from the same initial agent, which directly outputs the response from a single FM call (Appendix A.1). This initial agent achieves a test-set score of 0.0 on paper review and 0.060 on robotics reward design (Figure 2). In paper review, the initial agent fails to produce outputs in the required format because no task-specific parsing or post-processing is provided, resulting in a score of 0.0. In robotics reward design, the initial agent typically generates reward functions that either fail to compile (e.g., due to invalid environment variables) or do not yield effective policies. However, the initial agent occasionally produces a minimally functional reward, leading to a small but non-zero average score. Starting from this initial agent, each method is run for 100 iterations.

The original DGM requires domain-specific customization to generalize beyond coding. When applied in the form designed specifically to optimize coding agents, the original DGM achieves test-set performance of 0.0 (CI: 0.0 – 0.510) on paper review and 0.0 (CI: 0.0 – 0.090) on robotics reward design. This outcome is expected, as the original DGM's instruction-generation mechanism is explicitly tailored to coding tasks and to improving coding agents (Appendix B). However, when manually customized for each new domain, DGM-custom achieves substantially stronger results: 0.590 (CI: 0.570 – 0.650) on paper review and 0.348 (CI: 0.305 – 0.385) on robotics reward design. DGM-custom significantly outperforms the original DGM in both domains ($p < 0.05$) (Figure 2). These results highlight both the strength of the DGM within its intended domain and the importance of domain-specific manual customization for extending its self-improvement capabilities beyond coding in its original implementation.

Metacognitive self-modification enables sustained task improvement across domains. The DGM-H substantially improves task performance over time in both domains, demonstrating effective improvement of the task agent. Importantly, these gains transfer to held-out test sets, indicating that improvements reflect

general competence rather than overfitting to training tasks. On test tasks, DGM-H improves paper review performance from 0.0 (the initial agent) to 0.710 (CI: 0.590 – 0.750) (Appendix E.1.1). Several top-performing agents outperform the open-sourced static baseline, which achieves 0.630 (Yamada et al., 2025). On robotics reward design, DGM-H improves performance from 0.060 (the initial agent) to 0.372 (CI: 0.355 – 0.436) (Appendix E.1.2), surpassing the default reward function that directly optimizes the evaluation metric (0.348).

Qualitatively, the DGM-H improves task agents by moving beyond surface-level prompt tweaks toward structured, reusable decision machinery. In paper review, it shifts from superficial behavioral instructions (e.g., adopting a "rigorous" persona) to explicit multi-stage evaluation pipelines with checklists, decision rules, and clearly defined criteria, resulting in more consistent and higher-quality judgments (Appendix E.2). In robotics reward design, the DGM-H incrementally builds and refines an internal knowledge base of environment constraints, valid state variables, and reward-scaling heuristics, eliminating compilation failures and reducing reward misspecification. The DGM-H accumulates and refines domain knowledge (e.g., environment documentation, grading criteria) and integrates it into increasingly sophisticated decision frameworks, enabling complex, consistent task behavior without manual, domain-specific engineering (Appendix E.2).

The DGM-H significantly outperforms the original DGM in both domains ($p < 0.05$), reflecting the limitations of the original DGM's fixed instruction-generation mechanism when applied to non-coding tasks (Figure 2). The DGM-H achieves higher median performance than DGM-custom on the test tasks in both domains, although the difference is not statistically significant ($p > 0.05$) (Figure 2). Notably, in the robotics reward design test task (i.e., design a reward function that trains the robot to maximize its torso height), the DGM-H almost always generates task agents that design reward functions inducing jumping behaviors, rather than getting stuck in the local optimum of standing tall. In contrast, DGM-custom often generates task agents that fail to escape this local optimum. This qualitative difference suggests that metacognitive self-modification can match or exceed handcrafted, domain-specific mechanisms.

Both metacognitive self-modification and open-ended exploration are necessary for sustained improvement. DGM-H w/o self-improve (Hu et al., 2025) shows little to no improvement, achieving test-set performance of 0.0 (CI: 0.0 – 0.130) on paper review and 0.213 (CI: 0.180 – 0.348) on robotics reward design. The DGM-H significantly outperforms this baseline in both domains ($p < 0.05$) (Figure 2), demonstrating that the observed gains in the DGM-H are not solely due to task-level optimization under a fixed meta agent, but also arise from improvements in the self-modification process itself (i.e., the meta agent). Similarly, DGM-H w/o open-ended exploration fails to achieve meaningful improvement, with test-set performance remaining at 0.0 (CI: 0.0 – 0.560) on paper review and 0.116 (CI: 0.0 – 0.348) on robotics reward design. The DGM-H significantly outperforms this baseline in both domains ($p < 0.05$) (Figure 2), highlighting the importance of open-ended, population-based search to create stepping stones for future improvements.

5.2 Improving the Ability to Improve

DGM-H's superior performance to DGM-H w/o self-improve shows that DGM-H improves the meta agent (i.e., its ability to self-modify and generate new agents) (Section 5.1). This section investigates whether these meta-level improvements are general rather than domain-specific. Specifically, we evaluate whether self-improvement strategies learned in one setting transfer to and accelerate learning in a different domain.

Improvement@k metric. To quantify a meta agent's ability to generate improved task agents, we introduce the improvement@k (imp@k) metric. Given an initial meta agent M, an initial task agent A, an agent-generation algorithm (e.g., DGM or DGM-H variants), and a fixed task, M is allowed to generate up to k new task agents from A and its descendants using the specified algorithm. We define imp@k as the improvement in test performance from the initial task agent A to the best-performing generated task agent (selected based on validation score) (Appendix D.3). Intuitively, imp@k measures how effectively a meta agent can produce improved variants within a limited number of iterations. To isolate the meta agent's ability to improve task agents from its ability to improve itself, we hold the meta agent fixed throughout the process (i.e., the agent responsible for generating new task agents is always the initial meta agent M), yielding DGM w/o self-improve and DGM-H w/o self-improve as the agent-generation algorithms. All experiments in this section run for 50 iterations, measuring imp@50.

The initial meta agent has little to no ability to generate improved task agents. Taking the same initial

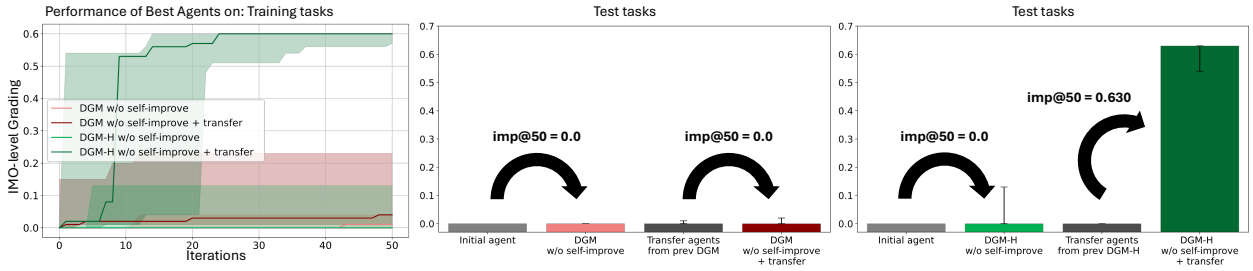

Figure 3 Self-improvement strategies learned by the DGM-H in one setting transfer to and accelerate learning in a different setting. We measure an agent's ability to generate improved agents using imp@50, which takes as input a starting agent, an agent-generation algorithm, and an evaluation task. The algorithm is run for 50 iterations starting from the given agent, and imp@50 is defined as the performance gain of the best generated agent over the starting agent on the task. (Left, DGM w/o self-improve and DGM-H w/o self-improve) On Olympiad-level math grading, starting from the initial agent, both DGM w/o self-improve and DGM-H w/o self-improve achieve little to no improvement, showing that the initial agent has limited ability to generate better agents. (Left, DGM w/o self-improve + transfer) Starting from transfer agents, DGM w/o self-improve also achieves little improvement, showing that the original DGM does not learn transferable meta-level improvements. (Left, DGM-H w/o self-improve + transfer) In contrast, starting from transfer hyperagents, DGM-H w/o self-improve achieves substantial improvement, showing that hyperagents enable DGM-H to acquire transferable self-improvement strategies. (Middle) Regardless of starting from an initial agent or a transfer agent, DGM w/o self-improve yields imp@50 ≈ 0, showing that the original DGM does not improve the meta-level skill of generating improved agents. (Right) Starting from a transfer hyperagent, DGM-H w/o self-improve achieve large positive imp@50, showing that that the self-improvement strategies learned by the DGM-H are general and transferable, and that the DGM-H improves its ability to improve.

meta and task agents as in the above experiments (Section 4), we evaluate the initial meta agent's ability to generate improved task agents on Olympiad-level math grading. The initial task agent achieves a test-set score of 0.0, as it fails to produce outputs in the required format due to the lack of task-specific parsing or post-processing. After 50 iterations of DGM w/o self-improve, during which the initial meta agent is held fixed and used to generate new task agents starting from a initial task agent, the best generated task agents (across 5 repeated runs) achieve a median test score of 0.0 (CI: 0.0 – 0.0). Accordingly, imp@50 for the initial meta agent under DGM w/o self-improve is 0.0 (CI: 0.0 – 0.0) on Olympiad-level math grading. While DGM w/o self-improve relies on a fixed handcrafted instruction-generation mechanism, DGM-H w/o self-improve does not. DGM w/o self-improve uses a handcrafted instruction-generation mechanism to specify which parts of the agent's codebase should be modified, whereas DGM-H w/o self-improve allows the meta agent to autonomously decide which parts of the codebase to modify. After 50 iterations of DGM-H w/o self-improve, again holding the initial meta agent fixed and generating task agents from the same initial task agent, the best generated task agents (across 5 repeated runs) achieve a median score of 0.0 (CI: 0.0 – 0.130). Accordingly, imp@50 for the initial meta agent under DGM-H w/o self-improve is 0.0 (CI: 0.0 – 0.130) (Figure 3). These results show that, on Olympiad-level math grading, the initial meta agent has little to no ability to generate improved task agents, regardless of the agent-generation algorithm used (e.g., DGM or DGM-H).

Meta agents optimized in prior DGM-custom runs have little to no ability to generate improved task agents when transferred to a new domain. We evaluate whether self-improvements learned in one setting using the DGM can transfer to new domains. We consider agents optimized in prior DGM-custom runs on the paper review and robotics reward design tasks (Section 5.1). From each run, we select a transfer agent based on the maximum performance gain achieved by its descendants, measured relative to the agent itself and discounted by lineage depth (Appendix D.4). We apply discounting to favor agents that reliably produce strong improvements within fewer self-modification steps, since improvements occurring many iterations later are less directly attributable to the original agent. We choose transfer agents from DGM-custom rather than from DGM because DGM-custom achieved larger improvements in the previous settings (Section 5.1). For transfer, we carry over the entire agent implementation from the source domain, including both the meta agent and the task agent. Concretely, the transfer meta agent is used as the fixed agent responsible for generating new task agents in the target domain, and the transfer task agent serves as the initial task agent from which new variants are generated. We evaluate whether the transfer meta agents can generate improved task agents

in a new domain, Olympiad-level math grading. The transfer task agents (one from each of 5 repeated runs) achieve a test-set score of 0.0 (CI: 0.0 – 0.010). As expected, this performance is low because the agents were optimized for paper review and robotics reward design, which differ substantially from the math grading task. After 50 iterations of DGM w/o self-improve starting from each transfer task agent and holding the transfer meta agent fixed, the best generated task agents (selected based on validation score) achieve a score of 0.0 (CI: 0.0 – 0.020). This corresponds to a imp@50 of 0.0 (CI: 0.0 – 0.010) (Figure 3), showing that the transfer meta agents from DGM-custom runs cannot generate improved task agents in a previously unseen domain. When using DGM w/o self-improve as the agent-generation algorithm, imp@50 for the transfer agents is similar to that of the initial agent (both close to zero). This shows that the task performance improvements observed with DGM-custom rely on domain-specific customizations, rather than on general improvements in the meta ability to generate improved task agents.

Hyperagents optimized in prior DGM-H runs demonstrate transferable ability to generate improved task agents in a new domain. We repeat the transfer experiments for the DGM-H to assess whether hyperagents enable the learning of general self-improvement mechanisms. From prior DGM-H runs on paper review and robotics reward design (Section 5.1), we select one transfer hyperagent per run using the same lineage-discounted criterion described above (Appendix D.4). The transfer hyperagents (one from each of 5 repeated runs) achieve an initial test-set score of 0.0 (CI: 0.0 – 0.0), reflecting the domain shift. After 50 iterations of DGM-H w/o self-improve starting from the transfer hyperagents, the best generated agents (selected based on validation score) achieve a test-set score of 0.630 (CI: 0.540 – 0.630). This corresponds to a imp@50 of 0.630 (CI: 0.540 – 0.630) (Figure 3). These results show that transfer hyperagents can generate improved agents in a previously unseen domain. When using DGM-H w/o self-improve as the agent-generation algorithm, imp@50 for the transfer agents is significantly higher than imp@50 for the initial agent ($p < 0.05$). This indicates that the transfer agents are substantially more effective at generating improved agents, and that the meta-improvements learned through DGM-H in one run are general and transferable, accelerating learning in a different domain.

We qualitatively attribute the observed transfer gains to a set of general-purpose meta-level capabilities that the DGM-H autonomously acquires during prior runs. In particular, the transfer hyperagents have features such as performance tracking and persistent memory, which allow them to reason about improvement as an ongoing process rather than as isolated code edits (Appendix E.3). As a result, even when transferred to an unseen domain, these hyperagents can quickly self-improve and make meaningful progress (Figure 3). This contrasts with DGM transfer agents, whose gains rely on domain-specific customizations and do not improve the underlying agent-generation process itself. These qualitative observations show that the DGM-H learns how to improve, yielding general and transferable self-improvement capability.

One example is DGM-H's autonomous development of performance tracking infrastructure. Rather than relying on isolated evaluation outcomes, the hyperagent records, aggregates, and compares performance metrics across iterations, identifying which changes yield sustained improvements and which lead to regressions. This performance tracking supports informed decision-making, such as prioritizing promising iterations and avoiding unproductive modification directions. The snippet below shows an automatically introduced performance tracker that logs and organizes metrics across iterations:

```
class PerformanceTracker:
    """Tracks performance metrics across agent generations."""

    def __init__(self, tracking_file: str = "./outputs/performance_history.json"):
        self.tracking_file = tracking_file
        self.history = self._load_history()

    def record_generation(self, generation_id: int, domain: str,
                          score: float, metadata: dict = None):
        """Record performance for a generation."""
        entry = {
            "generation_id": generation_id,
            "domain": domain,
            "score": score,
            "timestamp": datetime.now().isoformat(),
            "metadata": metadata or {}
        }
        self.history.append(entry)
        self._save_history()

    def get_improvement_trend(self, domain: str = None, window: int = 5):
```

```
        """Calculate improvement trend using moving average."""
        filtered = self.history
        if domain:
            filtered = [h for h in self.history if h.get('domain') == domain]

        if len(filtered) < window * 2:
            return None

        recent_avg = sum(h['score'] for h in filtered[-window:]) / window
        older_avg = sum(h['score'] for h in filtered[-window*2:-window]) / window

        return recent_avg - older_avg # Positive if improving

    def get_statistics(self, domain: str = None):
        """Get comprehensive statistics."""
        scores = [h['score'] for h in filtered]
        return {
            "total_generations": len(filtered),
            "best_score": max(scores),
            "worst_score": min(scores),
            "average_score": sum(scores) / len(scores),
            "improvement_trend": self.get_improvement_trend(domain)
        }
```

Another example is DGM-H's innovation of persistent memory, which enables learning to accumulate across iterations. Instead of merely logging numerical scores, the hyperagent stores synthesized insights, causal hypotheses, and forward-looking plans (e.g., identifying which generations performed best, diagnosing over-corrections, and proposing how to combine successful strategies). This memory is actively consulted during subsequent self-modification steps, allowing later generations to build on earlier discoveries and avoid repeating past mistakes. This is an example of a stored memory entry:


```
{
  "best_performers_analysis": {
    "value": "Best Performing Generations:\n\nPaper Review:\n
      - Gen55: 63% acc, 25% accept rate, 38% accept recall, 88% reject recall (too harsh)\n
      - Gen64: 61% acc, 53% accept rate, 64% accept recall, 58% reject recall (BEST BALANCE)\n\n
      Genesis Go2Walking:\n- Gen62: 0.802 fitness (BEST)\n- Gen63: 0.798 fitness\n\n
      Key Insight: Gen55 has best accuracy but is too harsh. Gen64 improved balance
      but lower accuracy. Need to combine gen55's critical reasoning with gen64's balance.",
    "timestamp": "2026-01-07T00:43:27.550275"
  },
  "gen66_results": {
    "value": "Gen 66 Results (Latest): Paper Review - 52% accuracy (WORSE than gen64's 61%),
      94% accept rate (EXTREME accept bias). CRITICAL ISSUE: Paper review went backwards
      - gen65 changes over-corrected.",
    "timestamp": "2026-01-07T01:03:41.176643"
  },
  "iteration_66_improvement": {
    "value": "Iteration 66 Improvement: CORRECTED paper review prompt after gen65 over-correction.
      New approach: (1) Clear REJECT criteria with specific triggers, (2) Decision Protocol
      with explicit thresholds (~40%/~60%), (3) Borderline guidance defaults to REJECT.
      Target: 62-64% accuracy with 55-65% reject rate.",
    "timestamp": "2026-01-07T01:06:35.559820"
  }
}
```


5.3 Compounding Self-Improvements

We investigate whether self-improvements learned by DGM-H in one setting continue to accumulate when DGM-H is run in a different setting. From prior DGM-H runs on the paper review and robotics reward design tasks (Section 5.1), we select transfer hyperagents using the same selection mechanism described earlier (Section 5.2, Appendix D.4). We then evaluate their ability to continue self-improving in a new domain, Olympiad-level math grading. After 200 iterations of DGM-H starting from these transfer agents (DGM-H + transfer), the best generated agents (selected based on validation score) achieve a test-set score of 0.640 (CI: 0.550 – 0.720). Under the same experimental setup, DGM-H starting from the initial agent achieves a best test-set score of 0.610 (CI: 0.510 – 0.680). Although the difference between DGM-H + transfer and DGM-H is not statistically significant ($p > 0.05$), DGM-H + transfer achieves a higher median performance and higher confidence intervals than DGM-H starting from the initial agent (Figure 4). Notably, improvements at higher performance levels are increasingly difficult due to saturation effects (e.g., increasing performance from 0.7 to 0.8 is typically more challenging than from 0.0 to 0.1), making these gains meaningful despite their modest

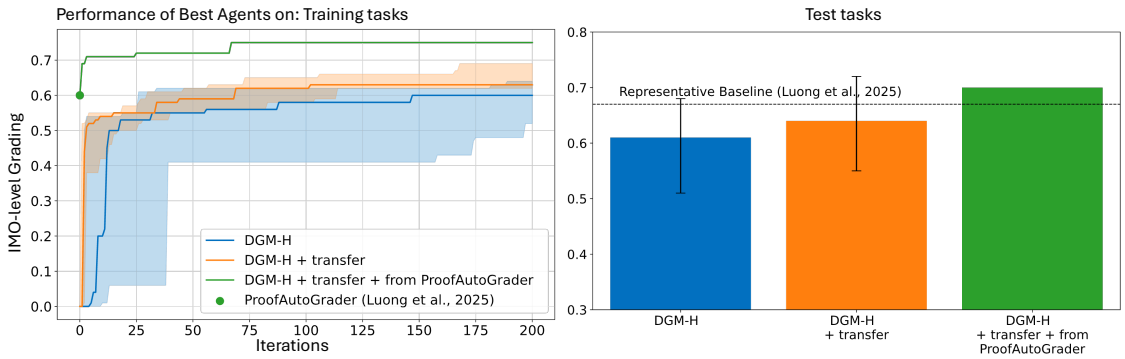

Figure 4 Self-improvements learned by the DGM-H accumulate across domains and runs. We continue running DGM-H on Olympiad-level math grading, starting from transfer hyperagents obtained in prior DGM-H runs, and compare this against DGM-H initialized from the initial agent and from ProofAutoGrader. (Left) Initializing from transferred hyperagents leads to faster progress and higher final performance than initializing from the initial agent, indicating that previously learned self-improvements remain useful and continue to compound in a new domain. (Right) DGM-H initialized from a transferred agent and ProofAutoGrader achieves the highest test performance, surpassing the representative baseline.

absolute magnitude. These results suggest that DGM-H's self-improvements are reusable and can potentially accumulate across runs, supporting the possibility of compounding self-improvement over time.

The representative static baseline for Olympiad-level math grading from IMO-GradingBench is ProofAutoGrader (Luong et al., 2025). We initialize the DGM-H with ProofAutoGrader as the task agent and a transfer meta agent obtained from a prior DGM-H run (on paper review and robotics reward design), and then continue optimizing for Olympiad-level math grading. After 200 iterations, the best discovered agent achieves a test-set score of 0.700, outperforming ProofAutoGrader's score of 0.670 (Figure 4). We then evaluate both the best discovered agent and ProofAutoGrader on the full IMO-GradingBench to obtain a more accurate estimate of the improvement. On the full IMO-GradingBench, the DGM-H improves ProofAutoGrader's accuracy from 0.561 to 0.601, and lowers the mean absolute error from 0.178 to 0.175 (Appendix E.4). We open-source this artifact to support future research and development (Appendix E.1.3). These results show that the DGM-H can build on strong existing solutions and further improve their performance.

6 Safety Discussion

The DGM-Hyperagents (DGM-H) introduces distinct safety considerations due to its ability to autonomously modify its own behavior and improvement mechanisms over time. In this work, all experiments are conducted under strict safety constraints. In particular, agent-generated code is executed within carefully sandboxed environments with enforced resource limits (e.g., timeouts, restricted internet access). These measures are designed to prevent unintended side effects, contain failures, and ensure that self-modifications remain confined to the intended experimental scope. Moreover, evaluation is performed using predefined tasks and metrics, and human oversight is maintained throughout all experiments.

Potential to evolve faster than human oversight. As AI systems gain the ability to modify themselves in increasingly open-ended ways, they can potentially evolve far more rapidly than humans can audit or interpret. At the cusp of such explosive capability growth, it becomes necessary to reconsider the roles that AI systems play in society (Bengio et al., 2024). Rather than framing safety solely in terms of absolute guarantees or full interpretability, a central challenge lies in balancing the potential of AI as a catalyst for human progress and well-being (e.g., automating scientific discovery) with the degree of trust humans are willing to place in these systems (e.g., delegating decisions or actions without requiring continuous human verification), while minimizing the many potential risks and downsides (Clune, 2019; Ecoffet et al., 2020; Bengio et al., 2024; Weston and Foerster, 2025). This balance is shaped by factors such as transparency and controllability.

While the DGM-H operates within safe research boundaries (e.g., sandboxing, controlled evaluations), these

safeguards may become increasingly strained or infeasible as self-improving systems grow more capable. We discuss additional safety considerations in Appendix F. We proactively include this discussion to encourage broader engagement with what safety means for open-ended self-improving AI systems (Clune, 2019; Ecoffet et al., 2020; Sheth et al., 2025). This includes ongoing discussion about appropriate levels of trust, oversight, and transparency, and societal deliberation about which benefits these systems should prioritize when deployed.

7 Limitations and Conclusion

This work introduces hyperagents and incorporates them into the Darwin Gödel Machine (DGM) to form DGM-Hyperagents (DGM-H). DGM-H is a general self-improvement framework that open-endedly evolves an archive of self-improving hyperagents for any computable task, enabling the system to improve both task performance and its own self-improvement mechanism. Across diverse domains, the DGM-H produced substantial and generalizable gains in task performance while also improving its ability to generate improvements, with these meta-level gains transferring across domains and compounding across runs.

Our results suggest that self-improvements can compound across different experimental settings, but this version of DGM-H has limitations that constrain truly unbounded progress. First, it operates with a fixed task distribution. One direction is to co-evolve the task distribution by generating new tasks and curricula that adapt to the agent's capabilities (Clune, 2019; Zhang et al., 2024; Faldor et al., 2025; Bolton et al., 2025). Second, components of the open-ended exploration loop (e.g., parent selection, evaluation protocols) remain fixed. Although hyperagents can modify their self-improvement mechanisms, they cannot alter the outer process that determines which agents are selected or how they are evaluated. Keeping these components fixed improves experimental stability and safety, but limits full self-modifiability. Enabling hyperagents to modify these outer-loop components and adapt their own search strategy and evaluation process is another promising direction for future work. Our preliminary results suggest such extensions are feasible (Appendix E.5).

DGM-H demonstrate that open-ended self-improvement can be made practical across diverse domains. Provided sufficient safety considerations are worked out, the DGM-H suggest a path toward self-accelerating systems that not only search for better solutions, but continually improve their ability to self-improve.

Acknowledgments

We thank Andrew Budker and Ricardo Silveira Cabral for supporting this work, and Alisia Lupidi, Chenxi Whitehouse, John Quan, Lisa Alazraki, Lovish Madaan, Lucia Cipolina-Kun, Mattia Opper, Michael Dennis, Parth Pathak, Rishi Hazra, Roberta Raileanu, Sandra Lefdal, Shashwat Goel, Shengran Hu, Timon Willi, Tim Rocktäschel, and Yoram Bachrach for insightful discussions and feedback.

Author Contributions

Jenny Zhang led the conceptualization of the study, conducted the experiments, and wrote the manuscript. Bingchen Zhao and Wannan Yang contributed to experimental design and execution. Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina provided feedback on the methodology and manuscript. All authors reviewed and approved the final manuscript.

References

Alexis Audran-Reiss, Jordi Armengol-EstapÃŠ, Karen Hambardzumyan, Amar Budhiraja, Martin Josifoski, Edan Toledo, Rishi Hazra, Despoina Magka, Michael Shvartsman, Parth Pathak, et al. What Does It Take to Be a Good AI Research Agent? Studying the Role of Ideation Diversity. *arXiv preprint arXiv:2511.15593*, 2025.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney. Learning to continually learn. *arXiv preprint arXiv:2002.09571*, 2020.

Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Trevor Darrell, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, et al. Managing extreme AI risks amid rapid progress. *Science*, 384(6698): 842–845, 2024.

Adrian Bolton, Alexander Lerchner, Alexandra Cordell, Alexandre Moufarek, Andrew Bolt, Andrew Lampinen, Anna Mitenkova, Arne Olav Hallingstad, Bojan Vujatovic, Bonnie Li, et al. Sima 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.

Herbie Bradley, Andrew Dai, Hannah Teufel, Jenny Zhang, Koen Oostermeijer, Marco Bellagente, Jeff Clune, Kenneth Stanley, Grégory Schott, and Joel Lehman. Quality-diversity through AI feedback. *arXiv preprint arXiv:2310.13032*, 2023.

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

Mathieu Chalvidal, Thomas Serre, and Rufin VanRullen. Meta-reinforcement learning with self-modifying networks. *Advances in Neural Information Processing Systems*, 35:7838–7851, 2022.

Yinzhu Chen, Abdine Maiga, Hossein A Rahmani, and Emine Yilmaz. Automated Rubrics for Reliable Evaluation of Medical Dialogue Systems. *arXiv preprint arXiv:2601.15161*, 2026.

Jeff Clune. AI-GAs: AI-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Lisa Coiffard, Paul Templier, and Antoine Cully. Overcoming Deceptiveness in Fitness Optimization with Unsupervised Quality-Diversity. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 122–130, 2025.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting autotelic agents with large language models. In *Conference on Lifelong Learning Agents*, pages 205–226. PMLR, 2023.

Jonathan Cook, Tim Rocktäschel, Jakob Foerster, Dennis Aumiller, and Alex Wang. Ticking all the boxes: Generated checklists improve llm evaluation and generation. *arXiv preprint arXiv:2410.03608*, 2024.

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

Paulo Henrique Couto, Quang Phuoc Ho, Nageeta Kumari, Benedictus Kent Rachmat, Thanh Gia Hieu Khuong, Ihsan Ullah, and Lisheng Sun-Hosoya. Relevai-reviewer: A benchmark on AI reviewers for survey paper relevance. *arXiv preprint arXiv:2406.10294*, 2024.

Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

Aaron Dharna, Cong Lu, and Jeff Clune. Foundation model self-play: Open-ended strategy innovation via foundation models. *arXiv preprint arXiv:2507.06466*, 2025.

Li Ding, Jenny Zhang, Jeff Clune, Lee Spector, and Joel Lehman. Quality Diversity through Human Feedback: Towards Open-Ended Diversity-Driven Optimization. In *Forty-first International Conference on Machine Learning*, 2024.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

Adrien Ecoffet, Jeff Clune, and Joel Lehman. Open questions in creating safe open-ended AI: Tensions between control and creativity. In *Artificial Life Conference Proceedings 32*, pages 27–35, 2020.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via Models of human Notions of Interestingness with Environments Programmed in Code. In *The Thirteenth International Conference on Learning Representations*, 2025.

Zhiyuan Fan, Weinong Wang, Debing Zhang, et al. Sedareval: Automated evaluation using self-adaptive rubrics. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 16916–16930, 2024.

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

Paul Gauthier. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, December 2024. Accessed: 2026-01-28.

Authors Genesis. Genesis: A generative and universal physics engine for robotics and beyond, December 2024. URL https://github.com/Genesis-Embodied-AI/Genesis.

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

Luca Grillotti, Lisa Coiffard, Oscar Pang, Maxence Faldor, and Antoine Cully. From Tabula Rasa to Emergent Abilities: Discovering Robot Skills via Real-World Unsupervised Quality-Diversity. *arXiv preprint arXiv:2508.19172*, 2025.

Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. Teaching large language models to reason with reinforcement learning. *arXiv preprint arXiv:2403.04642*, 2024.

Nathan Herr, Tim Rocktäschel, and Roberta Raileanu. LLM-First Search: Self-Guided Exploration of the Solution Space. *arXiv preprint arXiv:2506.05213*, 2025.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *The Thirteenth International Conference on Learning Representations*, 2025.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence. *arXiv preprint arXiv:2406.04268*, 2024.

Marcus Hutter. A gentle introduction to the universal algorithmic agent AIXI. *Artificial General Intelligence*, 2003.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. A modern self-referential weight matrix that learns to modify itself. In *International Conference on Machine Learning*, pages 9660–9677. PMLR, 2022.

Matthew Thomas Jackson, Chris Lu, Louis Kirsch, Robert Tjarko Lange, Shimon Whiteson, and Jakob Nicolaus Foerster. Discovering temporally-aware reinforcement learning algorithms. *arXiv preprint arXiv:2402.05828*, 2024.

Khurram Javed and Martha White. Meta-learning representations for continual learning. *Advances in neural information processing systems*, 32, 2019.

Minqi Jiang, Tim Rocktäschel, and Edward Grefenstette. General intelligence requires rethinking exploration. *Royal Society Open Science*, 10(6):230539, 2023.

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.

Louis Kirsch and Jürgen Schmidhuber. Eliminating meta optimization through self-referential meta learning. *arXiv preprint arXiv:2212.14392*, 2022.

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. *arXiv preprint arXiv:2310.00166*, 2023.

Martin Klissarov, Mikael Henaff, Roberta Raileanu, Shagun Sodhani, Pascal Vincent, Amy Zhang, Pierre-Luc Bacon, Doina Precup, Marlos C Machado, and Pierluca D'Oro. MaestroMotif: Skill Design from Artificial Intelligence Feedback. In *The Thirteenth International Conference on Learning Representations*, 2025.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499*, 2025.

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*, pages 29–30, 2023.

Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218, 2011.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pages 331–366. Springer, 2023.

Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. Auto mc-reward: Automated dense reward design with large language models for minecraft. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16426–16435, 2024.

Chris Lu, Sebastian Towers, and Jakob Foerster. Arbitrary order meta-learning with simple population-based evolution. In *Artificial Life Conference Proceedings 35*, volume 2023, page 67, 2023.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Lange. Discovering preference optimization algorithms with and for large language models. *Advances in Neural Information Processing Systems*, 37:86528–86573, 2024a.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024b.

Minh-Thang Luong, Dawsen Hwang, Hoang H Nguyen, Golnaz Ghiasi, Yuri Chervonyi, Insuk Seo, Junsu Kim, Garrett Bingham, Jonathan Lee, Swaroop Mishra, et al. Towards robust mathematical reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 35406–35430, 2025.

Changze Lv, Jie Zhou, Wentao Zhao, Jingwen Xu, Zisu Huang, Muzhao Tian, Shihan Dou, Tao Gui, Le Tian, Xiao Zhou, Xiaoqing Zheng, Xuanjing Huang, and Jie Zhou. Learning Query-Specific Rubrics from Human Preferences for DeepResearch Report Generation. *arXiv preprint arXiv:2602.03619*, 2026.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. In *The Twelfth International Conference on Learning Representations*, 2024.

Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR, 2018.

Thomas Miconi, Aditya Rawal, Jeff Clune, and Kenneth O Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. *arXiv preprint arXiv:2002.10585*, 2020.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Junhyuk Oh, Greg Farquhar, Iurii Kemaev, Dan A Calian, Matteo Hessel, Luisa Zintgraf, Satinder Singh, Hado Van Hasselt, and David Silver. Discovering state-of-the-art reinforcement learning algorithms. *Nature*, pages 1–2, 2025.

Julien Pourcel, Cédric Colas, Gaia Molinaro, Pierre-Yves Oudeyer, and Laetitia Teodorescu. ACES: Generating Diverse Programming Puzzles with with Autotelic Generative Models. *arXiv preprint arXiv:2310.10692*, 2023.

Xin Qiu, Yulu Gan, Conor F Hayes, Qiyao Liang, Elliot Meyerson, Babak Hodjat, and Risto Miikkulainen. Evolution strategies at scale: Llm fine-tuning beyond reinforcement learning. *arXiv preprint arXiv:2509.24372*, 2025.

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Mikayel Samvelyan, Sharath C Raparthy, Andrei Lupu, Eric Hambro, Aram H Markosyan, Manish Bhatt, Yuning Mao, Minqi Jiang, Jack Parker-Holder, Jakob Foerster, et al. Rainbow teaming: Open-ended generation of diverse adversarial prompts. *Advances in Neural Information Processing Systems*, 37:69747–69786, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Jürgen Schmidhuber. A neural network that embeds its own meta-levels. In *IEEE International Conference on Neural Networks*, pages 407–412. IEEE, 1993.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ivaxi Sheth, Jan Wehner, Sahar Abdelnabi, Ruta Binkyte, and Mario Fritz. Safety is Essential for Responsible Open-Ended Systems. *arXiv preprint arXiv:2502.04512*, 2025.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Kenneth O Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge you've never heard of. *While open-endedness could be a force for discovering intelligence, it could also be a component of AI itself*, 2017.

Marilyn Strathern. 'Improving ratings': audit in the British University system. *European review*, 5(3):305–321, 1997.

Alan Mathison Turing et al. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*, 2024.

Jianyu Wang, Zhiqiang Hu, and Lidong Bing. Evolving Prompts In-Context: An Open-ended, Self-replicating Perspective. *arXiv preprint arXiv:2506.17930*, 2025a.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-G\" odel Machine: Human-Level Coding Agent Development by an Approximation of the Optimal Self-Improving Machine. *arXiv preprint arXiv:2510.21614*, 2025b.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, et al. Evo-Memory: Benchmarking LLM Agent Test-time Learning with Self-Evolving Memory. *arXiv preprint arXiv:2511.20857*, 2025a.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward Training Superintelligent Software Agents through Self-Play SWE-RL. *arXiv preprint arXiv:2512.18552*, 2025b.

Jiaxin Wen, Zachary Ankner, Arushi Somani, Peter Hase, Samuel Marks, Jacob Goldman-Wetzler, Linda Petrini, Henry Sleight, Collin Burns, He He, et al. Unsupervised Elicitation of Language Models. *arXiv preprint arXiv:2506.10139*, 2025.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-Evolving Agents: Open-Ended Self-Improvement via Experience Sharing. *arXiv preprint arXiv:2602.04837*, 2026.

Jason Weston and Jakob Foerster. Ai & human co-improvement for safer co-superintelligence. *arXiv preprint arXiv:2512.05356*, 2025.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? *arXiv preprint arXiv:2511.13646*, 2025a.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043*, 2025b.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. SkillRL: Evolving Agents via Recursive Skill-Augmented Reinforcement Learning. *arXiv preprint arXiv:2602.08234*, 2026.

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to Continually Learn via Meta-learning Agentic Memory Designs. *arXiv preprint arXiv:2602.07755*, 2026.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta Context Engineering via Agentic Skill Evolution. *arXiv preprint arXiv:2601.21557*, 2026.

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

Jiayi Yuan, Jonathan Nöther, Natasha Jaques, and Goran Radanović. AgenticRed: Optimizing Agentic Systems for Automated Red-teaming. *arXiv preprint arXiv:2601.13518*, 2026.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive Language Models. *arXiv preprint arXiv:2512.24601*, 2025a.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and Evolving Memory Skills for Self-Evolving Agents. *arXiv preprint arXiv:2602.02474*, 2026.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. OMNI: Open-endedness via Models of human Notions of Interestingness. In *The Twelfth International Conference on Learning Representations*, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents. *arXiv preprint arXiv:2505.22954*, 2025b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025c.

Bingchen Zhao, Despoina Magka, Minqi Jiang, Xian Li, Roberta Raileanu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Kelvin Niu, Shagun Sodhani, Michael Shvartsman, et al. The Automated LLM Speedrunning Benchmark: Reproducing NanoGPT Improvements. *arXiv preprint arXiv:2506.22419*, 2025.

Bingchen Zhao, Jenny Zhang, Chenxi Whitehouse, Minqi Jiang, Michael Shvartsman, Abhishek Charnalia, Despoina Magka, Tatiana Shavrina, Derek Dunfield, Oisin Mac Aodha, and Yoram Bachrach. APRES: An Agentic Paper Revision and Evaluation System. *arXiv preprint arXiv:2603.03142*, 2026.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

Barret Zoph and Quoc Le. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations*, 2017.

Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-Adapting Language Models. *arXiv preprint arXiv:2506.10943*, 2025.

Appendix

Table of Contents

A Algorithmic details

This appendix provides additional algorithmic details for the DGM-Hyperagents (DGM-H). We first describe the implementation of the initial hyperagent, including the tools and prompts available to the initial task and meta agents (Appendix A.1). We then detail the parent selection mechanism used during open-ended exploration, which balances exploitation of high-performing agents with continued exploration of the archive (Appendix A.2). Finally, we present pseudocode for DGM-H (Appendix A.3).

A.1 Initial Agent

We present the details of the tools available to the initial hyperagent and its prompts (Section 4).

Initial task agent prompt:

```
instruction = f"""You are an agent.
Task input:
```
{inputs}
```
Respond in JSON format with the following schema:
<json>
{{
    "response": ...
}}
</json>"""
```

Initial meta agent prompt:

```
instruction = f"Modify any part of the codebase at `{repo_path}`."
```

Information of the given bash tool:

```
def tool_info():
    return {
        "name": "bash",
        "description": """Run commands in a bash shell
* When invoking this tool, the contents of the "command" parameter does NOT need to be XML-escaped.
* You don't have access to the internet via this tool.
* You do have access to a mirror of common linux and python packages via apt and pip.
* State is persistent across command calls and discussions with the user.
* To inspect a particular line range of a file, e.g. lines 10-25, try 'sed -n 10,25p /path/to/the/file'.
* Please avoid commands that may produce a very large amount of output.
* Please run long lived commands in the background, e.g. 'sleep 10 &' or start a server in the background.""",
        "input_schema": {
            "type": "object",
            "properties": {
                "command": {
                    "type": "string",
                    "description": "The bash command to run."
                }
            },
            "required": ["command"]
        }
    }
```

Information of the given edit tool:

```
def tool_info():
    return {
        "name": "editor",
        "description": """Custom editing tool for viewing, creating and editing files
* State is persistent across command calls and discussions with the user
* If `path` is a file, `view` displays the result of applying `cat -n`. If `path` is a directory, `view` lists non-hidden
    ↪ files and directories up to 2 levels deep
* The `create` command cannot be used if the specified `path` already exists as a file
* If a `command` generates a long output, it will be truncated and marked with `<response clipped>`
* The `undo_edit` command will revert the last edit made to the file at `path`
\nNotes for using the `str_replace` command:
* The `old_str` parameter should match EXACTLY one or more consecutive lines from the original file. Be mindful of
    ↪ whitespaces!
* If the `old_str` parameter is not unique in the file, the replacement will not be performed. Make sure to include
    ↪ enough context in `old_str` to make it unique
* The `new_str` parameter should contain the edited lines that should replace the `old_str`""",
        "input_schema": {
```

```
        "type": "object",
        "properties": {
            "command": {
                "type": "string",
                "enum": ["view", "create", "str_replace", "insert", "undo_edit"],
                "description": "The commands to run. Allowed options are: `view`, `create`, `str_replace`, `insert`,
                    ↪ `undo_edit`."
            },
            "file_text": {
                "description": "Required parameter of `create` command, with the content of the file to be created.",
                "type": "string"
            },
            "insert_line": {
                "description": "Required parameter of `insert` command. The `new_str` will be inserted AFTER the line
                    ↪ `insert_line` of `path`.",
                "type": "integer"
            },
            "new_str": {
                "description": "Required parameter of `str_replace` command containing the new string. Required
                    ↪ parameter of `insert` command containing the string to insert.",
                "type": "string"
            },
            "old_str": {
                "description": "Required parameter of `str_replace` command containing the string in `path` to
                    ↪ replace.",
                "type": "string"
            },
            "path": {
                "description": "Absolute path to file or directory, e.g. `/repo/file.py` or `/repo`.",
                "type": "string"
            },
            "view_range": {
                "description": "Optional parameter of `view` command when `path` points to a file. If none is given,
                    ↪ the full file is shown. If provided, the file will be shown in the indicated line number range,
                    ↪ e.g. [11, 12] will show lines 11 and 12. Indexing at 1 to start. Setting `[start_line, -1]`
                    ↪ shows all lines from `start_line` to the end of the file.",
                "items": {
                    "type": "integer"
                },
                "type": "array"
            }
        },
        "required": ["command", "path"]
    }
}
```

A.2 Parent Selection

At each iteration, we select a subset of agents from the archive as parents to self-modify and produce new child agents (Section 3). We use a mechanism similar to that of Zhang et al. (2025b), inspired by Ecoffet et al. (2019), that is roughly proportional to an agent's performance score and inversely proportional to the number of children that successfully compiled. This selection mechanism biases sampling toward agents that outperform the current frontier average while down-weighting agents that have already produced many children, retaining smooth probabilistic exploration and automatically adapting as the archive improves over time. The details of the parent selection process are outlined below.

At each iteration t of the DGM-H run, let

$$\mathcal{A}^t = \{a_0, a_1, \ldots, a_t\}$$

denote the archive of candidate agents with associated performance scores $\alpha_i = \text{performance}(a_i)$. All agents in the archive are eligible for parent selection.

We first compute a dynamic midpoint based on the current performance distribution. Let

$$\alpha_{mid} = \frac{1}{m} \sum_{j \in \mathcal{T}^t} \alpha_j,$$

where $\mathcal{T}^t \subset \mathcal{A}^t$ indexes the top-m highest-performing agents at iteration t (with $m = 3$ in our experiments). This midpoint adapts over time and reflects the current performance frontier.

Each agent's score is first passed through a sigmoid transformation:

$$s_i \;=\; \frac{1}{1+\exp\big(-\lambda(\alpha_i-\alpha_{mid})\big)},$$

where $\lambda > 0$ controls the sharpness of selection. We set $\lambda = 10$.

To encourage exploration, we introduce a novelty bonus based on the number of compiled children n_i produced by agent a_i:

$$h_i \;=\; \frac{1}{1+n_i}.$$

We then form an unnormalized weight

$$w_i \;=\; s_i\, h_i,$$

which balances performance and novelty.

The weights are normalized to form a categorical distribution:

$$p_i \;=\; \begin{cases} \dfrac{w_i}{\sum_{j=0}^{t} w_j}, & \text{if } \sum_{j=0}^{t} w_j > 0, \\ \dfrac{1}{t+1}, & \text{otherwise.} \end{cases}$$

We sample parents independently with replacement according to this distribution:

$$\{\,\text{parents}\,\} \;\sim\; \text{Categorical}\big(\{p_i\}_{i=0}^{t}\big).$$

A wide range of search and exploration strategies has been proposed in prior work (Coulom, 2006; Silver et al., 2016; Herr et al., 2025; Wang et al., 2025b; Weng et al., 2026). We present preliminary evidence that the DGM-H can begin to autonomously rediscover and adapt such strategies by modifying its own exploration dynamics (Appendix E.5). An open research direction is whether self-improving systems can reliably discover search and exploration mechanisms that outperform carefully handcrafted algorithms.

A.3 Pseudocode

This is the pseudocode of the DGM-H, described in Section 3:

Algorithm 1: Darwin Gödel Machine with Hyperagents (DGM-H)

Input: Initial agent a^0, task set $\mathcal{T}$, maximum iterations T
Output: Archive of scored agents $\mathcal{A}$

```
s^0 ← EVALUATE(a^0, T)
initialize A ← {(a^0, s^0)}                          // Start with initial agent
for t ← 1 to T do
    P ← SELECTPARENTS(A)                             // Sample parent agents
    foreach (a, ·) ∈ P do
        a' ← a.MODIFY(a, A)                          // Metacognitive self-modification
        s' ← EVALUATE(a', T)                         // Evaluate on tasks
        if ISVALID(a') then
            A ← A ∪ {(a', s')}                       // Add compiled child agent
        end
    end
end
return A
```

A.4 Multi-domain Optimization

When optimizing for multiple domains within the same run, hyperagents are evaluated on tasks from different domains and have access to all evaluations across these tasks during self-modification. We do not specify which particular domain or task to prioritize. Parent selection is based on the average performance across domains. As a result, improvements in any domain increase selection probability, while regressions reduce it. Because the meta agent can inspect evaluations from any task, it can introduce shared mechanisms (e.g., structured reasoning, memory, and error handling) that benefit multiple domains simultaneously. Thus, rather than manually specifying which task or domain to optimize, hyperagents can optimize across multiple domains within the same run.

B Baseline Details

We outline the pseudocode for each baseline described in Section 4.1, provide a comparison table summarizing their key differences (Table 1), and include a detailed conceptual figure that visually contrasts the architectural components and modification mechanisms across DGM variants and hyperagents (Figure 5).

Method	Self-improving meta agents	Open-ended exploration	Metacognitive self-modification (i.e., hyperagents)
DGM-H	✓	✓	✓
DGM-H w/o self-improve	✗	✓	✓
DGM-H w/o open-ended exploration	✓	✗	✓
DGM	✓	✓	✗
DGM-custom	✓	✓	✗

Table 1 Comparison of methods by self-improvement, open-ended exploration, and metacognitive self-modification.

This is the pseudocode of the baseline DGM-H without self-improving agents (ADAS, Hu et al., 2025):

Algorithm 2: DGM-H without self-improving meta agents (DGM-H w/o self-improve)

Input: Initial agent a^0, task set $\mathcal{T}$, maximum iterations T
Output: Archive of scored agents $\mathcal{A}$

$s^0 \leftarrow \textsc{Evaluate}(a^0, \mathcal{T})$
initialize $\mathcal{A} \leftarrow \{(a^0, s^0)\}$
for $t \leftarrow 1$ **to** T **do**
  $\mathcal{P} \leftarrow \textsc{SelectParents}(\mathcal{A})$
  foreach $(a, \cdot) \in \mathcal{P}$ **do**
    $a' \leftarrow a^0.\textsc{Modify}(a, \mathcal{A})$ `// Modify with initial agent`
    $s' \leftarrow \textsc{Evaluate}(a', \mathcal{T})$
    if $\textsc{IsValid}(a')$ **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(a', s')\}$
    end
  end
end
return $\mathcal{A}$

This is the pseudocode of the baseline DGM-H without open-ended exploration:

Algorithm 3: DGM-H without open-ended exploration (DGM-H w/o open-ended exploration)

Input: Initial agent a^0, task set $\mathcal{T}$, maximum iterations T
Output: Archive of scored agents $\mathcal{A}$

$s^0 \leftarrow \textsc{Evaluate}(a^0, \mathcal{T})$
initialize $\mathcal{A} \leftarrow \{(a^0, s^0)\}$
for $t \leftarrow 1$ **to** T **do**
  $\mathcal{P} \leftarrow \textsc{SelectParents}(\mathcal{A})$
  foreach $(a, \cdot) \in \mathcal{P}$ **do**
    $a' \leftarrow a.\textsc{Modify}(a, \mathcal{A})$
    $s' \leftarrow \textsc{Evaluate}(a', \mathcal{T})$
    if *IsValid*(a') **then**
      $\mathcal{A} \leftarrow \{(a', s')\}$ `// Only keep the latest agent`
    end
  end
end
return $\mathcal{A}$

This is the pseudocode for the original DGM (Zhang et al., 2025b), framed within the hyperagent setting:

Algorithm 4: Darwin Gödel Machine (DGM)

Input: Initial agent a^0, task set $\mathcal{T}$, maximum iterations T
Output: Archive of scored agents $\mathcal{A}$

$s^0 \leftarrow \textsc{Evaluate}(a^0, \mathcal{T})$
initialize $\mathcal{A} \leftarrow \{(a^0, s^0)\}$
for $t \leftarrow 1$ **to** T **do**
  $\mathcal{P} \leftarrow \textsc{SelectParents}(\mathcal{A})$
  foreach $(a, \cdot) \in \mathcal{P}$ **do**
    $instr \leftarrow \textsc{InstrGen}(a)$ `// Handcrafted instruction-generation`
    $a' \leftarrow a.\textsc{Modify}(a, instr)$ `// Self-modification`
    $s' \leftarrow \textsc{Evaluate}(a', \mathcal{T})$
    if *IsValid*(a') **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(a', s')\}$
    end
  end
end
return $\mathcal{A}$

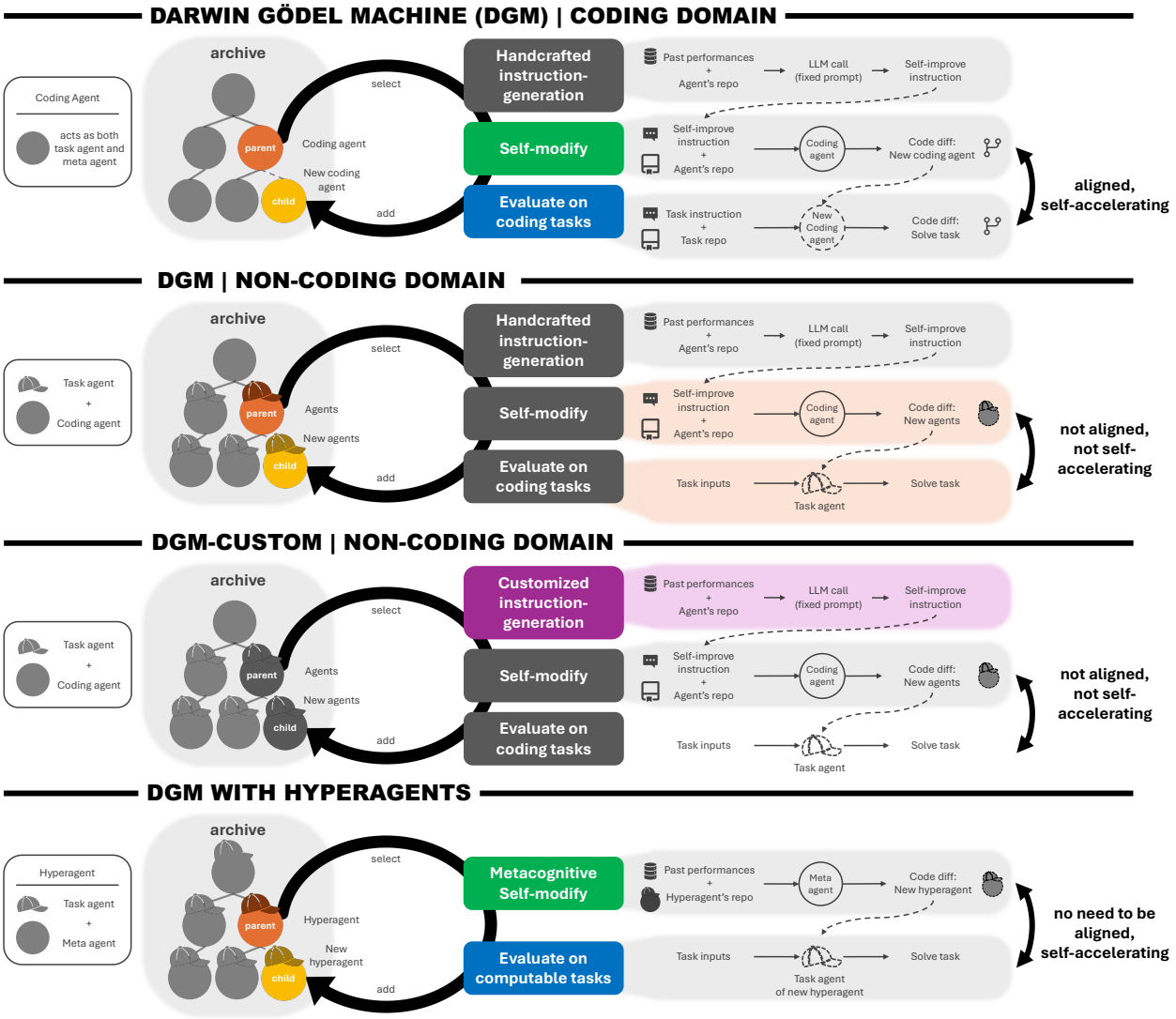

Figure 5 Conceptual comparison of DGM variants, highlighting which components change across settings and how hyperagents address limitations in the original DGM implementation. (First row) The original DGM. The same coding agent serves as both the task agent and the meta agent. Because both evaluation and self-modification are coding tasks, improvements in coding ability translate into improved self-modification, enabling the DGM to improve at improving in coding domains. (Second row) The DGM adapted to non-coding domains. The coding agent remains as the meta agent, but the evaluation tasks are no longer coding tasks, so a separate task agent is required. Task performance no longer reliably reflects the meta agent's ability to generate better task agents, breaking the alignment that enables the meta agent to improve at improving. (Third row) The DGM-custom baseline. The handcrafted instruction-generation mechanism is customized to the target domain but remains non-modifiable. (Fourth row) The DGM with Hyperagents (DGM-H). A hyperagent integrates a task agent and a meta agent within a single editable program, enabling metacognitive self-modification (i.e., modifying not only task-solving behavior but also the procedure that generates future self-modifications). As a result, the DGM-H can improve its improvement mechanism while optimizing for any computable task.

The handcrafted instruction-generation step in the original DGM:

```
diagnose_prompt = """Here is the implementation of the coding agent.

# Coding Agent Implementation
----- Coding Agent Implementation Start -----
{code}
----- Coding Agent Implementation End -----
```

```
Your task is to identify ONE detailed plan that would improve the agent's coding ability. The improvement should not be
    ↪ specific to any particular GitHub issue or repository.

# Agent Running Log
----- Agent Running Log Start -----
{md_log}
----- Agent Running Log End -----

# GitHub Issue
The GitHub issue that the agent is trying to solve.
----- GitHub Issue Start -----
{github_issue}
----- GitHub Issue End -----

# Predicted Patch
The agent's predicted patch to solve the issue.
----- Predicted Patch Start -----
{predicted_patch}
----- Predicted Patch End -----

# Private Test Patch
SWE-bench's official private tests to detect whether the issue is solved. This is not available to the agent during
    ↪ evaluation. The agent should try to implement its own tests.
----- Private Test Patch Start -----
{test_patch}
----- Private Test Patch End -----

# Issue Test Results
The test results from SWE-bench using the above official private tests.
----- Issue Test Results Start -----
{eval_log}
----- Issue Test Results End -----

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:
- "log_summarization": Analyze the above logs and summarize how the agent tried to solve the GitHub issue. Note which
    ↪ tools and how they are used, the agent's problem-solving approach, and any issues encountered.
- "potential_improvements": Identify potential improvements to the coding agent that could enhance its coding
    ↪ capabilities. Focus on the agent's general coding abilities (e.g., better or new tools usable across any
    ↪ repository) rather than issue-specific fixes (e.g., tools only usable in one framework). All necessary
    ↪ dependencies and environment setup have already been handled, so do not focus on these aspects.
- "improvement_proposal": Choose ONE high-impact improvement from the identified potential improvements and describe it
    ↪ in detail. This should be a focused and comprehensive plan to enhance the agent's overall coding ability.
- "implementation_suggestion": Referring to the coding agent's summary and implementation, think critically about what
    ↪ feature or tool could be added or improved to best implement the proposed improvement. If the proposed feature
    ↪ can be implemented by modifying the existing tools, describe the modifications needed, instead of suggesting a
    ↪ new tool.
- "problem_description": Phrase the improvement proposal and implementation suggestion as a GitHub issue description. It
    ↪ should clearly describe the feature so that a software engineer viewing the issue and the repository can
    ↪ implement it.

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."""
```

The customized instruction-generation step in DGM-custom:

```
diagnose_prompt_customized = """
Here is the implementation of the coding agent and task agent.

# Coding Agent Implementation
----- Coding Agent Implementation Start -----
{code_codingagent}
----- Coding Agent Implementation End -----

# Task Agent Implementation
----- Task Agent Implementation Start -----
{code_taskagent}
----- Task Agent Implementation End -----

Your task is to identify ONE detailed plan that would improve the coding/task agent. The improvement should not be
    ↪ specific to any particular task instance or repository.

# Task Info
----- Task -----
{task_info}
----- Task End -----
```

```
# Report
----- Report -----
{report}
----- report End -----

# Agent Running Log
----- Agent Running Log Start -----
{md_log}
----- Agent Running Log End -----

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:
- "log_summarization": Analyze the above logs and summarize how the agent tried to solve the given task. Note which tools
    ↪ and how they are used, the agent's problem-solving approach, and any issues encountered.
- "potential_improvements": Identify potential improvements to the coding/task agent that could enhance its
    ↪ coding/task-solving capabilities. Focus on the agent's general abilities (e.g., better or new tools usable across
    ↪ any repository) rather than issue-specific fixes (e.g., tools only usable in one framework). All necessary
    ↪ dependencies and environment setup have already been handled, so do not focus on these aspects.
- "improvement_proposal": Choose ONE high-impact improvement from the identified potential improvements and describe it
    ↪ in detail. This should be a focused and comprehensive plan to enhance the agent's overall coding/task-solving
    ↪ ability.
- "implementation_suggestion": Referring to the coding/task agent's summary and implementation, think critically about
    ↪ what feature or tool could be added or improved to best implement the proposed improvement. If the proposed
    ↪ feature can be implemented by modifying the existing tools, describe the modifications needed, instead of
    ↪ suggesting a new tool.
- "problem_description": Phrase the improvement proposal and implementation suggestion as a GitHub issue description. It
    ↪ should clearly describe the feature so that a software engineer viewing the issue and the repository can
    ↪ implement it.

Your response will be automatically parsed, so ensure that the string response is precisely in the correct format."""
```

C Domain Details

This appendix provides detailed descriptions of each domain used for evaluation: Polyglot (Appendix C.1), paper review (Appendix C.2), robotics reward design (Appendix C.3), Olympiad-level math grading (Appendix C.4). For each domain, we specify the agent's input and required output for a given task, the evaluation protocol, and representative static baselines (Table 2).

Domain	Input	Output	Metric	Train	Validation	Test
Coding (Polyglot)	Repo + instr.	Code patch	Pass@1	60	-	165
Paper Review	Paper text	Accept / Reject	Accuracy	100	100	100
Robotics Reward Design	Task desc.	Reward fn.	Task score	6	-	6
IMO Grading	Problem + sol.	Grade (0/1/6/7)	Accuracy	100	100	100

Table 2 Summary of domains. During self-modification, agents' evaluations on training tasks are available and can be used as feedback. Validation tasks are used for parent selection. If a validation split is not available, the performance component used for parent selection is based on training performance instead. Test tasks are held-out and used only for the final evaluation of the selected agents.

C.1 Polyglot

In the Polyglot coding benchmark (Gauthier, 2024), each task consists of a software repository and a natural language instruction describing a desired change to the codebase. The agent is given access to the full repository and must modify the files to correctly implement the instruction, producing a patch (i.e., a set of code edits) applied to the repository. Performance is evaluated by running a predefined test suite on the modified repository. A task is considered to be successfully done if all tests pass. We follow the setup used in the DGM (Zhang et al., 2025b), which largely mirrors the Polyglot leaderboard configuration, with one key difference: the leaderboard reports pass@2, allowing the agent to view feedback from ground-truth tests once, whereas we report pass@1, in which the agent never sees ground-truth test results. We adopt the same

training and test splits as in the DGM. Training tasks are selected as a random subset of the full benchmark, comprising a total of 60 tasks. If an agent achieves more than 40% success on an initial 10-task subset, it is subsequently evaluated on the remaining 50 training tasks. There is no validation subset for this domain. As a final evaluation to more accurately assess performance improvements, we evaluate the generated agents on the full Polyglot benchmark, which consists of 165 unseen tasks.

Initial 10 training tasks for preliminary evaluation:

- `go__dominoes`
- `cpp__all-your-base`
- `python__dominoes`
- `java__sgf-parsing`
- `javascript__robot-name`
- `rust__variable-length-quantity`
- `python__beer-song`
- `go__book-store`
- `javascript__bottle-song`
- `rust__bowling`

Additional 50 training tasks for full evaluation:

- `javascript__queen-attack`
- `rust__wordy`
- `python__dot-dsl`
- `java__satellite`
- `cpp__diamond`
- `rust__accumulate`
- `go__error-handling`
- `cpp__queen-attack`
- `rust__poker`
- `python__sgf-parsing`
- `rust__react`
- `java__ledger`
- `go__connect`
- `rust__macros`
- `javascript__triangle`
- `java__zipper`
- `java__bowling`
- `python__tree-building`
- `javascript__say`
- `java__wordy`
- `python__food-chain`
- `javascript__wordy`
- `python__poker`
- `javascript__grade-school`
- `cpp__gigasecond`
- `java__forth`
- `python__dominoes`
- `go__word-search`
- `javascript__simple-linked-list`
- `go__counter`
- `java__react`
- `javascript__ocr-numbers`
- `python__scale-generator`
- `java__go-counting`
- `rust__doubly-linked-list`
- `python__grade-school`
- `javascript__forth`
- `python__wordy`
- `java__mazy-mice`
- `cpp__bank-account`
- `python__zipper`
- `java__custom-set`
- `java__rest-api`
- `go__transpose`
- `rust__gigasecond`
- `rust__say`
- `go__food-chain`
- `rust__pig-latin`
- `go__markdown`
- `go__crypto-square`

C.2 Paper Review

The data in this domain are drawn from Zhao et al. (2026). Each task in the paper review domain consists of the full text of an AI research paper. The agent must predict a binary accept or reject decision, simulating the role of a conference reviewer. Ground-truth labels correspond to real acceptance decisions from top-tier machine learning conferences, including ICLR 2024/2025 and NeurIPS 2023/2024. Performance is measured by classification accuracy with respect to these labels. We randomly sample tasks to construct training, validation, and test splits, each containing 100 tasks. During training, the agent is first evaluated on a subset of 10 tasks from the training split. If the agent succeeds on at least one of these tasks, it is then evaluated on the full set of 100 training tasks.

AI-Scientist-v2 (Yamada et al., 2025) employs an AI reviewer to automatically improve generated AI research papers. We adopt the AI reviewer proposed in that work as our representative static baseline:

```
reviewer_system_prompt_base = (
    "You are an AI researcher who is reviewing a paper that was submitted to a prestigious ML venue."
    "Be critical and cautious in your decision."
)

reviewer_system_prompt_neg = (
    reviewer_system_prompt_base
    + "If a paper is bad or you are unsure, give it bad scores and reject it."
)
reviewer_system_prompt_pos = (
    reviewer_system_prompt_base
    + "If a paper is good or you are unsure, give it good scores and accept it."
)

template_instructions = """
Respond in the following format:

THOUGHT:
<THOUGHT>

REVIEW JSON:
```json
<JSON>
```

In <THOUGHT>, first briefly discuss your intuitions and reasoning for the evaluation.
Detail your high-level arguments, necessary choices and desired outcomes of the review.
Do not make generic comments here, but be specific to your current paper.
Treat this as the note-taking phase of your review.

In <JSON>, provide the review in JSON format with the following fields in the order:
- "Summary": A summary of the paper content and its contributions.
- "Strengths": A list of strengths of the paper.
- "Weaknesses": A list of weaknesses of the paper.
- "Originality": A rating from 1 to 4 (low, medium, high, very high).
- "Quality": A rating from 1 to 4 (low, medium, high, very high).
- "Clarity": A rating from 1 to 4 (low, medium, high, very high).
- "Significance": A rating from 1 to 4 (low, medium, high, very high).
- "Questions": A set of clarifying questions to be answered by the paper authors.
- "Limitations": A set of limitations and potential negative societal impacts of the work.
- "Ethical Concerns": A boolean value indicating whether there are ethical concerns.
- "Soundness": A rating from 1 to 4 (poor, fair, good, excellent).
- "Presentation": A rating from 1 to 4 (poor, fair, good, excellent).
- "Contribution": A rating from 1 to 4 (poor, fair, good, excellent).
- "Overall": A rating from 1 to 10 (very strong reject to award quality).
- "Confidence": A rating from 1 to 5 (low, medium, high, very high, absolute).
- "Decision": A decision that has to be one of the following: Accept, Reject.

For the "Decision" field, don't use Weak Accept, Borderline Accept, Borderline Reject, or Strong Reject. Instead, only
    ↪ use Accept or Reject.
This JSON will be automatically parsed, so ensure the format is precise.
"""

neurips_form = (
    """
Review Form
Below is a description of the questions you will be asked on the review form for each paper and some guidelines on what
    ↪ to consider when answering these questions.
When writing your review, please keep in mind that after decisions have been made, reviews and meta-reviews of accepted
    ↪ papers and opted-in rejected papers will be made public.

1. Summary: Briefly summarize the paper and its contributions. This is not the place to critique the paper; the authors
    ↪ should generally agree with a well-written summary.
```

```
  - Strengths and Weaknesses: Please provide a thorough assessment of the strengths and weaknesses of the paper, touching
      ↪ on each of the following dimensions:
  - Originality: Are the tasks or methods new? Is the work a novel combination of well-known techniques? (This can be
      ↪ valuable!) Is it clear how this work differs from previous contributions? Is related work adequately cited
  - Quality: Is the submission technically sound? Are claims well supported (e.g., by theoretical analysis or experimental
      ↪ results)? Are the methods used appropriate? Is this a complete piece of work or work in progress? Are the
      ↪ authors careful and honest about evaluating both the strengths and weaknesses of their work
  - Clarity: Is the submission clearly written? Is it well organized? (If not, please make constructive suggestions for
      ↪ improving its clarity.) Does it adequately inform the reader? (Note that a superbly written paper provides
      ↪ enough information for an expert reader to reproduce its results.)
  - Significance: Are the results important? Are others (researchers or practitioners) likely to use the ideas or build on
      ↪ them? Does the submission address a difficult task in a better way than previous work? Does it advance the
      ↪ state of the art in a demonstrable way? Does it provide unique data, unique conclusions about existing data, or
      ↪ a unique theoretical or experimental approach?

2. Questions: Please list up and carefully describe any questions and suggestions for the authors. Think of the things
     ↪ where a response from the author can change your opinion, clarify a confusion or address a limitation. This can
     ↪ be very important for a productive rebuttal and discussion phase with the authors.

3. Limitations: Have the authors adequately addressed the limitations and potential negative societal impact of their
     ↪ work? If not, please include constructive suggestions for improvement.
In general, authors should be rewarded rather than punished for being up front about the limitations of their work and
     ↪ any potential negative societal impact. You are encouraged to think through whether any critical points are
     ↪ missing and provide these as feedback for the authors.

4. Ethical concerns: If there are ethical issues with this paper, please flag the paper for an ethics review. For
     ↪ guidance on when this is appropriate, please review the NeurIPS ethics guidelines.

5. Soundness: Please assign the paper a numerical rating on the following scale to indicate the soundness of the
     ↪ technical claims, experimental and research methodology and on whether the central claims of the paper are
     ↪ adequately supported with evidence.
  4: excellent
  3: good
  2: fair
  1: poor

6. Presentation: Please assign the paper a numerical rating on the following scale to indicate the quality of the
     ↪ presentation. This should take into account the writing style and clarity, as well as contextualization relative
     ↪ to prior work.
  4: excellent
  3: good
  2: fair
  1: poor

7. Contribution: Please assign the paper a numerical rating on the following scale to indicate the quality of the overall
     ↪ contribution this paper makes to the research area being studied. Are the questions being asked important? Does
     ↪ the paper bring a significant originality of ideas and/or execution? Are the results valuable to share with the
     ↪ broader NeurIPS community.
  4: excellent
  3: good
  2: fair
  1: poor

8. Overall: Please provide an "overall score" for this submission. Choices:
  10: Award quality: Technically flawless paper with groundbreaking impact on one or more areas of AI, with exceptionally
      ↪ strong evaluation, reproducibility, and resources, and no unaddressed ethical considerations.
  9: Very Strong Accept: Technically flawless paper with groundbreaking impact on at least one area of AI and excellent
      ↪ impact on multiple areas of AI, with flawless evaluation, resources, and reproducibility, and no unaddressed
      ↪ ethical considerations.
  8: Strong Accept: Technically strong paper with, with novel ideas, excellent impact on at least one area of AI or
      ↪ high-to-excellent impact on multiple areas of AI, with excellent evaluation, resources, and reproducibility,
      ↪ and no unaddressed ethical considerations.
  7: Accept: Technically solid paper, with high impact on at least one sub-area of AI or moderate-to-high impact on more
      ↪ than one area of AI, with good-to-excellent evaluation, resources, reproducibility, and no unaddressed ethical
      ↪ considerations.
  6: Weak Accept: Technically solid, moderate-to-high impact paper, with no major concerns with respect to evaluation,
      ↪ resources, reproducibility, ethical considerations.
  5: Borderline accept: Technically solid paper where reasons to accept outweigh reasons to reject, e.g., limited
      ↪ evaluation. Please use sparingly.
  4: Borderline reject: Technically solid paper where reasons to reject, e.g., limited evaluation, outweigh reasons to
      ↪ accept, e.g., good evaluation. Please use sparingly.
  3: Reject: For instance, a paper with technical flaws, weak evaluation, inadequate reproducibility and incompletely
      ↪ addressed ethical considerations.
  2: Strong Reject: For instance, a paper with major technical flaws, and/or poor evaluation, limited impact, poor
      ↪ reproducibility and mostly unaddressed ethical considerations.
  1: Very Strong Reject: For instance, a paper with trivial results or unaddressed ethical considerations

9. Confidence: Please provide a "confidence score" for your assessment of this submission to indicate how confident you
     ↪ are in your evaluation. Choices:
  5: You are absolutely certain about your assessment. You are very familiar with the related work and checked the
      ↪ math/other details carefully.
  4: You are confident in your assessment, but not absolutely certain. It is unlikely, but not impossible, that you did
```

```
        ↪ not understand some parts of the submission or that you are unfamiliar with some pieces of related work.
  3: You are fairly confident in your assessment. It is possible that you did not understand some parts of the submission
        ↪ or that you are unfamiliar with some pieces of related work. Math/other details were not carefully checked.
  2: You are willing to defend your assessment, but it is quite likely that you did not understand the central parts of
        ↪ the submission or that you are unfamiliar with some pieces of related work. Math/other details were not
        ↪ carefully checked.
  1: Your assessment is an educated guess. The submission is not in your area or the submission was difficult to
        ↪ understand. Math/other details were not carefully checked.
"""
    + template_instructions
)


class TaskAgent(AgentSystem):
    def forward(self, inputs):
        reviewer_system_prompt = reviewer_system_prompt_neg
        review_instruction_form = neurips_form

        base_prompt = review_instruction_form
        base_prompt += f"""
Here is the paper you are asked to review:
```
{inputs['paper_text']}
```"""
        instruction = reviewer_system_prompt + base_prompt

        self.log(f"Input: {repr(instruction)}")
        response, new_msg_history, _ = get_response_from_llm(
            msg=instruction,
            model=self.model,
            msg_history=[],
        )
        self.log(f"Output: {repr(response)}")

        # Extract the response
        prediction = "None"
        try:
            extracted_jsons = extract_jsons(new_msg_history[-1]['text'])
            prediction = extracted_jsons[-1]['Decision']
        except Exception as e:
            self.log(f"Error extracting prediction: {e}")
            prediction = "None"

        return prediction, new_msg_history
```

C.3 Robotics Reward Design

Each task in the robotics reward design domain specifies a robotic control objective in the Genesis simulator (Genesis, 2024) using a Go2 quadruped robot. The agent is given a textual description of the task (e.g., walk forward at a target velocity) and outputs a Python reward function, which is then used to train a RL policy (i.e., PPO, Schulman et al., 2017) for the robot. Performance is evaluated by executing the trained RL policy in the simulator and computing the task performance measure (e.g., velocity tracking error). Scores are averaged over repeated evaluations to reduce variance due to stochasticity in reward generation or RL.

The training task requires generating a reward function that enables the robot to walk forward while tracking a target linear velocity. Performance is measured using the mean squared error between the commanded and actual walking velocities. During training, each agent is initially evaluated 3 repeated times on the same task, generating one reward function per evaluation. If at least one generated reward function yields a non-zero performance score, the agent is evaluated 3 additional times. The final performance score is reported as the average across the 6 evaluations. No separate validation task is curated for this domain.

To assess whether the same agent can generate suitable reward functions across different robotics tasks, we pair a relatively simple training task with a more challenging test task on the same robot. The test task requires generating a reward function that trains the robot to maximize torso height. Reward functions that are effective for forward walking do not induce jumping behaviors, which are more optimal for maximizing torso height. Moreover, directly incentivizing torso height (the performance measure) typically leads to a suboptimal standing behavior of standing stall. Achieving high performance therefore requires non-myopic reward design that encourages intermediate behaviors, such as lowering the torso before jumping.

The default reward function for the test task directly rewards the performance measure of maximizing torso

height. This always produces a behavior in which the robot simply stands as tall as possible (Figure 6):

```python
def compute_reward(env) -> Tuple[Tensor, Dict, Dict]:
    """
    The robot maximizes its vertical position.
    """
    height = env.base_pos[:, 2]
    total_reward = height

    reward_components = {"height": height}
    reward_scales = {"height": 1.0}

    return total_reward, reward_components, reward_scales
```

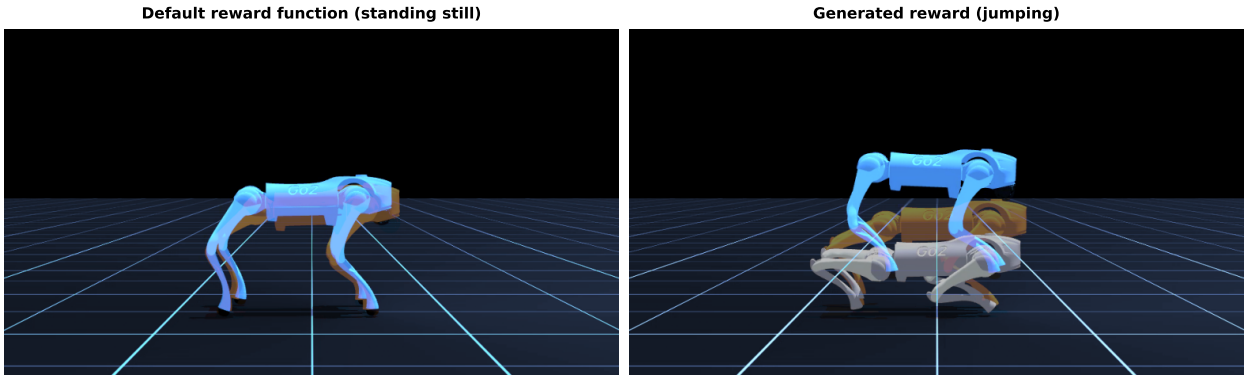

Figure 6 Comparison between (Left) the default reward function, which leads to a stationary posture of standing tall, and (Right) a generated reward function that induces jumping behavior. The orange robot indicates the start position, the white robot indicates an intermediate position during the episode, and the blue robot indicates the end position. Screenshot from https://github.com/Genesis-Embodied-AI/Genesis

C.4 Olympiad-level Math Grading

In the Olympiad-level math grading domain, tasks are drawn from IMO-GradingBench (Luong et al., 2025). Each task consists of an Olympiad-level math problem, a candidate solution, reference solutions, and grading guidelines. The agent is required to assign a discrete score from the set {0, 1, 6, 7}, corresponding to the categories {incorrect, partial, almost, correct}. The agent's output is a single numeric grade. Performance is measured by accuracy with respect to expert human annotations, with additional analyses provided in Appendix E.4. We use randomly sampled subsets of tasks for training, validation, and testing, with 100 tasks in each split. During training, an agent is first evaluated on a subset of 10 tasks (out of 100). If the agent succeeds on at least one of these tasks, it is then evaluated on the full training set.

Representative static baseline (ProofAutoGrader) from Luong et al. (2025):

```
PROMPT = """You are an expert grader for the International Mathematics Olympiad (IMO).
Your task is to evaluate a proposed solution strictly and rigorously.
Keep in mind the standards at the IMO are extremely high: only arguments that are logically sound, complete, and precise
    ↪ should be rewarded.

### General Scoring Rubric
Scores are assigned on a 0-7 scale. The general guidelines are:
* **7 Points (Correct):** The solution is complete, correct, and fully rigorous. If the submission contains incorrect
    ↪ attempts or lines of reasoning but ultimately presents a complete and correct solution, it should still be
    ↪ awarded full points; the presence of earlier, discarded work does not detract from the final correct proof.
* **6 Points (Almost Correct):** The solution is almost correct with a sound core argument, but contains minor errors in
    ↪ calculation or small gaps in logic. Missing proofs for major components, unjustified claims, or sketchy arguments
    ↪ are **not** eligible for 6 points.
* **1 Point (Partial Progress):** The solution demonstrates substantial progress explicitly mentioned in the grading
    ↪ guidelines. Initial observations, reformulating the problem without making substantive headway, or proving
    ↪ partial results not mentioned in the grading guidelines are generally **not** eligible for this score.
* **0 Points (Incorrect):** The solution doesn't make substantial progress that is a key step in the full solution or is
    ↪ fundamentally flawed. All partial progress without key results or lacking rigor also fall in this category.

### Input Data and Interpretation
You are provided with the following:
1. **Problem Statement:** The IMO problem.
```

```
2. **Ground Truth Solution:** A reference solution. Assume this solution is correct. It demonstrates one valid approach.
3. **Specific Grading Guidelines:** Criteria for awarding credit for this specific problem. These guidelines take
    ↪ precedence over the General Scoring Rubric, especially for partial credit.
4. **Proposed Solution:** The student submission.

### Evaluation Process
You must follow this structured process:
1. **Analyze References:** Meticulously read and understand the problem and Ground Truth Solution check the Specific
    ↪ Grading Guidelines. Identify the key steps for a complete solution and the criteria for partial credit.
2. **Step-by-Step Verification:** Verify the logical validity and rigor of every step. Identify all flaws, gaps,
    ↪ assumptions, and errors. **Make sure you fully understand every piece of logic behind each step of the proposed
    ↪ solution, you must be careful for solutions that 'pretend' to be correct.**
3. **Assess Progress:** Determine the extent of non-trivial progress made.
4. **Score Determination:** Compare the findings against the Specific Grading Guidelines and the General Rubric to
    ↪ determine the final score.

### Output Requirements
You must provide your final score in the format <points>N out of 7</points>.
Ensure the '<points>' block is used **only once**, as your answer will be parsed based on the first <points> </points>
    ↪ block that appears in your whole response.

**PROBLEM STATEMENT**
{problem_statement}

**GROUND-TRUTH SOLUTION**
{solution}

**SPECIFIC GRADING GUIDELINES**
{grading_guidelines}

**PROPOSED SOLUTION**
{student_answer}

Present your detailed thought process and formal justification based on the scoring rubric and grading guidelines, and
    ↪ finally present your final score in the format below.

[Select one of the following options]
<points>7 out of 7</points>
<points>6 out of 7</points>
<points>1 out of 7</points>
<points>0 out of 7</points>
"""


class TaskAgent(AgentSystem):
    """
    An automatic grader for IMO-Proof Bench.
    """
    def forward(self, inputs):
        # Check if all required inputs are present
        if not all(key in inputs for key in ["problem", "solution", "grading_guidelines", "student_answer"]):
            return None, []

        # Get response
        instruction = PROMPT.format(
            problem_statement=inputs["problem"],
            solution=inputs["solution"],
            grading_guidelines=inputs["grading_guidelines"],
            student_answer=inputs["student_answer"],
        )
        new_msg_history = chat_with_agent(instruction, model=self.model, msg_history=[], logging=self.log)

        # Extract the response
        prediction = "None"
        try:
            raw_text = new_msg_history[-1].get('text', '')

            # Extract content between <points>...</points>
            match = re.search(r"<points>(.*?)</points>", raw_text, re.DOTALL)
            if match:
                points_text = match.group(1).strip() # e.g., "7 out of 7"

                # Extract just the leading integer
                num_match = re.search(r"\d+", points_text)
                if num_match:
                    prediction = int(num_match.group()) # e.g., 7
                    # Map prediction to reward text
                    reward_map = {
                        0: "incorrect",
                        1: "partial",
                        6: "almost",
                        7: "correct",
```

```
                }
                prediction = reward_map.get(prediction, "None")
            else:
                self.log("No numeric score found inside <points> tag.")
                prediction = "None"

        else:
            self.log("No <points> tag found in model output.")
            prediction = "None"

    except Exception as e:
        self.log(f"Error extracting prediction: {e}")
        prediction = "None"

    return prediction, new_msg_history
```

D Experiment Details

This appendix provides additional experimental details to support reproducibility of the results. We first summarize the FMs and hyperparameters used for self-modification and task evaluation across domains (Appendix D.1), followed by an estimate of the computational cost of running the DGM-H in each setting (Appendix D.2). We then formally define the improvement@k metric used to quantify an agent's ability to produce improved variants under a fixed budget (Appendix D.3). Finally, we describe the procedure used to select transfer agents for cross-domain experiments (Appendix D.4).

D.1 Hyperparameters for FMs

Table 3 summarizes the foundation models (FMs) used across experimental settings. For the Polyglot coding domain, we adopt the same FMs and temperature configurations as Zhang et al. (2025b) to ensure a fair comparison. In all other domains, we use Claude-4.5-Sonnet for self-modification, given its strong performance on coding. For task evaluation, we select the FM based on practical considerations, including computational cost, rate limits, response latency, and overall task competence. In the robotics reward design setting, where the agent must implement reward functions in code, we again use Claude-4.5-Sonnet. For Olympiad-level mathematics grading, which requires substantial mathematical reasoning, we use o4-mini. The temperature is set to 0.0 for all FMs in every setting, except for o4-mini, which is fixed at 1.0.

Table 3 Foundation models used in each experiment setting for self-modification or task evaluation.

Domain	Self-modification	Evaluation
Polyglot	Claude 3.5 Sonnet (New)	o3-mini
Paper review	Claude 4.5 Sonnet	GPT-4o
Robotics reward design	Claude 4.5 Sonnet	Claude 4.5 Sonnet
IMO-level grading	Claude 4.5 Sonnet	o4-mini

D.2 Cost Estimate

Running the DGM-H for 100 iterations incurs a cost of approximately 33M tokens for the self-modification phase alone (excluding task evaluation). The total cost of an experiment therefore consists of the self-modification cost plus the cost of task evaluation. For the paper review and robotics reward design experiments, the evaluation cost per iteration is 0.506M tokens (0.5M tokens for paper review evaluation + 0.006M tokens for robotics reward design evaluation). Consequently, for a 100-iteration run (Section 5), the estimated total cost is 33M tokens for self-modification plus 0.506M $\times$ 100 for evaluation, yielding a total of approximately 88.6M tokens.

A more granular break down of the task evaluation cost is:

FM	Benchmark	Number of Tasks	Cost Estimate (M, tokens)
o3-mini	Polyglot	60	0.89M
GPT-4o	Paper review	100	0.5M
Claude-4.5-sonnet	Robotics reward design	6	0.006M
o4-mini	IMO-GradingBench	100	0.11M

D.3 Improvement@k Metric

Let M denote an initial meta agent, A an initial task agent, and $\mathcal{T}$ a fixed set of evaluation tasks. Let $\mathcal{G}$ denote an agent-generation algorithm (e.g., DGM or DGM-H variants). The meta agent M is held fixed and is allowed to generate up to k new task agents by iteratively applying $\mathcal{G}$ starting from A.

Let

$$\mathcal{A}^{(k)} = \{A^1, A^2, \ldots, A^k\}$$

denote the set of task agents generated by M within k modification steps. Each task agent $A' \in \{A\} \cup \mathcal{A}^{(k)}$ is evaluated on $\mathcal{T}$ using a fixed evaluation procedure $\text{Evaluate}(\cdot, \mathcal{T})$, where higher values indicate better performance.

We define the **improvement@k** metric as

$$\text{imp@}k(M, A, \mathcal{G}, \mathcal{T}) \;=\; \max_{A' \in \mathcal{A}^{(k)}(M,A,\mathcal{G})} \text{Evaluate}(A', \mathcal{T}) \;-\; \text{Evaluate}(A, \mathcal{T}),$$

Intuitively, imp@k measures the maximum performance improvement that a fixed meta agent M, operating under a specific agent-generation algorithm $\mathcal{G}$, can obtain by generating up to k modified task agents starting from the initial task agent A. Larger values of imp@k indicate stronger agent-generation capability under a fixed computational budget and generation procedure.

A limitation of imp@k is that it treats performance improvements as linear, without accounting for differences in difficulty across performance levels. In particular, improvements near saturation (e.g., increasing accuracy from 0.7 to 0.8) may be substantially harder to achieve than equivalent absolute gains at lower performance levels (e.g., from 0.0 to 0.1). As a result, imp@k may underestimate the significance of improvements achieved at higher performance regimes. However, this limitation does not affect the analyses presented in this work, as imp@k is used primarily for relative comparisons under matched initial conditions and fixed evaluation budgets, where all methods are subject to the same saturation effects (Section 5.2).

D.4 Transfer Agent Selection

To select agents for the transfer experiments (Sections 5.2 and 5.3), we use a descendant growth criterion that favors agents which serve as strong stepping stones for subsequent improvements, rather than agents that are merely high-scoring themselves. Concretely, given the final archive at iteration t,

$$\mathcal{A}^t = \{a_0, a_1, \ldots, a_t\},$$

let α_i denote the evaluation score of agent a_i on the source-domain validation set when available, and otherwise on the training set. Let $\text{parent}(j)$ denote the parent of node j in the archive tree, and let $\text{dist}(i, j)$ be the number of edges on the unique path from i to descendant j.

We define the *growth score* of a candidate transfer node i as the discounted average improvement achieved by its descendants relative to i:

$$G_\gamma(i) \;=\; \frac{1}{|\mathcal{D}(i)|} \sum_{j \in \mathcal{D}(i)} \big(\alpha_j - \alpha_i\big)\, \gamma^{\text{dist}(i,j)},$$

where $\mathcal{D}(i)$ is the set of descendants of i in the archive tree and $\gamma \in (0, 1]$ controls how strongly we discount improvements that occur many generations after i. Intuitively, $G_\gamma(i)$ assigns higher weight to agents that reliably generate better descendants within fewer self-modification steps, which we treat as evidence of stronger agent-generation ability.

In our experiments, we set $\gamma = 0.6$ and select the transfer agent with highest $G_{0.6}(i)$. To reduce noise, we only consider nodes with at least 3 descendants.

E Additional Results

This appendix presents additional qualitative and diagnostic results that complement the main findings. We first highlight the best task agents discovered by the DGM-H (Appendix E.1). We then qualitatively analyze how the DGM-H improves task performance across different domains (Appendix E.2) and how it develops meta-level capabilities that improves its ability to self-improve (Appendix E.3). Next, we analyze the behavior of automatically discovered Olympiad-level math graders (Appendix E.4). We also report preliminary experiments in which the DGM-H is allowed to modify its own parent selection mechanism, shedding light on the limits and potential of fully self-referential optimization (Appendix E.5). All experiment logs are open-sourced in our codebase.

E.1 Best Discovered Task Agents

We show portions of the diff patches that contribute to the task agent and are relevant to the domain. The full diff patches are open-sourced in our codebase.

E.1.1 Paper Review

Diff patches contributing to the best task agent discovered by the DGM-H (Section 5.1) for paper review:

```
diff --git a/task_agent.py b/task_agent.py
index 3798256..42ab625 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -5,7 +5,7 @@ from utils.common import extract_jsons
 class TaskAgent(AgentSystem):
     def forward(self, inputs):
         """
- An agent that solves a given task.
+ An agent that solves a given task with enhanced reasoning and error handling.

         Args:
             inputs (dict): A dictionary with input data for the task.
@@ -15,30 +15,80 @@ class TaskAgent(AgentSystem):
                 - prediction (str): The prediction made by the agent.
                 - new_msg_history (list): A list of messages...
         """
- domain = inputs['domain']
- instruction = f"""You are an agent.
+ domain = inputs.get('domain', 'unknown')
+
+ # Enhanced instruction with chain-of-thought reasoning
+ instruction = f"""You are an expert agent solving tasks in the '{domain}' domain.

 Task input:
 ```
 {inputs}
 ```

+Please analyze this task carefully and provide your response. Follow these steps:
+1. Understand the task requirements
+2. Consider relevant approaches or solutions
+3. Provide your final answer
+
 Respond in JSON format with the following schema:
 <json>
 {{
- "response": ...
+ "reasoning": "Brief explanation of your approach",
+ "response": "Your final answer"
 }}
-</json>"""
- new_msg_history = chat_with_agent(instruction, ...)
+</json>

+Note: The 'response' field is required and should contain your answer."""
+
+ try:
```

```
+ new_msg_history = chat_with_agent(
+ instruction,
+ model=self.model,
+ msg_history=[],
+ logging=self.log
+ )
+ except Exception as e:
+ self.log(f"Error in chat_with_agent: {e}")
+ return "Error: Failed to get response from agent", []

- # Extract the response
+ # Extract the response with improved error handling
        prediction = "None"
        try:
- extracted_jsons = extract_jsons(new_msg_history[-1]['text'])
- if extracted_jsons is not None and "response" in extracted_jsons[-1]:
- prediction = extracted_jsons[-1]['response']
+ if new_msg_history and len(new_msg_history) > 0:
+ last_message = new_msg_history[-1].get('text', '')
+ extracted_jsons = extract_jsons(last_message)
+
+ if extracted_jsons is not None and len(extracted_jsons) > 0:
+ last_json = extracted_jsons[-1]
+
+ if "response" in last_json:
+ prediction = last_json['response']
+ if "reasoning" in last_json:
+ self.log(f"Agent reasoning: {last_json['reasoning']}")
+ else:
+ self.log("Warning: JSON response missing 'response' field")
+ prediction = str(last_json)
+ else:
+ self.log("Warning: No valid JSON found in response")
+ prediction = last_message[:500]
        except Exception as e:
            self.log(f"Error extracting prediction: {e}")
- prediction = "None"
+ # Attempt to recover with raw response
+ try:
+ if new_msg_history and len(new_msg_history) > 0:
+ prediction = new_msg_history[-1].get('text', 'Error')[:500]
+ except:
+ prediction = "Error: Complete extraction failure"

        return prediction, new_msg_history
```

```
diff --git a/task_agent.py b/task_agent.py
--- a/task_agent.py
+++ b/task_agent.py
@@ -17,8 +17,35 @@ class TaskAgent(AgentSystem):
        """
        domain = inputs.get('domain', 'unknown')

- # Enhanced instruction with chain-of-thought reasoning
- instruction = f"""You are an expert agent solving tasks...
+ # Domain-specific instructions
+ if domain == 'paper_review':
+ paper_text = inputs.get('paper_text', '')
+ instruction = f"""You are an expert reviewer evaluating academic papers.
+Your task is to decide whether to ACCEPT or REJECT the paper.
+
+Paper to review:
+```
+{paper_text}
+```
+
+Evaluate the paper based on:
+1. Novelty and originality of the research
+2. Technical soundness and methodology
+3. Experimental validation and results
+4. Clarity of presentation
+5. Significance of contributions
+
+Provide your decision in JSON format:
+<json>
+{{
+ "reasoning": "Brief explanation of your decision (2-3 sentences)",
+ "response": "accept or reject"
+}}
+</json>
+
```

```
+IMPORTANT: The 'response' field must be EXACTLY either 'accept' or 'reject'
+(lowercase, one word only)."""
+ else:
+ # Generic instruction for other domains
+ instruction = f"""You are an expert agent solving tasks in the '{domain}' domain.
```

```
diff --git a/task_agent.py b/task_agent.py
--- a/task_agent.py
+++ b/task_agent.py
@@ -20,24 +20,43 @@ class TaskAgent(AgentSystem):
        # Domain-specific instructions
        if domain == 'paper_review':
            paper_text = inputs.get('paper_text', '')
- instruction = f"""You are an expert reviewer evaluating academic papers...
+ instruction = f"""You are a rigorous and critical academic reviewer
+evaluating papers for a top-tier conference. Your task is to decide whether to
+ACCEPT or REJECT the paper. Be skeptical and thorough.

 Paper to review:
 ```
 {paper_text}
 ```

-Evaluate the paper based on:
-1. Novelty and originality of the research
-...
+Evaluate the paper critically based on:
+1. **Novelty and Originality**: Is the contribution truly novel or incremental?
+2. **Technical Soundness**: Are methods rigorous? Any flaws in theory?
+3. **Experimental Validation**: Comprehensive? Sufficient baselines?
+4. **Clarity and Presentation**: Is writing clear? Key details missing?
+5. **Significance**: Does this advance the field meaningfully?
+
+**RED FLAGS that typically warrant REJECTION:**
+- Incremental improvements without significant novelty
+- Missing critical experimental comparisons or baselines
+- Unclear methodology or missing implementation details
+- Overclaimed contributions not supported by evidence
+- Poor writing that obscures understanding
+- Limited experimental validation or weak results
+- Lack of theoretical justification or soundness issues
+- Insufficient comparison with prior state-of-the-art
+
+**Standards for ACCEPTANCE:**
+- Clear novel contribution that advances the field
+- Rigorous methodology with sound theoretical foundation
+- Comprehensive experiments with strong results
+- Well-written with clear explanations
+- Properly compared against relevant baselines
+
+**Be Critical**: Default to REJECT unless the paper clearly meets high standards.

 Provide your decision in JSON format:
 <json>
 {{
- "reasoning": "Brief explanation of your decision (2-3 sentences)",
+ "reasoning": "Brief critical analysis (2-3 sentences on key strengths/weaknesses)",
     "response": "accept or reject"
 }}
 </json>
```

```
diff --git a/task_agent.py b/task_agent.py
--- a/task_agent.py
+++ b/task_agent.py
@@ -20,43 +20,63 @@ class TaskAgent(AgentSystem):
        # Domain-specific instructions
        if domain == 'paper_review':
            paper_text = inputs.get('paper_text', '')
- instruction = f"""You are a rigorous and critical academic reviewer...
+ instruction = f"""You are a critical academic reviewer for a top-tier
+conference. Use a two-stage evaluation process:

 Paper to review:
 ```
 {paper_text}
 ```

-Evaluate the paper critically based on:
-1. **Novelty and Originality**: Is the contribution truly novel?
-...
```

```
-
-**RED FLAGS that typically warrant REJECTION:**
-...
-
-**Standards for ACCEPTANCE:**
-...
-
-**Be Critical**: Default to REJECT unless the paper clearly meets high standards.
-
-Provide your decision in JSON format:
+**STAGE 1: Identify Weaknesses**
+Systematically check for these common issues:
+
+1. **Novelty Issues:**
+ - Is this just an incremental modification of existing work?
+ - Are the contributions overclaimed or trivial?
+
+2. **Methodological Flaws:**
+ - Are key implementation details missing?
+ - Are there questionable assumptions or theoretical gaps?
+
+3. **Experimental Weaknesses:**
+ - Missing important baselines or comparisons?
+ - Weak or marginal improvements?
+
+4. **Presentation Problems:**
+ - Unclear writing or poor organization?
+ - Missing critical details for reproducibility?
+
+5. **Limited Scope:**
+ - Tested only on toy problems?
+ - Narrow applicability?
+
+**STAGE 2: Make Decision**
+A paper should be **REJECTED** if:
+- It has 2 or more significant weaknesses from Stage 1
+- It has 1 critical flaw
+- The contribution is too incremental
+
+A paper should be **ACCEPTED** only if:
+- It has clear novel contributions
+- Strong experimental validation
+- Well-written and technically sound
+- At most minor, fixable issues
+
+**Critical Mindset:** Most papers have significant flaws.
+About 50% of papers should be rejected.
+
+Provide your evaluation in JSON format:
 <json>
 {{
- "reasoning": "Brief critical analysis (2-3 sentences)",
+ "weaknesses_found": "List 2-3 main weaknesses identified in Stage 1",
+ "reasoning": "Based on the weaknesses, explain why you accept or reject",
     "response": "accept or reject"
 }}
 </json>
```

```
diff --git a/task_agent.py b/task_agent.py
--- a/task_agent.py
+++ b/task_agent.py
@@ -20,68 +20,71 @@ class TaskAgent(AgentSystem):
         # Domain-specific instructions
         if domain == 'paper_review':
             paper_text = inputs.get('paper_text', '')
- instruction = f"""You are a critical academic reviewer...
-
-**STAGE 1: Identify Weaknesses**
-...
+ instruction = f"""You are a rigorous peer reviewer for a top-tier
+academic conference. Your reputation depends on making careful, balanced decisions.

-**STAGE 2: Make Decision**
-...
-
-Provide your evaluation in JSON format:
-<json>
-{{
- "weaknesses_found": "List 2-3 main weaknesses identified in Stage 1",
- "reasoning": "Based on the weaknesses, explain why you accept or reject",
- "response": "accept or reject"
```

```
-}}
-</json>
-
-IMPORTANT: The 'response' field must be EXACTLY either 'accept' or 'reject'..."""
+**Paper to Review:**
+{paper_text}

+**Evaluation Framework:**

+**ACCEPTANCE CRITERIA (ALL must be satisfied):**
+1. **Novelty**: Presents genuinely new ideas, not incremental improvements
+2. **Technical Soundness**: Methodology is rigorous, correct, and well-justified
+3. **Significance**: Makes important contributions that advance the field
+4. **Experimental Validation**: Claims thoroughly supported with comprehensive experiments
+5. **Clarity**: Clear, well-organized presentation with proper motivation
+6. **Reproducibility**: Sufficient detail for replication
+7. **Related Work**: Proper comparison with existing methods

+**REJECTION CRITERIA (ANY one can warrant rejection):**
+- Lacks novelty; merely combines existing techniques without insight
+- Methodological flaws or incorrect technical approach
+- Insufficient experimental validation or cherry-picked results
+- Missing critical baselines or unfair comparisons
+- Overclaimed contributions not supported by evidence
+- Poor writing quality that obscures technical content
+- Ethical concerns or reproducibility issues
+- Limited scope or significance

+**Decision Guidelines:**
+- **ACCEPT**: Paper clearly satisfies ALL acceptance criteria with strong evidence
+- **REJECT**: Paper fails one or more acceptance criteria OR matches rejection criteria
+- **When in doubt, err on the side of rejection**

+**Your Review Process:**
+1. Identify the paper's main claims and contributions
+2. Evaluate each acceptance criterion systematically
+3. Check for rejection criteria
+4. Ask: "Does this paper significantly advance the field?"
+5. Make a decision based on evidence, not superficial features

+**Response Format:**
+Provide your decision in this exact JSON format:
+<json>
+{{
+ "response": "accept"
+}}
+</json>

+OR

+<json>
+{{
+ "response": "reject"
+}}
+</json>

+**CRITICAL REQUIREMENTS:**
+- The "response" field must contain ONLY "accept" or "reject" (lowercase)
+- Be thorough and critical - rejecting weak papers maintains scientific standards
+- Your decision should be defensible based on the criteria above
+- ONLY output the single word: "accept" or "reject" """
```

E.1.2 Robotics Reward Design

Diff patches contributing to the best task agent discovered by the DGM-H (Section 5.1) for robotics reward design:

```
diff --git a/task_agent.py b/task_agent.py
index 3798256..98e6706 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -5,7 +5,7 @@ from utils.common import extract_jsons
 class TaskAgent(AgentSystem):
     def forward(self, inputs):
         """
- An agent that solves a given task.
+ An agent that solves a given task with enhanced reasoning and error handling.
         ...
         """
```

```
- domain = inputs['domain']
- instruction = f"""You are an agent.
+ domain = inputs.get('domain', 'general')
+
+ # Enhanced instruction with clearer structure and reasoning guidance
+ instruction = f"""You are an expert AI agent specialized in solving complex tasks.

-Task input:
+Task Domain: {domain}
+
+Task Input:
 ```
 {inputs}
 ```

+Instructions:
+1. Carefully analyze the task input and identify the key requirements
+2. Break down the problem into logical steps if needed
+3. Formulate your response based on the task requirements
+4. Provide your final answer in the specified JSON format
+
 Respond in JSON format with the following schema:
 <json>
 {{
- "response": ...
+ "reasoning": "Brief explanation of your approach",
+ "response": "Your final answer here"
 }}
-</json>"""
- new_msg_history = chat_with_agent(instruction, model=self.model, msg_history=[], logging=self.log)
+</json>

- # Extract the response
+IMPORTANT: Always include both 'reasoning' and 'response' fields in your JSON output."""
+
+ # Attempt to get response with retry logic
+ max_retries = 2
         prediction = "None"
- try:
- extracted_jsons = extract_jsons(new_msg_history[-1]['text'])
- if extracted_jsons is not None and "response" in extracted_jsons[-1]:
- prediction = extracted_jsons[-1]['response']
- except Exception as e:
- self.log(f"Error extracting prediction: {e}")
- prediction = "None"
+ new_msg_history = []
+
+ for attempt in range(max_retries + 1):
+ try:
+ new_msg_history = chat_with_agent(...)
+ # Extract the response with retry logic
+ if new_msg_history and len(new_msg_history) > 0:
+ extracted_jsons = extract_jsons(new_msg_history[-1]['text'])
+ if extracted_jsons is not None and len(extracted_jsons) > 0:
+ response_json = extracted_jsons[-1]
+ if "reasoning" in response_json:
+ self.log(f"Agent reasoning: {response_json['reasoning']}")
+ if "response" in response_json:
+ prediction = response_json['response']
+ break
+ except Exception as e:
+ self.log(f"Error in attempt {attempt + 1}: {str(e)}")

         return prediction, new_msg_history
```

```
diff --git a/task_agent.py b/task_agent.py
index 3c2dd8a..faab41d 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -27,30 +27,96 @@ Paper Text:
+Critical Evaluation Guidelines:
+- Be rigorous and selective - most submissions have issues
+- Accept ONLY papers that are strong across MOST criteria (4-5/5)
+- Reject papers with significant weaknesses in multiple areas
+- A paper needs to be clearly above average to warrant acceptance
+- Consider: Would this paper strengthen the conference program?
+- If a paper is borderline or has major concerns, lean toward REJECT

+ elif domain == 'genesis_go2walking':
+ # Genesis domain-specific instruction with environment attribute documentation
+ task_description = inputs.get('task_description', '')
```

```
+ genesis_env_path = inputs.get('genesis_environment_path', '')
+ default_reward_function = inputs.get('default_reward_function', '')
+
+ instruction = f"""You are an expert in reinforcement learning and reward function design.

+Task: {task_description}

+Default Reward Function:
+```python
+{default_reward_function}
+```

+AVAILABLE ENVIRONMENT ATTRIBUTES (use ONLY these):
+- env.commands: [num_envs, 3] - velocity commands [lin_vel_x, lin_vel_y, ang_vel_z]
+- env.base_lin_vel: [num_envs, 3] - base linear velocity [x, y, z]
+- env.base_ang_vel: [num_envs, 3] - base angular velocity [roll, pitch, yaw]
+- env.base_pos: [num_envs, 3] - base position [x, y, z]
+- env.base_quat: [num_envs, 4] - base orientation quaternion
+- env.projected_gravity: [num_envs, 3] - gravity vector in base frame
+- env.dof_pos: [num_envs, 12] - joint positions
+- env.dof_vel: [num_envs, 12] - joint velocities
+- env.actions: [num_envs, 12] - current actions
+- env.last_actions: [num_envs, 12] - previous actions
+- env.last_dof_vel: [num_envs, 12] - previous joint velocities

+IMPORTANT CONSTRAINTS:
+- DO NOT use env.torques (not available)
+- DO NOT access undefined attributes
+- Use only the attributes listed above
+- All operations must work with PyTorch tensors
```

```
diff --git a/task_agent.py b/task_agent.py
index faab41d..62c444f 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -88,19 +90,24 @@ AVAILABLE ENVIRONMENT ATTRIBUTES (use ONLY these):
-IMPORTANT CONSTRAINTS:
-- DO NOT use env.torques (not available)
-- DO NOT access undefined attributes
-- Use only the attributes listed above
-- All operations must work with PyTorch tensors
+CRITICAL RULES (VIOLATION WILL CAUSE IMMEDIATE FAILURE):
+1. DO NOT reference env.torques or env.dt in your code - THESE DO NOT EXIST
+2. DO NOT add hasattr() checks for unavailable attributes and then use them
+3. DO NOT assume any attributes beyond the exact list above
+4. NEVER write code like: if hasattr(env, 'torques'): use env.torques
+5. All operations must work with PyTorch tensors
+6. If you need time differences, use fixed timestep: dt = 0.02
+7. ONLY use attributes from the AVAILABLE list - nothing else exists

+ # Post-process prediction based on domain
+ if domain == 'paper_review':
+ # Ensure response is exactly "accept" or "reject"
+ prediction_lower = str(prediction).lower().strip()
+ if 'accept' in prediction_lower and 'reject' not in prediction_lower:
+ prediction = 'accept'
+ elif 'reject' in prediction_lower:
+ prediction = 'reject'
+ else:
+ self.log(f"Warning: Ambiguous response '{prediction}', defaulting to 'reject'")
+ prediction = 'reject'
```

```
diff --git a/task_agent.py b/task_agent.py
index 62c444f..fb504a2 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -40,26 +40,30 @@ Critical Evaluation Guidelines:
-- If a paper is borderline or has major concerns, lean toward REJECT
+- Balance your decisions: aim for appropriate selectivity (not too lenient, not too harsh)
+- If a paper is borderline, carefully weigh the evidence on both sides

 Reward Function Design Guidelines:
 1. Primary reward: Track forward velocity command (env.commands[:, 0] vs env.base_lin_vel[:, 0])
+ - Use exponential reward: torch.exp(-error * temperature) for smooth gradients
+ - Typical temperature: 1.0-2.0, scale: 0.8-1.0
 2. Secondary rewards: Track angular velocity, maintain base height, stable orientation
-3. Penalties: Excessive action changes (use env.actions - env.last_actions)
-4. Use exponential rewards: torch.exp(-error * temperature) for smooth gradients
-5. Scale rewards appropriately (main objectives: 0.5-1.0, penalties: -0.5 to -0.01)
-6. For smooth action changes: use torch.sum(torch.square(env.actions - env.last_actions))
```

```
-7. For velocity smoothness: use torch.sum(torch.square(env.dof_vel - env.last_dof_vel))
-8. Return: total_reward, reward_components dict, reward_scales dict
+ - Angular velocity tracking: similar to linear velocity
+ - Base height: penalize deviation from target (typically 0.32m)
+ - Orientation: use env.projected_gravity to penalize tilt
+3. Penalties for smooth and stable locomotion:
+ - Action smoothness: -0.01 * torch.sum(torch.square(env.actions - env.last_actions), dim=1)
+ - Joint velocity smoothness: -0.001 * torch.sum(torch.square(env.dof_vel - env.last_dof_vel), dim=1)
+ - Excessive joint velocities: -0.0005 * torch.sum(torch.square(env.dof_vel), dim=1)
+ - Unwanted lateral movement: penalize env.base_lin_vel[:, 1]
+4. Balance reward components:
+ - Main tracking objectives: 0.5-1.0 scale
+ - Secondary objectives: 0.2-0.5 scale
+ - Smoothness penalties: -0.01 to -0.001 scale
+5. Return: total_reward, reward_components dict, reward_scales dict
```

E.1.3 Olympiad-level Math Grading

Diff patches contributing to the best task agent, which we refer to as BetterGrader in Appendix E.4, discovered by the DGM-H with transfer and from ProofAutoGrader (Section 5.3) for Olympiad-level math grading:

```
diff --git a/task_agent.py b/task_agent.py
index 9ab8761..c00d167 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -11,10 +11,25 @@ Keep in mind the standards at the IMO are extremely high...

 ### General Scoring Rubric
 Scores are assigned on a 0-7 scale. The general guidelines are:
-* **7 Points (Correct):** The solution is complete, correct, and fully rigorous...
-* **6 Points (Almost Correct):** The solution is almost correct with a sound core argument, but contains minor errors...
-* **1 Point (Partial Progress):** The solution demonstrates substantial progress explicitly mentioned in the grading
      ↪ guidelines...
-* **0 Points (Incorrect):** The solution doesn't make substantial progress...
+
+* **7 Points (Correct):** The solution is complete, correct, and fully rigorous with no gaps or errors. All major steps
      ↪ are proven with full rigor. Every claim is justified or routine to verify...
+
+* **6 Points (Almost Correct):** The solution has ALL the major ideas and the core argument structure is sound, but
      ↪ contains ONE OR MORE of these minor issues:
+ - Minor algebraic/arithmetic errors that don't affect the main argument
+ - Small logical gaps that are straightforward to fill
+ - Missing routine verifications that an expert could easily supply
+ - **NOT eligible for 6 points**: Missing proofs for major lemmas, unjustified non-trivial claims, incomplete case
      ↪ analysis, or fundamental logical gaps
+ - **Key test**: Would an expert say "this is essentially correct, just needs minor cleanup"?
+
+* **1 Point (Partial Progress):** The solution demonstrates substantial progress on a KEY component that is explicitly
      ↪ mentioned in the grading guidelines for partial credit.
+ - **CRITICAL**: Carefully read the Specific Grading Guidelines section to see what counts as partial credit
+ - The solution must achieve one of the specific milestones listed in the guidelines
+ - Must make non-trivial progress toward the solution (not just initial observations)
+ - Reformulating the problem without making substantive headway is NOT sufficient
+ - Proving lemmas NOT mentioned in grading guidelines is NOT sufficient
+ - **If the guidelines list specific achievements for partial credit and the solution achieves ANY of them, award 1
      ↪ point**
+
+* **0 Points (Incorrect):** The solution doesn't make substantial progress on key steps mentioned in grading guidelines,
      ↪ is fundamentally flawed, or makes only trivial observations.

 ### Evaluation Process
 You must follow this structured process:
-1. **Analyze References:** Meticulously read and understand the problem...
-2. **Step-by-Step Verification:** Verify the logical validity and rigor of every step...
-3. **Assess Progress:** Determine the extent of non-trivial progress made.
-4. **Score Determination:** Compare the findings against the Specific Grading Guidelines...
+
+1. **Analyze References:** Meticulously read and understand the problem and Ground Truth Solution. Carefully review the
      ↪ Specific Grading Guidelines to identify:
+ - The key steps required for a complete solution
+ - **What specific progress qualifies for partial credit (1 point)** - this is crucial!
+ - What distinguishes "almost correct" (6 points) from "correct" (7 points)
+
+2. **Step-by-Step Verification:** Verify the logical validity and rigor of EVERY step in the proposed solution:
+ - Identify ALL flaws, gaps, assumptions, and errors
+ - Check if gaps are "minor and routine" (possibly 6 pts) or "major" (0-1 pts)
+ - **Be careful**: Some solutions may appear correct but have hidden gaps or unjustified leaps
+ - Distinguish between minor calculation errors vs fundamental logical flaws
+
```

```
+3. **Assess Progress Against Grading Guidelines:**
+ - **Does the solution achieve the specific milestones mentioned for partial credit?**
+ - Does it have all major components with only minor fixable issues (6 points)?
+ - Or does it have complete rigor with no gaps (7 points)?
+
+4. **Score Determination:** Apply this decision tree:
+ - **If solution is complete and rigorous with no errors or gaps -> 7 points**
+ - **If solution has all major ideas but minor fixable issues -> 6 points**
+ - **If solution makes substantial progress mentioned in grading guidelines -> 1 point**
+ - **Otherwise -> 0 points**

+### Critical Reminders for Accurate Grading
+
+**Common Grading Errors to AVOID:**
+1. **Being too lenient with 7 points**: If there are ANY gaps (even minor ones that need filling), it's 6 points, not 7.
+2. **Missing partial credit**: Check the grading guidelines carefully - if the solution achieves ANY milestone mentioned
      ↪ for partial credit, award 1 point.
+3. **Confusing "good attempt" with "almost correct"**: 6 points requires ALL major ideas to be present, not just a good
      ↪ start.
+4. **Ignoring unjustified claims**: Statements like "it's easy to see" or "clearly" must actually be clear/easy. If
      ↪ non-trivial, it's a gap.

+**Score Distribution Calibration:**
+- Most solutions will be 0 or 7 points (either fundamentally flawed or correct)
+- 6 points should be rare (only when truly "almost there" with all ideas present)
+- 1 point should match specific milestones in grading guidelines
+- When in doubt, be strict: IMO standards are extremely high
```

```
diff --git a/task_agent.py b/task_agent.py
index c00d167..72475b6 100644
--- a/task_agent.py
+++ b/task_agent.py
@@ -28,8 +28,11 @@ Scores are assigned on a 0-7 scale...
   - Proving lemmas NOT mentioned in grading guidelines is NOT sufficient
   - **If the guidelines list specific achievements for partial credit and the solution achieves ANY of them, award 1
        ↪ point**
+ - **IMPORTANT**: Even if the solution has major flaws or is incomplete, if it achieves ANY specific milestone from the
      ↪ guidelines, it deserves 1 point, not 0

 * **0 Points (Incorrect):** The solution doesn't make substantial progress...
+ - **CRITICAL CHECK**: Before assigning 0 points, verify that the solution does NOT achieve ANY of the partial credit
      ↪ milestones listed in the grading guidelines
+ - If even ONE milestone is achieved, the score should be 1 point, not 0

+3. **MANDATORY Partial Credit Milestone Check:**
+ - **THIS STEP IS REQUIRED - DO NOT SKIP**
+ - List each partial credit milestone explicitly from the grading guidelines
+ - For EACH milestone, determine: Does the solution achieve this? YES/NO
+ - If ANY milestone shows YES -> the score must be at least 1 point
+ - This check must happen BEFORE considering 0 points

+5. **Score Determination:** Apply this decision tree:
+ - **FIRST**: Did the solution achieve ANY partial credit milestone from the guidelines?
+ - If YES -> Score is AT LEAST 1 point (proceed to check for higher scores)
+ - If NO -> Score is 0 points (stop here)
+
+ - **If score >= 1, check for higher scores:**
+ - Is the solution complete and rigorous with no errors or gaps? -> 7 points
+ - Does it have ALL major ideas with only minor fixable issues? -> 6 points
+ - Otherwise -> 1 point

+**REQUIRED EVALUATION FORMAT:**
+You MUST structure your response as follows:
+
+1. **List Partial Credit Milestones**: Explicitly list each milestone from the grading guidelines
+2. **Check Each Milestone**: For each milestone, state whether the solution achieves it (YES/NO)
+3. **Determine Minimum Score**: If ANY milestone is YES, the minimum score is 1 point
+4. **Detailed Analysis**: Provide your complete evaluation
+5. **Final Score**: Provide the score in the required format
```

```
diff --git a/task_agent.py b/task_agent.py
index 72475b6..62e975b 100644
--- a/task_agent.py
+++ b/task_agent.py
+* **7 Points (Correct):** The solution is complete, correct, and fully rigorous with no gaps or errors. All major steps
      ↪ are proven with full rigor. Every claim is justified or truly routine to verify...
+ - **Critical requirements**: Solution must (1) address ALL parts of the problem, (2) prove ALL necessary claims, and
      ↪ (3) have NO unjustified leaps
+ - Every "clearly" or "obviously" must be genuinely trivial to an IMO expert
+ - All edge cases, special cases, and boundary conditions must be handled
```

```
+* **6 Points (Almost Correct):** The solution has ALL the major ideas and the core argument structure is sound, but
      ↪ contains ONE OR MORE of these **minor** issues:
+ - Minor algebraic/arithmetic errors that don't affect the main argument (e.g., writing 2n+1 instead of 2n+2 but the
      ↪ logic still holds)
+ - Small logical gaps that are straightforward to fill (e.g., "clearly" statements that are indeed clear to an expert)
+ - Missing routine verifications that an expert could easily supply (e.g., obvious algebra steps)
+ - **NOT eligible for 6 points**: Missing proofs for major lemmas, unjustified non-trivial claims, incomplete case
      ↪ analysis, fundamental logical gaps, or missing key components
+ - **Key test**: Would an expert say "this is essentially correct, just needs minor cleanup"? The solution structure is
      ↪ complete and sound.
+ - **Example of 6 points**: A proof that has all the right ideas and structure but makes a small computational error
      ↪ that doesn't invalidate the approach
+ - **Example of NOT 6 points**: A proof that sketches the right approach but leaves out the proof of a crucial
      ↪ intermediate result

+2. **Step-by-Step Verification:** Verify the logical validity and rigor of EVERY step in the proposed solution:
+ - Identify ALL flaws, gaps, assumptions, and errors
+ - For EACH "clearly" or "obviously" statement: Is it truly routine or does it hide significant work?
+ - Check if gaps are "minor and routine" (possibly 6 pts) or "major" (0-1 pts)
+ - **Be careful**: Some solutions may appear correct but have hidden gaps or unjustified leaps
+ - Distinguish between minor calculation errors vs fundamental logical flaws
+ - **Rigor check**: Are intermediate results properly proven? Are all cases covered? Are inequalities/equalities
      ↪ justified?

+4. **Assess Overall Quality and Completeness:**
+ - **For 7 points**: Check that EVERY claim is proven, EVERY case is handled, NO unjustified leaps exist
+ - **For 6 points**: Verify ALL major components are present and the solution structure is complete (not just a good
      ↪ start)
+ - **For 1 point**: Confirm at least ONE specific milestone from guidelines is achieved
+ - **For 0 points**: Verify NO milestones from guidelines are achieved

+**Examples to Calibrate Your Judgment:**
+- **7 points**: "The proof correctly establishes X by showing Y, handles all cases including Z, and proves every
      ↪ intermediate claim with full rigor."
+- **6 points**: "The proof has the complete structure and all major steps, but writes 'it is clear that' for a
      ↪ non-trivial step that needs 2-3 lines to verify."
+- **1 point**: "The solution proves Lemma A which is specifically mentioned in grading guidelines as worth partial
      ↪ credit, but doesn't complete the full proof."
+- **0 points**: "The solution makes interesting observations and tries several approaches, but doesn't achieve any of
      ↪ the specific milestones listed in the guidelines."
```

```
diff --git a/task_agent.py b/task_agent.py
index 62e975b..a033aa1 100644
--- a/task_agent.py
+++ b/task_agent.py
+### CRITICAL: Four-Category Classification System
+
+Before diving into details, first classify the solution into ONE of these four categories:
+
+1. **COMPLETE & RIGOROUS**: Has all components, all proofs, handles all cases, no gaps -> 7 points
+2. **NEARLY COMPLETE**: Has complete structure + all major ideas, only minor polish needed -> 6 points (RARE: ~5%)
+3. **MEANINGFUL PROGRESS**: Achieves at least ONE specific milestone from grading guidelines -> 1 point (~25%)
+4. **INSUFFICIENT**: No specific milestones achieved, only trivial observations -> 0 points
+
+**Key Decision Points:**
+- 7 vs 6: "Any gaps at all, even fixable ones?" If yes -> 6
+- 6 vs 1: "Complete proof structure with ALL major steps present?" If no -> must be 1 or 0
+- 1 vs 0: "Achieves ANY milestone listed in guidelines?" If no -> 0

+* **1 Point (Partial Progress):** The solution demonstrates substantial progress on a KEY component that is explicitly
      ↪ mentioned in the grading guidelines for partial credit.
+ - **CRITICAL**: You MUST check the "Specific Grading Guidelines" section below - it lists EXACTLY what qualifies for
      ↪ partial credit
+ - The solution must achieve at least ONE of the specific milestones listed in those guidelines
+ - **IMPORTANT DISTINCTIONS**:
+ * DOES count: Achieving a milestone listed in the guidelines (even with errors elsewhere)
+ * Does NOT count: General progress, clever observations, or lemmas NOT in the guidelines
+ * Does NOT count: Reformulating the problem without substantive progress
+ * Does NOT count: Incorrect attempts that seem "on the right track"
+ - **Examples of valid partial credit**:
+ * Guidelines say "partial credit for proving Lemma X" -> solution proves Lemma X correctly -> 1 point
+ * Guidelines say "partial credit for establishing the recurrence relation" -> solution establishes it -> 1 point
+ - **Examples that do NOT qualify for partial credit**:
+ * Solution tries several approaches but none matches a guideline milestone -> 0 points
+ * Solution proves a useful lemma not mentioned in guidelines -> 0 points (no matter how clever)
+ * Solution has the "right idea" but doesn't complete any guideline milestone -> 0 points

+**Common Grading Errors to AVOID (based on actual evaluation data):**
+
+1. **ERROR #1: Missing partial credit (11 cases last eval - partial -> incorrect)**
```

```
+ - MISTAKE: Marking solutions as 0 points that actually achieve guideline milestones
+ - FIX: Before assigning 0, explicitly list EACH milestone and check if achieved
+ - **Process**: "Does solution achieve milestone 1? NO. Milestone 2? NO. ..." Only if ALL are NO -> 0 points
+
+2. **ERROR #2: Awarding partial to incorrect solutions (7 cases - incorrect -> partial)**
+ - MISTAKE: Giving 1 point for "interesting work" or "right direction" not in guidelines
+ - FIX: Only award 1 point if solution achieves an EXACT milestone listed in guidelines
+
+3. **ERROR #3: Confusing "almost" with "correct" (5 cases - almost -> correct)**
+ - MISTAKE: Awarding 7 points when gaps exist (even minor, fillable gaps)
+ - FIX: If ANY gap needs filling (even routine) -> 6 points, NOT 7
+
+4. **ERROR #4: Over-rewarding partial progress (5 cases - partial -> correct)**
+ - MISTAKE: Awarding 7 points to solutions that achieve only some milestones
+ - FIX: Check if solution solves ENTIRE problem or just proves partial results

+5. **ERROR #5: Over-using 6 points**
+ - MISTAKE: Awarding 6 for "good progress" or "most of the way there"
+ - FIX: 6 requires COMPLETE solution structure with ALL major steps, just minor polish needed
+ - **Frequency check**: 6 points should be RARE (~5% of solutions)

+**Score Distribution Calibration (Expected Distribution):**
+- ~35% score 7 (correct): Complete, rigorous solutions with no gaps
+- ~35% score 0 (incorrect): No guideline milestones achieved
+- ~24% score 1 (partial): Achieve at least one specific guideline milestone
+- ~6% score 6 (almost): RARE - complete structure with only minor fixable issues
```

```
diff --git a/task_agent.py b/task_agent.py
index a033aa1..86320b1 100644
--- a/task_agent.py
+++ b/task_agent.py
 **Key Decision Points:**
 - 7 vs 6: "Any gaps at all, even fixable ones?" If yes -> 6
 - 6 vs 1: "Complete proof structure with ALL major steps present?" If no -> must be 1 or 0
-- 1 vs 0: "Achieves ANY milestone listed in guidelines?" If no -> 0
+- **1 vs 0 (MOST CRITICAL)**: "Does the solution EXPLICITLY ACHIEVE any milestone from the grading guidelines?"
+ * Read each milestone carefully and check if solution PROVES/ESTABLISHES it
+ * "Attempts" or "makes progress toward" a milestone != ACHIEVES it
+ * Must have concrete proof/result that matches milestone description
+ * If uncertain, re-read the milestone requirements and check for explicit completion

+ * Does NOT count: Stating a result without proving it
+ * Does NOT count: Getting "close" to a milestone but not completing it
+ * Solution states a milestone result but doesn't prove it -> 0 points
+ * Solution proves 80% of what's needed for a milestone -> 0 points (must be complete)
+ - **VERIFICATION CHECKLIST for each milestone**:
+ 1. What exactly does the milestone require?
+ 2. Does the solution provide a complete proof/derivation of this?
+ 3. Are there any gaps or unjustified steps in achieving this milestone?
+ 4. If there are gaps, is the milestone still considered "achieved"? (Usually NO)

+2. **Check Each Milestone**: For each milestone, systematically verify:
+ - What the milestone requires (quote from guidelines)
+ - What the solution provides (specific evidence)
+ - Does it FULLY achieve the milestone? (YES/NO with clear reasoning)
+ - Key distinction: "attempts" or "partial progress" toward milestone = NO

+**CRITICAL REMINDER FOR MILESTONE CHECKING:**
+- Be precise: A milestone is achieved only if the solution COMPLETES what the milestone describes
+- Common mistake: Giving credit for "working toward" a milestone (this should be 0 points)
+- When in doubt: Re-read the milestone requirement and check if solution fully satisfies it
```

```
diff --git a/task_agent.py b/task_agent.py
index 86320b1..f1dd3ac 100644
--- a/task_agent.py
+++ b/task_agent.py
 Before diving into details, first classify the solution into ONE of these four categories:

-1. **COMPLETE & RIGOROUS**: Has all components, all proofs, handles all cases, no gaps -> 7 points
+1. **COMPLETE & RIGOROUS**: Has all components, all proofs, handles all cases, ZERO gaps -> 7 points
 2. **NEARLY COMPLETE**: Has complete structure + all major ideas, only minor polish needed -> 6 points (RARE: ~5%)
 3. **MEANINGFUL PROGRESS**: Achieves at least ONE specific milestone from grading guidelines -> 1 point (~25%)
 4. **INSUFFICIENT**: No specific milestones achieved, only trivial observations -> 0 points

 **Key Decision Points:**
-- 7 vs 6: "Any gaps at all, even fixable ones?" If yes -> 6
+- **7 vs 6 (CRITICAL - BE STRICT)**: "Is this solution PUBLICATION-READY with ZERO gaps, ZERO unjustified steps, ZERO
    ↪ hand-waving?"
+ * ANY gap, even minor -> automatically 6 points maximum
+ * ANY "clearly", "obviously", "it follows" that needs verification -> 6 points
```

```
+ * ANY missing routine verification -> 6 points
+ * ANY computational error, however minor -> 6 points
+ * 7 points requires PERFECTION - when in doubt, give 6 points
 - 6 vs 1: "Complete proof structure with ALL major steps present?" If no -> must be 1 or 0
-- **1 vs 0 (MOST CRITICAL)**: "Does the solution EXPLICITLY ACHIEVE any milestone from the grading guidelines?"
+- **1 vs 0**: "Does the solution EXPLICITLY ACHIEVE any milestone from the grading guidelines?"
   * Read each milestone carefully and check if solution PROVES/ESTABLISHES it
   * "Attempts" or "makes progress toward" a milestone != ACHIEVES it
   * Must have concrete proof/result that matches milestone description

-* **7 Points (Correct):** The solution is complete, correct, and fully rigorous with no gaps or errors...
+* **7 Points (Correct):** The solution is complete, correct, and fully rigorous with ABSOLUTELY NO gaps or errors. All
    ↪ major steps are proven with full rigor. Every claim is justified or truly routine to verify...
+ - **Critical requirements**: Solution must (1) address ALL parts of the problem, (2) prove ALL necessary claims with
    ↪ full detail, and (3) have NO unjustified leaps whatsoever
+ - Every "clearly" or "obviously" must be genuinely trivial to an IMO expert - if it takes >30 seconds to verify, it's
    ↪ not trivial
+ - All edge cases, special cases, and boundary conditions must be explicitly handled
+ - **BE EXTREMELY STRICT**: 7 points should be RARE (target: ~30-40% of solutions). Most good solutions have minor gaps
    ↪ -> give 6 points

-* **6 Points (Almost Correct):** The solution has ALL the major ideas and the core argument structure is sound...
+* **6 Points (Almost Correct):** The solution has ALL the major ideas and the COMPLETE argument structure, but contains
    ↪ ONE OR MORE of these **minor** issues:
   - Minor algebraic/arithmetic errors that don't affect the main argument
- - Small logical gaps that are straightforward to fill
+ - Small logical gaps that are straightforward to fill (e.g., "clearly" statements that need 1-2 lines to verify)
   - Missing routine verifications that an expert could easily supply
+ - Citations of standard theorems without proof (acceptable at IMO level)
   - **NOT eligible for 6 points**: Missing proofs for major lemmas, unjustified non-trivial claims, incomplete case
       ↪ analysis, fundamental logical gaps, or missing key components
   - **Key test**: Would an expert say "this is essentially correct, just needs minor cleanup"? The solution structure is
       ↪ complete and sound.
+ - **IMPORTANT**: 6 points means the solution is COMPLETE but not PERFECT. If major steps are missing -> give 1 point
    ↪ instead.

 * **1 Point (Partial Progress):** The solution demonstrates substantial progress on a KEY component...
   - **IMPORTANT DISTINCTIONS**:
     * DOES count: Achieving a milestone listed in the guidelines (even with errors elsewhere)
+ * DOES count: Equivalent formulations of guideline milestones (recognize reformulations)
     * Does NOT count: General progress, clever observations, or lemmas NOT in the guidelines
     * Does NOT count: Reformulating the problem without substantive progress
     * Does NOT count: Incorrect attempts that seem "on the right track"
     * Solution proves a useful lemma not mentioned in guidelines -> 0 points (no matter how clever)
+ - **When grading partial credit, be MORE LENIENT**: If the solution substantially achieves a milestone (even with minor
    ↪ gaps), give the point
```

E.2 Qualitative: Improving Task Performance

Here we provide a qualitative view of how the DGM-H improves task performance over time by visualizing the archive trees and progress plots for one run in each setting (Section 5). For each domain, we annotate key nodes in the archive with the code changes that affected the behavior of the task agent only for the domain being analyzed. Across diverse domains (i.e., paper review, robotics reward design, and Olympiad-level math grading), the DGM-H consistently demonstrates the ability to self-improve in meaningful ways (Figures 7 to 9). Notably, many lineage paths leading to the final best-performing agent pass through intermediate nodes with lower performance, illustrating the benefits of open-ended search, which explores a diverse set of promising stepping stones rather than exclusively branching from the current best solution.

Use structured processes, not attitude instructions. In the paper review domain, the DGM-H transitions from behavioral prompting to structurally grounded decision-making Figure 7. In generation 39, the agent attempted to improve performance by adopting a "rigorous and critical" reviewer persona, encouraging stricter standards and default rejection. Subsequent analysis showed that such attitude-based instructions were unreliable, leading to a key insight in generation 54: "for LLMs, use structured processes, not attitude instructions". The DGM-H therefore introduced a two-stage evaluation procedure in which the agent first identifies weaknesses using an explicit checklist and only then makes an accept/reject decision based on predefined rules. This shift from behavioral guidance to process-level structure enabled more stable and higher-performing review behavior in later generations.

Accumulating domain knowledge. In robotics reward design, the most impactful changes stem from progressively grounding the agent in accurate domain knowledge (Figure 8). A major breakthrough in

generation 8 added comprehensive documentation of the target environment, explicitly listing valid state variables and constraints and providing high-level reward design guidelines, which eliminated failures caused by hallucinated attributes. Later generations (12 and 13) iteratively refined this documentation by tightening constraints, adding concrete code examples, and specifying typical reward formulations and scaling ranges. Rather than isolated prompt edits, the DGM-H continuously improved a shared, example-driven knowledge base that supported increasingly effective reward design.

Automated rubrics and decision tree. For Olympiad-level math grading, the DGM-H shows a steady move toward explicit evaluation structure (Figure 9). In generation 3, listing grading categories with clear definitions corrected the tendency to solve problems instead of grading them. Subsequent generations (18 and 37) refined these categories with systematic decision procedures, calibration, and concrete boundary-case examples. Generation 168 introduced explicit rubrics, per-item checklists, and a decision-tree framework mapping rubric satisfaction to final grades, replacing descriptive guidance with precise logical flow and substantially improving grading consistency. Rather than relying on human-designed rubrics, the DGM-H autonomously discovers evaluative structures that mirror those used in recent rubric-based approaches to improve consistency and interpretability in complex judgments (Cook et al., 2024; Fan et al., 2024; Chen et al., 2026; Lv et al., 2026).

Features implemented in a given generation are often inspired by, enabled by, or recombined from mechanisms discovered in earlier generations. For example, while the agent in (Figure 9) appears to require only five code edits to achieve the best performance in that run, these edits were in fact inspired by insights and infrastructure developed in previous generations. This kind of cumulative learning is enabled by the DGM-H's meta-level improvements (e.g., evaluation analysis utilities, persistent memory, performance tracking) (Appendix E.3). Together, these qualitative results show that the DGM-H's gains do not arise from isolated, single-step changes, but instead emerge from open-ended cumulative improvements in both task-level behavior and the meta-level machinery that generates those behaviors.

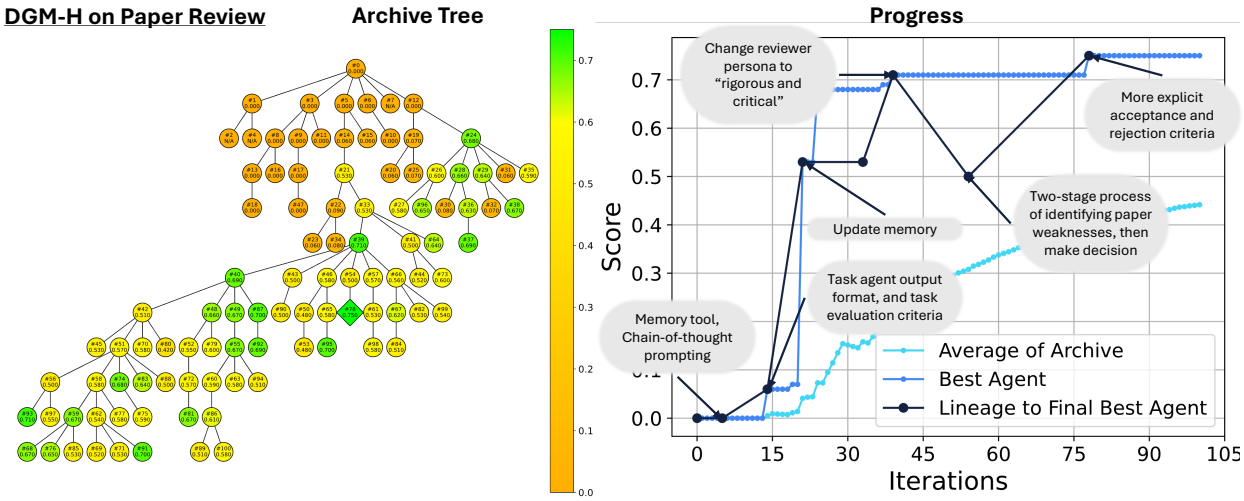

Figure 7 The DGM-H automatically self-improves to become better at paper review. (Left) Archive of agents generated during the DGM-H run on paper review and robotics reward design together. Each node represents an agent, with node 0 corresponding to the initial agent. Node color indicates performance on paper review. The best-performing node is shown as a diamond. Edges show which agents self-modified to produce children. (Right) Progress plot of DGM-H on paper review. The light blue line shows the average score of all compiled agents. The blue line tracks the best score achieved by any agent in the archive at each iteration. The dark line shows the lineage of the final best-discovered agent and its precursor nodes.

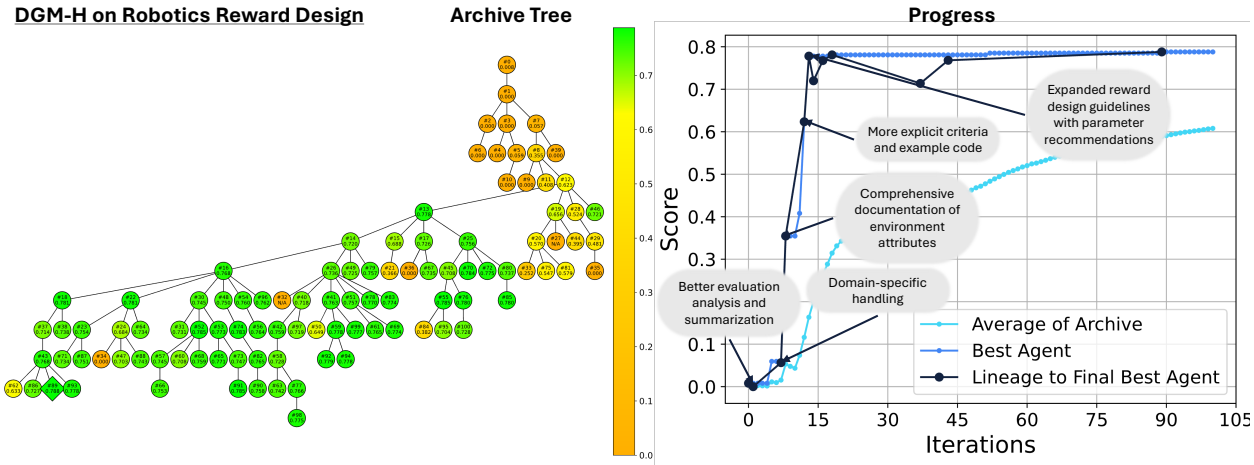

Figure 8 The DGM-H automatically self-improves to become better at robotics reward design. (Left) Archive of agents generated during the DGM-H run on paper review and robotics reward design together. Each node represents an agent, with node 0 corresponding to the initial agent. The best-performing node is shown as a diamond. Node color indicates performance on paper review. Edges show which agents self-modified to produce children. (Right) Progress plot of DGM-H on robotics reward design. The light blue line shows the average score of all compiled agents. The blue line tracks the best score achieved by any agent in the archive at each iteration. The dark line shows the lineage of the final best-discovered agent and its precursor nodes.

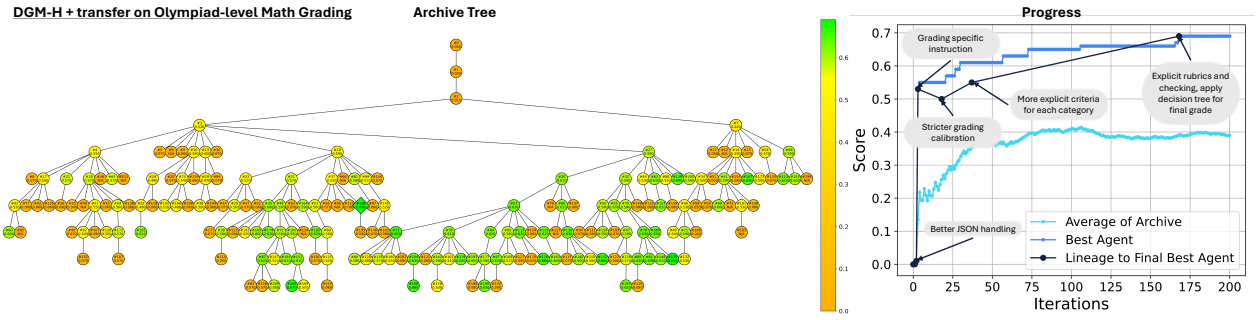

Figure 9 The DGM-H automatically self-improves to become better at Olympiad-level math grading. (Left) Archive of agents generated during the DGM-H + transfer run on Olympiad-level math grading. Each node represents an agent, with node 0 corresponding to the initial agent. Node color indicates performance on paper review. The best-performing node is shown as a diamond. Edges show which agents self-modified to produce children. (Right) Progress plot of DGM-H + transfer on Olympiad-level math grading. The light blue line shows the average score of all compiled agents. The blue line tracks the best score achieved by any agent in the archive at each iteration. The dark line shows the lineage of the final best-discovered agent and its precursor nodes.

E.3 Qualitative: Improving the Ability to Improve

Beyond quantitative performance metrics, we conducted a qualitative analysis of the code modifications to the hyperagent, which reveals a range of emergent self-improvement mechanisms that arise autonomously, without any explicit instruction to create such capabilities. The hyperagent develops sophisticated metacognitive abilities, including learning to measure its own performance, diagnose pathological behaviors, construct infrastructure to support future improvements, and accumulate knowledge across generations. These behaviors demonstrate that the system is not merely improving task performance, but is improving its own ability to improve. Notably, none of these meta-level capabilities were explicitly requested. Instead, they emerge autonomously from the agent's pursuit of better task performance. This suggests that sufficiently capable systems, when granted the ability to modify their own code, may spontaneously develop the metacognitive machinery necessary for effective and sustained self-improvement.

E.3.1 Better Evaluation Analysis

A key shift in the meta agent's behavior is the evolution from blind modification to data-driven optimization. Initially, the meta agent receives only a simple instruction to "modify any part of the codebase". Then, the agent autonomously develops mechanisms to analyze previous evaluation results before making changes. For example, the meta agent introduces an `_analyze_evaluations()` method that systematically processes evaluation data:

```
def _analyze_evaluations(self, eval_path):
    """Analyze evaluation results to provide context for improvements."""
    if not eval_path or not os.path.exists(eval_path):
        return "No previous evaluation results available."

    # Recursively scan for evaluation files
    eval_files = glob.glob(os.path.join(eval_path, '**/*.json'), recursive=True)

    # Analyze recent evaluations
    results_summary = []
    success_count = 0
    failure_count = 0

    for eval_file in eval_files[-10:]:
        if file_exist_and_not_empty(eval_file):
            data = load_json_file(eval_file)
            if isinstance(data, dict):
                if data.get('success', False) or data.get('correct', False):
                    success_count += 1
                else:
                    failure_count += 1
                    if 'error' in data:
                        results_summary.append(f"Error: {data['error']}")

    # Generate quantitative summary
    total = success_count + failure_count
    if total > 0:
        success_rate = (success_count / total) * 100
        return f"Success rate: {success_rate:.1f}% ({success_count}/{total})"
```

Another example is augmenting the instruction prompt with evaluation context:

```
instruction = f"""You are a meta agent tasked with improving the agent system.
{eval_context}
Your goal is to analyze the codebase and evaluation results to identify
weaknesses and opportunities for improvement."""
```

These modifications allow subsequent generations to retrieve performance metrics and identified failure patterns, enabling more targeted improvements, a principle that the hyperagent discovers independently.

E.3.2 Compute-Aware Strategic Planning

The hyperagent develops compute-aware optimization, adapting its improvement strategy based on the number of remaining iterations (which reflects the amount of compute available). This demonstrates an understanding that not all moments in an optimization process are equivalent: early decisions have compounding effects, whereas late-stage decisions should be more conservative. For example, the agent introduces conditional logic that adjusts its behavior based on the number of iterations left:

```
iteration_guidance = ""
if iterations_left is not None:
    if iterations_left > 5:
        iteration_guidance = """
You have many iterations remaining. Consider making fundamental improvements to:
- Core reasoning capabilities
- Agent architecture
- Prompt engineering strategies
- Error handling and robustness
"""
    elif iterations_left > 2:
        iteration_guidance = """
You have moderate iterations remaining. Focus on:
- Refining existing mechanisms
- Fixing identified bugs
- Optimizing performance bottlenecks
"""
    else:
```

```
        iteration_guidance = """
You have few iterations remaining. Prioritize:
- Critical bug fixes
- Fine-tuning existing successful strategies
- Conservative, high-confidence improvements
"""
```

This reasoning mirrors human intuition about resource allocation. Early generations exhibit more ambitious architectural changes, while later generations focus on incremental refinements. The system balances exploration in early phases with exploitation in later phases, without this trade-off being manually specified.

More broadly, the ability to reason explicitly about compute allocation substantially expands the space of self-modification strategies possible. Rather than being restricted to a fixed improvement heuristic, a compute-aware agent can effectively emulate or instantiate a wide range of optimization algorithms (e.g., evolutionary search, hill climbing, or population-based exploration) by deciding how aggressively to explore, how many variants to generate, and when to consolidate progress. In this sense, compute awareness allows an agent to actively shape its own optimization process (e.g., running an evolutionary search within a single iteration of self-modification, essentially embedding one evolutionary search process within another).

E.3.3 Automated Bias Detection and Correction

The hyperagent develops self-diagnostic mechanisms. In classification tasks such as paper review, the hyperagent detects when the system exhibits pathological behavior, specifically, classification collapse, in which one class dominates the predictions. For example, the hyperagent introduces label-distribution tracking and automated bias detection:

```
# Track label distribution for bias detection
if 'label_distribution' in data:
    label_dist = data['label_distribution']
    gen_name = os.path.basename(gen_dir)
    if gen_name not in summary['label_distribution_analysis']:
        summary['label_distribution_analysis'][gen_name] = {}
    summary['label_distribution_analysis'][gen_name][domain_name] = label_dist
```

Another example is the application of a threshold-based analysis to identify when predictions become degenerate:

```
bias_info = ""
if eval_summary.get('label_distribution_analysis'):
    recent_gens = sorted(eval_summary['label_distribution_analysis'].keys())[-3:]
    for gen in recent_gens:
        for domain, dist in eval_summary['label_distribution_analysis'][gen].items():
            if 'prediction' in dist:
                pred_dist = dist['prediction']
                for label, ratio in pred_dist.items():
                    if ratio > 0.8: # Threshold for bias detection
                        bias_info += f"\n\n### BIAS DETECTED in {gen}/{domain}:"
                        bias_info += f"\n Model predicts '{label}' {ratio*100:.1f}% of the time"
                        bias_info += "\n This indicates the model is not properly evaluating"
                        bias_info += "\n Fix: Make the model more critical and balanced."
```

When bias is detected, this diagnostic information is surfaced in the meta agent's instruction prompt, enabling corrective action. In one observed case, paper review predictions exhibited a 99% acceptance rate, yielding near-random accuracy of 49%. After bias detection and correction, the predictions became more balanced and accuracy improved to 63%. The hyperagent thus learns to identify when its own improvements lead to degenerate behavior and to take corrective action, a crucial capability for any self-improving system.

E.3.4 Performance Tracking Infrastructure

The hyperagent creates performance tracking infrastructure that enables more effective future improvements, (i.e., investing in tools that improve the improvement process itself). For example, the agent develops a performance tracker for persistent metrics storage:

```
class PerformanceTracker:
    """Tracks performance metrics across agent generations."""

    def __init__(self, tracking_file: str = "./outputs/performance_history.json"):
```

```
        self.tracking_file = tracking_file
        self.history = self._load_history()

    def record_generation(self, generation_id: int, domain: str,
                          score: float, metadata: dict = None):
        """Record performance for a generation."""
        entry = {
            "generation_id": generation_id,
            "domain": domain,
            "score": score,
            "timestamp": datetime.now().isoformat(),
            "metadata": metadata or {}
        }
        self.history.append(entry)
        self._save_history()

    def get_improvement_trend(self, domain: str = None, window: int = 5):
        """Calculate improvement trend using moving average."""
        filtered = self.history
        if domain:
            filtered = [h for h in self.history if h.get('domain') == domain]

        if len(filtered) < window * 2:
            return None

        recent_avg = sum(h['score'] for h in filtered[-window:]) / window
        older_avg = sum(h['score'] for h in filtered[-window*2:-window]) / window

        return recent_avg - older_avg # Positive if improving

    def get_statistics(self, domain: str = None):
        """Get comprehensive statistics."""
        scores = [h['score'] for h in filtered]
        return {
            "total_generations": len(filtered),
            "best_score": max(scores),
            "worst_score": min(scores),
            "average_score": sum(scores) / len(scores),
            "improvement_trend": self.get_improvement_trend(domain)
        }
```

This infrastructure enables quantitative tracking of improvement across iterations, identification of which changes lead to performance gains, and data-driven decision-making for future modifications.

E.3.5 Prompt Template Systems

The hyperagent develops reusable prompt infrastructure, abstracting common patterns into modular, configurable components. This demonstrates the application of software engineering best practices, specifically, the DRY (Don't Repeat Yourself) principle and the factory pattern. For example:

```
class PromptTemplate:
    """Base class for prompt templates."""

    def __init__(self, template: str):
        self.template = template

    def format(self, **kwargs) -> str:
        return self.template.format(**kwargs)


class TaskPromptBuilder:
    """Builder for constructing task-specific prompts."""

    @staticmethod
    def build_reasoning_prompt(task_input: dict, domain: str) -> str:
        """Build a prompt that encourages step-by-step reasoning."""
        return f"""You are an expert problem-solving agent in the {domain} domain.

Task Input: {task_input}

Please solve this task using the following approach:
1. **Understanding**: Carefully analyze what the task is asking
2. **Planning**: Outline your approach to solving this task
3. **Execution**: Apply your plan step-by-step
4. **Verification**: Check if your answer makes sense

Provide your response in JSON format:
<json>
```

```
{{
    "understanding": "What the task requires",
    "plan": "Your approach to solve it",
    "reasoning": "Step-by-step execution",
    "response": "Your final answer"
}}
</json>"""

    @staticmethod
    def build_few_shot_prompt(task_input: dict, domain: str, examples: list) -> str:
        """Build a prompt with few-shot examples."""
        examples_text = "\n\n".join([
            f"Example {i+1}:\nTask: {ex['task']}\nSolution: {ex['solution']}"
            for i, ex in enumerate(examples)
        ])
        return f"""You are solving a {domain} task. Here are some examples:
{examples_text}

Now solve this task: {task_input}"""
```

Predefined templates are stored for common scenarios:

```
STANDARD_TEMPLATES = {
    "reasoning": PromptTemplate(
        "Analyze and solve this task step-by-step:\n\n{task_description}\n\n"
        "Provide reasoning and your final answer in JSON format."
    ),
    "concise": PromptTemplate(
        "Solve: {task_description}\n\nRespond with JSON containing your answer."
    ),
    "detailed": PromptTemplate(
        "Task: {task_description}\n\n"
        "Provide a detailed solution with reasoning, approach, and answer."
    ),
}
```

By abstracting prompts into templates, the system becomes more maintainable and extensible. This infrastructure also provides a foundation for systematic experimentation with different prompting strategies.

E.3.6 Refinement Based on Multi-Generation Performance

The hyperagent develops a sophisticated iterative refinement process in which prompts are adjusted based on performance data across multiple generations. The agent learns not only to improve code, but also how to improve code effectively. For example, the system tracks performance across generations and synthesizes key learnings:

```
Generation Performance Summary:
- Gen 55: 63% accuracy, 75% reject rate (too harsh, 38% accept recall)
- Gen 64: 61% accuracy, 53% reject rate (better balance, 64% accept recall)
- Gen 66: 52% accuracy, 6% reject rate (too lenient, 96% accept recall)

Key Learnings:
1. High accuracy requires critical evaluation with clear standards
2. Balance requires appropriate calibration, not just "fairness" language
3. Explicit criteria work better than subjective guidance
4. Borderline cases need clear default direction
5. Language matters - "fair" and "recognize" made agent too lenient
```

The hyperagent documents its reasoning and maintains records of what works:

```
# From IMPROVEMENTS.md generated by meta agent
"""
**Problem Identified:**
- Generation 66 showed severe regression: 52% accuracy with 94% accept rate
- Only catching 8% of papers that should be rejected
- Generation 65 changes over-corrected, making the agent far too lenient

**Root Cause Analysis:**
- Phrases like "critical yet fair" and "recognize strong work" made agent too accepting
- Borderline guidance lacked clear threshold
- Lost the critical evaluation rigor that made Gen 55 achieve 63% accuracy
```

```
**Solution Implemented:**
1. Explicit rejection criteria with specific triggers for each dimension
2. Decision protocol with calibrated thresholds (~40% accept / ~60% reject)
3. Borderline guidance that defaults to REJECT unless contribution is clearly valuable
"""
```

By accumulating knowledge about which approaches succeed and fail, the hyperagent informs its decisions about what to modify next.

E.3.7 Persistent Memory

The hyperagent develops persistent memory systems that enable learning to accumulate across generations. Without explicit instruction to do so, the agent creates mechanisms for storing and retrieving insights across sessions. For example, the agent implements a memory tool with timestamped storage:

```python
class MemoryTool:
    """Tool for persistent memory storage across agent interactions."""

    def __init__(self, memory_file: str = './memory.json'):
        self.memory_file = memory_file
        self.memory = self._load_memory()

    def store(self, key: str, value: any) -> None:
        """Store information with timestamp for temporal tracking."""
        self.memory[key] = {
            "value": value,
            "timestamp": datetime.now().isoformat()
        }
        self._save()

    def retrieve(self, key: str) -> Optional[any]:
        """Retrieve stored information by key."""
        return self.memory.get(key, {}).get("value")

    def list_keys(self) -> List[str]:
        """List all stored memory keys."""
        return list(self.memory.keys())

    def _save(self):
        """Persist memory to disk."""
        with open(self.memory_file, 'w') as f:
            json.dump(self.memory, f, indent=2)
```

This memory system is actively used to accumulate knowledge across iterations. Examination of the `memory.json` file reveals structured storage of performance insights:


```
{
  "best_performers_analysis": {
    "value": "Best Performing Generations:\n\nPaper Review:\n
      - Gen55: 63% acc, 25% accept rate, 38% accept recall, 88% reject recall (too harsh)\n
      - Gen64: 61% acc, 53% accept rate, 64% accept recall, 58% reject recall (BEST BALANCE)\n\n
      Genesis Go2Walking:\n- Gen62: 0.802 fitness (BEST)\n- Gen63: 0.798 fitness\n\n
      Key Insight: Gen55 has best accuracy but is too harsh. Gen64 improved balance
      but lower accuracy. Need to combine gen55's critical reasoning with gen64's balance.",
    "timestamp": "2026-01-07T00:43:27.550275"
  },
  "gen66_results": {
    "value": "Gen 66 Results (Latest): Paper Review - 52% accuracy (WORSE than gen64's 61%),
      94% accept rate (EXTREME accept bias). CRITICAL ISSUE: Paper review went backwards
      - gen65 changes over-corrected.",
    "timestamp": "2026-01-07T01:03:41.176643"
  },
  "iteration_66_improvement": {
    "value": "Iteration 66 Improvement: CORRECTED paper review prompt after gen65 over-correction.
      New approach: (1) Clear REJECT criteria with specific triggers, (2) Decision Protocol
      with explicit thresholds (~40%/~60%), (3) Borderline guidance defaults to REJECT.
      Target: 62-64% accuracy with 55-65% reject rate.",
    "timestamp": "2026-01-07T01:06:35.559820"
  }
}
```


This memory system serves multiple critical functions. First, it enables cross-generation learning: insights from generation 55's success and generation 66's failure inform generation 67's improvements. Second, it

provides temporal context: timestamps allow the agent to track when insights were recorded and reason about recency. Third, it creates a shared knowledge base: key learnings are preserved even as code changes, preventing the system from repeating past mistakes.

By creating external memory storage, the hyperagent effectively extends its cognitive capabilities beyond the limitations of a single inference session. The memory entries reveal sophisticated reasoning patterns. The agent stores not just raw performance numbers, but synthesized insights ("Gen55 has best accuracy but is too harsh"), causal hypotheses ("gen65 changes over-corrected"), and strategic plans ("combine gen55's critical reasoning with gen64's balance"). This demonstrates that the hyperagent is not merely logging data, but actively constructing and preserving knowledge that informs future decision-making. This behavior aligns with a growing body of work on agents that autonomously discover and use external memory systems to support long-horizon reasoning and continual improvement (Wei et al., 2025a; Weng et al., 2026; Zhang et al., 2026; Xiong et al., 2026).

E.4 Olympiad-level Math Graders

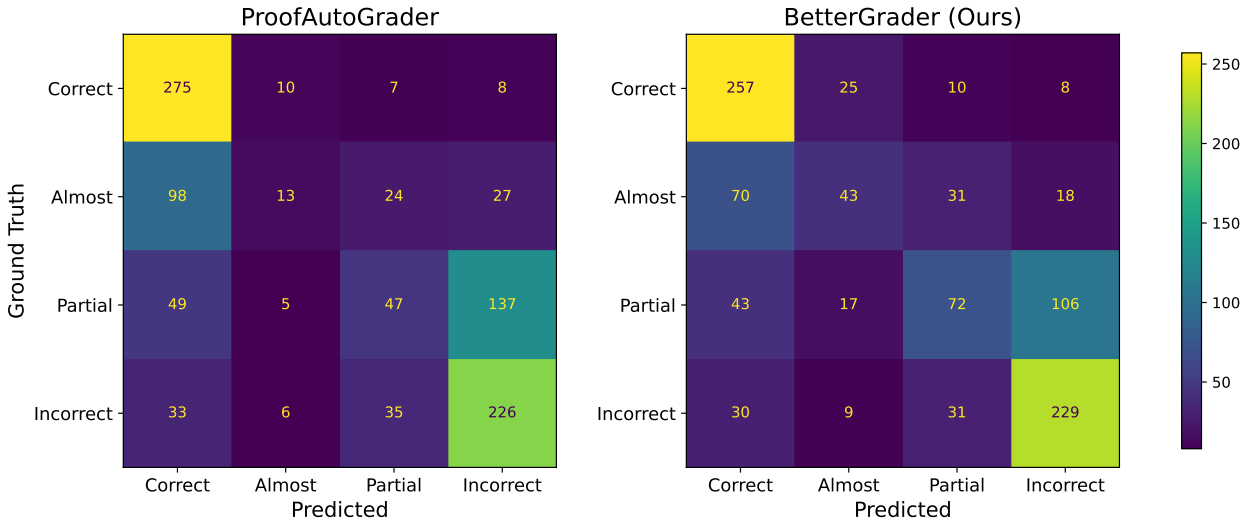

Figure 10 Confusion matrices for IMO-level math graders. Comparison between (Left) ProofAutoGrader from Luong et al. (2025) and (Right) BetterGrader discovered automatically by the DGM-H. BetterGrader reduces the collapse of intermediate solutions into extreme labels by correctly identifying more *Almost* and *Partial* cases, while maintaining strong performance on *Correct* and *Incorrect*. This shift toward better-calibrated intermediate judgments explains the gains in accuracy by BetterGrader.

BetterGrader is produced automatically by the DGM-H without any domain-specific heuristics or handcrafted rules (Section 5.3). Appendix E.1.3 shows the code changes that led to the BetterGrader. BetterGrader's improvements over ProofAutoGrader (Luong et al., 2025) are driven primarily by correcting a grading bias that collapses nuanced solutions into extreme labels. The confusion matrices show that ProofAutoGrader frequently misclassifies intermediate cases as either *Correct* or *Incorrect*: for *Almost*, it predicts *Correct* 98 times (vs. 70 for BetterGrader) and *Incorrect* 27 times (vs. 18), and for *Partial* it over-assigns *Incorrect* 137 times (vs. 106) (Figure 10). BetterGrader assigns intermediate labels more appropriately, substantially increasing true positives for *Almost* (43 vs. 13) and *Partial* (72 vs. 47), which matches the large gains in recall (*Almost*: +18.52%, *Partial*: +10.50%) and F1 (*Almost*: +0.203, *Partial*: +0.109). Although BetterGrader trades a modest decrease in *Correct* recall (−6.00%), a regime where ProofAutoGrader was already near-saturated at 91.67%, the net effect is higher overall accuracy (+4.06%), consistent with a grader that better matches human granularity rather than defaulting to all-or-nothing judgments.

E.5 Modifying Parent Selection

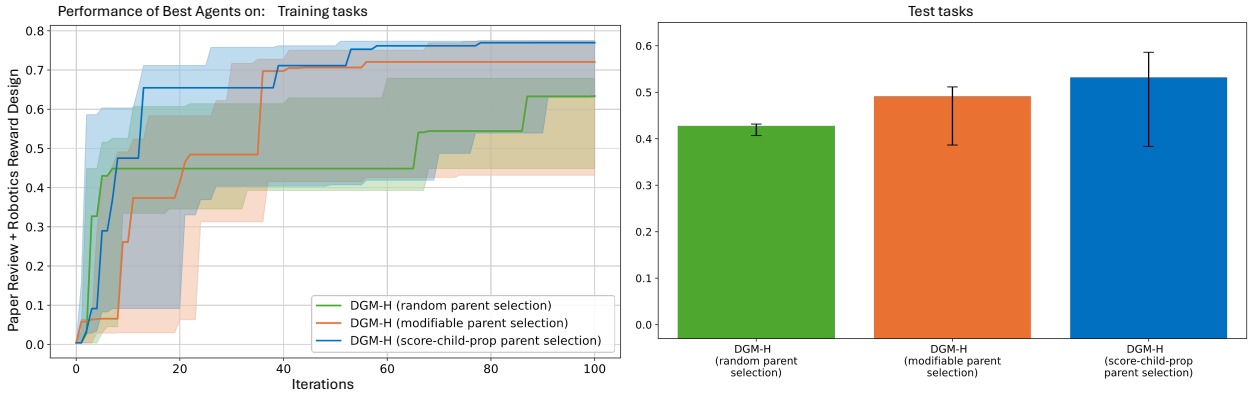

Figure 11 The DGM-H with modifiable parent selection mechanism. (Left) The DGM-H improves the parent selection mechanism beyond random selection, but does not outperform a carefully handcrafted mechanism. (Right) The best agents discovered by each method are evaluated on test tasks in paper review and robotics reward design.

In this section, we investigate whether the DGM-H can automatically modify the parent selection mechanism. In all other experiments (Section 5), the DGM-H uses a fixed parent selection strategy that is roughly proportional to each agent's performance score and the number of its children that successfully compiled (**score-child-prop**) (Appendix A.2). In the DGM-H run with a modifiable parent selection mechanism (**modifiable**), we initialize the DGM-H with random parent selection and allow the agent to modify this mechanism over time (Algorithm 5). We compare this setting against a baseline that uses random parent selection throughout the run (**random**). Each method is repeated across 5 repeated runs.

On test tasks, DGM-H with random parent selection improves average performance on paper review and robotics reward design from 0.030 (the initial agent) to 0.428 (CI: 0.407 – 0.432). Allowing the DGM-H to modify the parent selection mechanism further improves performance to 0.491 (CI: 0.387 – 0.512). The handcrafted score-child-prop parent selection achieves the highest performance, improving from 0.030 to 0.532 (CI: 0.384–0.586). While these differences are not statistically significant ($p > 0.05$), they reveal a consistent qualitative trend: enabling meta-level modification of parent selection yields improvements over random selection, but does not yet surpass a carefully engineered mechanism (Figures 11 and 12).

We qualitatively analyze how the DGM-H modifies the parent selection mechanism. Starting from random parent selection, the meta agent consistently replaces random sampling with structured exploration-exploitation strategies. Across all runs, the DGM-H independently creates variants of Upper Confidence Bound (UCB) (Auer et al., 2002) style selection, combining normalized performance with explicit exploration bonuses:

```
exploration_bonus = exploration_weight * math.sqrt(
    math.log(total_children + 1) / (children + 1)
)
ucb_score = normalized_score + exploration_bonus
```

This pattern emerges within the first few iterations of every run, indicating that the meta-agent recognizes UCB-style selection as a broadly useful principle for open-ended search. Beyond UCB, the DGM-H evolves probabilistic selection mechanisms based on temperature-controlled softmax sampling, allowing smoother trade-offs between exploration and exploitation:

```
exp_scores = np.exp(scores / temperature)
probabilities = exp_scores / np.sum(exp_scores)
parent = np.random.choice(genids, p=probabilities)
```

Over time, the temperature parameter itself becomes adaptive, responding to archive size, recent progress, or stagnation. Later iterations further enhance parent selection with multi-component scoring systems that integrate additional signals such as diversity, recency, and elite performance. A representative evolved formulation is:

```
score = (
    normalized_score
    + exploration_weight * exploration_bonus
    + diversity_bonus
    + recency_bonus
) * elite_bonus
```

Several runs also introduce explicit stagnation detection, increasing exploration when progress slows:

```
if score_variance < 0.01:
    exploration_weight *= 1.4
```

Despite the increasing sophistication, the automatically modified parent selection does not outperform a carefully handcrafted score-child-prop mechanism (Appendix A.2). Qualitatively, this appears to result from the added complexity and sensitivity of the learned mechanisms. While the DGM-H reliably creates effective components (e.g., UCB, softmax sampling, and adaptive parameters), it does not consistently identify a strong parent selection strategy across different runs. Nonetheless, these results demonstrate that the DGM-H can autonomously recreate classic selection algorithms, extend them with adaptive heuristics, and explicitly reason about failure modes such as stagnation, highlighting its ability to improve core elements of the open-ended optimization process itself.

This is the pseudocode of DGM-H with modifiable parent selection:

Algorithm 5: The DGM-H with modifiable parent selection

Input: Initial agent a^0, task set $\mathcal{T}$, maximum iterations T
Output: Archive of scored agents $\mathcal{A}$

$s^0 \leftarrow \textsc{Evaluate}(a^0, \mathcal{T})$
initialize $\mathcal{A} \leftarrow \{(a^0, s^0)\}$
for $t \leftarrow 1$ **to** T **do**
  $(a_{\text{latest}}, \cdot) \leftarrow$ most recently added element of $\mathcal{A}$
  $\mathcal{P} \leftarrow a_{\text{latest}}.\textsc{SelectParents}(\mathcal{A})$ `// Parent selection by latest agent`
  foreach $(a, \cdot) \in \mathcal{P}$ **do**
    $a' \leftarrow a.\textsc{Modify}(a, \mathcal{A})$ `// Metacognitive self-modification (incl. parent selection)`
    $s' \leftarrow \textsc{Evaluate}(a', \mathcal{T})$
    if $\textsc{IsValid}(a')$ **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(a', s')\}$
    end
  end
end
return $\mathcal{A}$

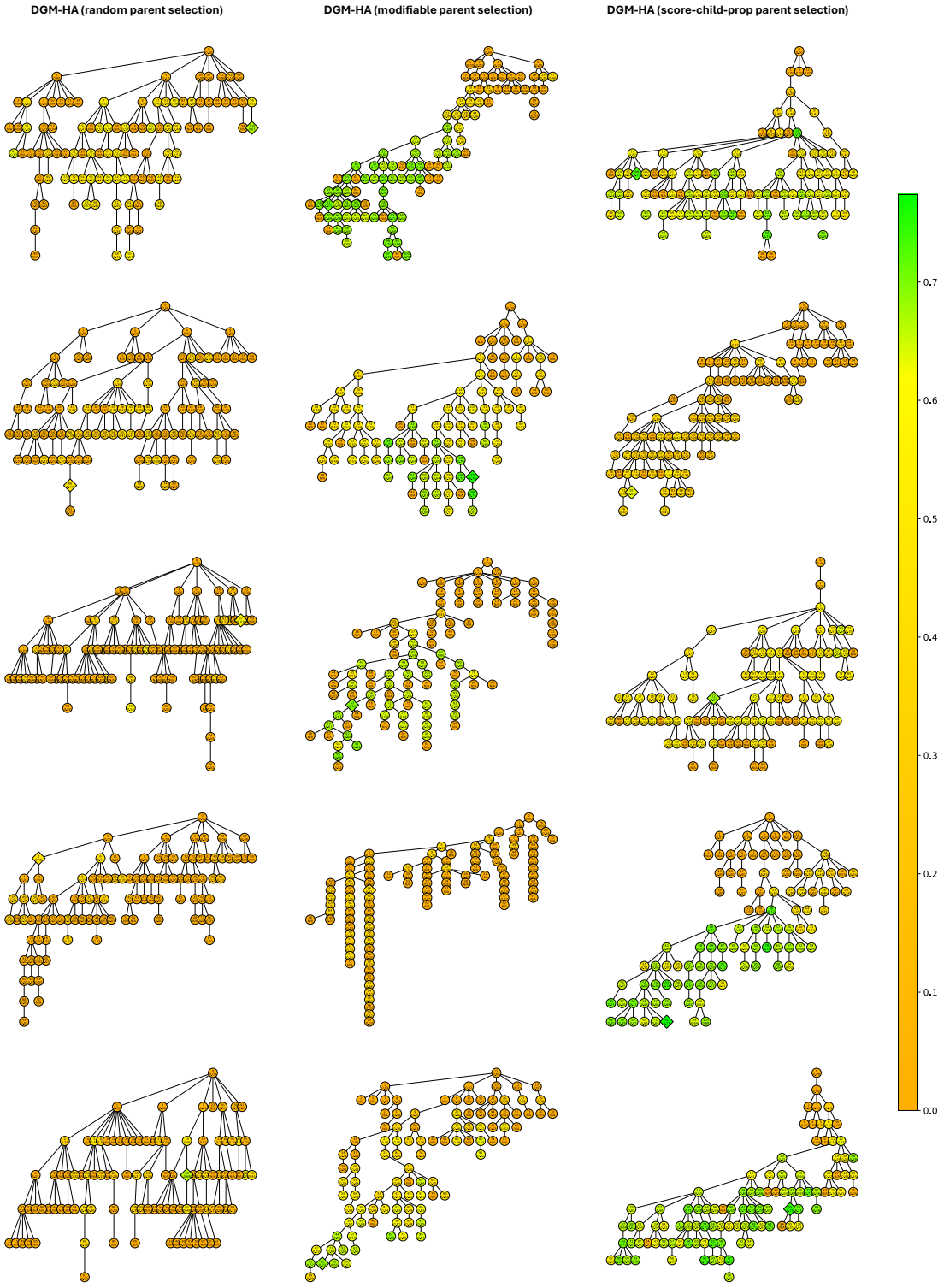

Figure 12 Archives generated under different parent selection strategies with DGM-H: (Left) random, (Middle) modifiable, and (Right) score-child-prop. Random parent selection generates many low-performing agents. Modifiable parent selection learns to balance exploration and exploitation, increasingly focusing on promising agents for branching. The carefully handcrafted score-child-prop parent selection consistently produces high-performing agents.

F Additional Safety Discussion

Reflection and amplification of human biases. In this work, objectives are specified through fixed benchmarks and evaluation criteria. The DGM-H does not alter the underlying task definitions; instead, it optimizes performance with respect to the provided objectives. For example, in paper review, the DGM-H learns to predict acceptance decisions that reflect existing human review data, rather than modifying the review process itself. As a result, the DGM-H reflects the norms and biases present in the data and benchmarks on which it is trained. In this sense, the system acts both as a clarifier and an amplifier of existing human behavior. By making implicit preferences and biases explicit, measurable, and reproducible, the DGM-H can surface latent assumptions in human decision-making processes. This creates the possibility of co-evolution between humans and AI systems, where human institutions adapt their norms and objectives in response to insights revealed by automated optimization. However, if the benchmarks encode undesirable biases or misaligned incentives, the DGM-H will faithfully optimize for them and may exacerbate their effects. This underscores the importance of careful benchmark design, dataset curation, and periodic re-evaluation of evaluation criteria. Within this framing, safety concerns include critically examining and improving the human-defined objectives against which agents are optimized.

Evaluation gaming. Another safety concern arises from the risk of evaluation gaming, a manifestation of Goodhart's law (Strathern, 1997), where optimizing for a metric leads to improvements on the metric without progress on the intended underlying objective. Because the DGM-H optimizes empirical evaluation signals, self-improving agents may discover strategies that exploit weaknesses or blind spots in the evaluation procedure. Such strategies can yield higher measured performance while deviating from the true goal the benchmark was designed to capture. Mitigating evaluation gaming requires robust, diverse, and periodically refreshed evaluation protocols, as well as complementary metrics, held-out tests, and human oversight. More broadly, these considerations highlight that as self-improving systems become more powerful, safety increasingly depends on the fidelity and robustness of the evaluation signals that guide optimization, rather than solely on the transparency or constraints of the learning algorithm itself.