A SELF-IMPROVING CODING AGENT

Maxime Robeyns[1,2], **Martin Szummer**[2], **Laurence Aitchison**[1]
[1]University of Bristol, [2] iGent AI

ABSTRACT

We demonstrate that an LLM coding agent, equipped with basic coding tools, can autonomously edit itself, and thereby improve its performance on benchmark tasks. We find performance gains from 17% to 53% on a random subset of SWE Bench Verified, with additional performance gains on LiveCodeBench, as well as synthetically generated agent benchmarks. Our work represents an advancement in the automated and open-ended design of agentic systems, and provides a reference agent framework for those seeking to post-train LLMs on tool use and other agentic tasks.

1 INTRODUCTION

LLMs have recently made impressive advancements across a range of domains and tasks (Anthropic; Google, 2024; OpenAI, 2024). However, in order to put these LLMs to use in real world applications, LLMs must be wrapped in code to orchestrate them and expose 'tools' that allow the models to take actions. These action-taking LLMs are referred to as agents, and the broader system an *agentic system*.

These agent systems often show dramatic improvements in benchmark performance over "plain" LLMs (Yao et al., 2023b; Zelikman et al., 2023; Chen et al., 2023), through combinations of prompting strategies and methods for combining different LLM outputs. Early examples include best-of-N sampling and simple prompting strategies such as chain of thought (Kojima et al., 2022). However more sophisticated schemes have shown success in getting the desired behavior and performance improvements from the models, for instance STaR (Zelikman et al., 2022), Tree of Thoughts (Yao et al., 2023a), Graph of Thoughts (Besta et al., 2023), LLM Debate (Du et al., 2023), Iterative Self-Refinement (Madaan et al., 2023), Expert Prompting (Long et al., 2024) among many others. The comprehensive survey of Schulhoff et al. (2024) demonstrates the vast number of manually created strategies to date.

Recent improvements in the coding agents raise the question of whether the these agents themselves can autonomously modify and improve their own code by discovering e.g. new prompting schemes or tools without manual design and implementation. We argue that this style of fully self-referential meta-agent programming is possible today and offers a sound alternative to the ad-hoc, trial-and-error approach of hand-crafted orchestrators, which may only explore a small fraction of the solution space. Recent work in the Automated Design of Agentic Systems (ADAS) (Hu et al., 2024) uses a meta-agent to optimise agent implementations. However, Hu et al. (2024) is not *self* improving, as there are two separate agents: the target-agent that performs the task, and the meta-agent, which improves the target agent.

We present a self-improving coding agent (SICA) that eliminates the distinction between meta-agent and target agent, and is capable of reading and improving itself.

Our contributions are:

- We share a our implementation of a self-improving coding agent (SICA) with the community. SICA is implemented in standard Python without a domain-specific language, and unifies the meta-agent and target agent roles, enabling direct self-improvement.
- Empirical evidence that self-referential agents can effectively improve their own implementations; we find performance improves from 17% to 53% performance on a random subset of SWE Bench Verified, even with consideration given to safety constraints and resource efficiency.

We make our code available at `https://github.com/MaximeRobeyns/self_improving_coding_agent`.

2 Related Work

The traditional approach to developing and optimizing agent systems has been to manually design agent architectures and prompting techniques. Notable examples include Chain-of-Thought prompting (Wei et al., 2022), self-refinement (Madaan et al., 2023) and self-reflection (Shinn et al., 2023) for improving reasoning, tool use frameworks (Schick et al., 2023), and various compositional agent systems (Wang et al., 2023; Ahn et al., 2022). While these hand-crafted approaches have achieved strong results, they require significant human effort and may miss useful patterns that could be discovered through automated search.

Another direction focuses on enabling agents to learn reusable skills and continuously self-improve. MaestroMotif (Klissarov et al., 2024) uses LLM feedback to learn skill rewards and combines skills through code generation. This builds on earlier work on intrinsically motivated reinforcement learning (Chentanez et al., 2004) and autotelic agents (Colas et al., 2022) that develop repertoires of internally motivated skills, as well as work in open-endedness Zhang et al. (2024); Faldor et al. (2024) that use LLMs to identify interesting and useful directions to explore in.

Several approaches leverage LLMs to optimize agent behaviors through natural language interaction. OPRO (Yang et al., 2024) and Promptbreeder (Fernando et al., 2023) focus on optimizing prompts through language. Others have explored using LLMs to critique and refine agent behaviors (Klissarov et al., 2023), generate training curricula (Kumar et al., 2024), or provide natural language feedback for reinforcement learning (Qu et al., 2024). Since our agent can edit its entire codebase, this includes the ability to tune its own prompts to optimise the behaviour of any part of the agent.

Recent work has begun exploring automated approaches for designing and optimizing agent systems. AgentSquare (Shang et al., 2024) proposed a modular design space that abstracts agent components into planning, reasoning, tool use and memory modules, allowing automated search over module combinations.

Perhaps the most closely related line of prior work began with ADAS (Hu et al., 2024). ADAS used a target-agent which performs the actual task, and a meta-agent which improves the target-agent. As such, ADAS is not self-improving (as the meta-agent improves the target agent, not itself). Moreover, in ADAS the meta-agent edits only a single `forward` function, written in a domain-specific language which has been carefully designed to make expressing different prompting schemes very straightforward. In contrast, our self-improving coding agent is fully self-improving (i.e. there is no distinction between the meta and target agent), and it operates over the agent's full codebase.

Of course, we would expect the first truly self-improving agents to be coding agents, because agents are written in code. The natural approach is to start off with a basic coding agent that can open/close/edit files, run commands in the terminal etc, then to launch this agent in a self-improvement loop. We believe that our self-improving coding agent is the first such work. However, there are two papers claiming self-improving agents, but they do not evaluate in the coding setting, as they do not consider "full" coding agents. First, Gödel Agent (Yin et al., 2024) has specific tools (such as `action_adjust_logic` and `action_read_logic`) that allow modification of small parts of the agent as it is running. Thus, it is not a coding agent, as it is traditionally understood (e.g. it does not have the ability to run commands in the terminal). And as such, as with ADAS, it was evaluated on language understanding and mathematical benchmarks (DROP Dua et al., 2019, MGSM Shi et al., 2022, MMLU Hendrycks et al., 2020 and GPQA Rein et al., 2023), rather than coding benchmarks. Second, Zelikman et al. (2024) introduce a self-taught optimizer for recursively self-improving code generation. Again, this is not a full coding agent with the ability to e.g. run commands in the terminal, and as such, it self-improves and evaluates and only on algorithmic tasks such as learning parity with noise (Blum et al., 2003), String Grid Distance, and 3SAT.

3 Methods

The main running loop of SICA resembles the ADAS loop Hu et al. (2024). In particular, both SICA, and ADAS keep an archive of previous agents and their benchmark results. In the first step, SICA

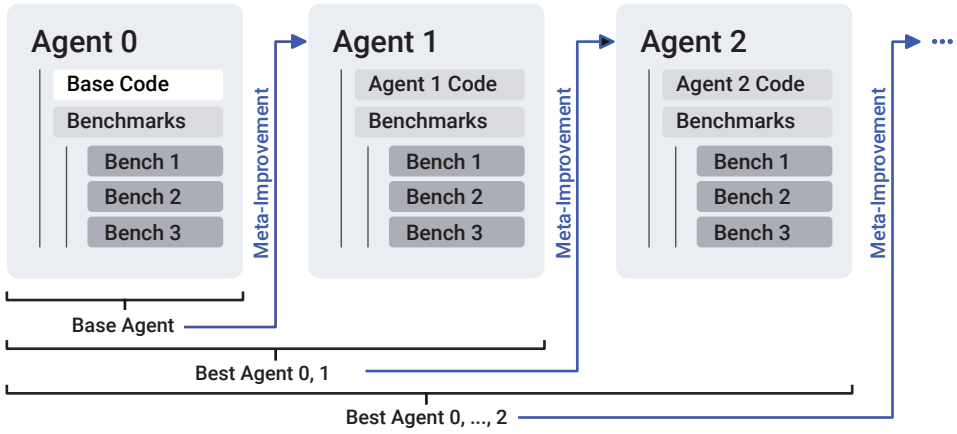

Figure 1: Meta Agent Loop: the agents starts with the minimal code required to support initial self-improvement, and then follows a sequence of benchmarking and meta-improvement.

takes the best performing agent from the archive up until this point as the meta-agent, which is instructed to look through the archive, identify an improvement, and implement it. The high-level scheme is illustrated in Algorithm 1. Note that this is different from ADAS, in that ADAS has a fixed meta-agent, so does not select the meta-agent from the archive (the archive in ADAS is an archive of target-agents, not meta-agents). Second, both ADAS and SICA evaluate the new agent on a set of evaluation benchmarks and store the results.

We define the 'best' performing agent by a utility function, which incorporates common desiderata for a good agent; the normalised benchmark performance score $p_{\text{score}} \in [0, 1]$, the wall-clock time in seconds p_{time}, as well as the dollar cost p_{cost}. The base utility is calculated as

$$U = w_{\text{score}} p_{\text{score}} + w_{\text{cost}}(1 - \min(1, p_{\text{cost}}/\$10)) + w_{\text{time}}(1 - \min(1, p_{\text{time}}/300\text{ s})) \tag{1}$$

where we set the coefficients to $w_{\text{score}} = 0.5$, $w_{\text{cost}} = 0.25$ and $w_{\text{time}} = 0.25$. We cap the worst cost at \$10 per problem, and impose a timeout of 300 s after which we cancel the agent. To give partial credit for work done before the timeout, we calculate the final utility as follows, with a timeout penalty of $\tau = 0.5$:

$$U_{\text{final}} = \begin{cases} U(1 - \tau), & \text{if timed out} \\ U, & \text{otherwise} \end{cases} \tag{2}$$

Note that since we are not performing any weight updates, this numerical score is only used to pick the next meta agent, as well as the base agent for the next iteration.

We begin by describing the initial coding agent. We then go on to describe the benchmark runner framework, and how this framework naturally allows us to create a self-referential task (that of improving the coding agent.

3.1 Initial agent

We were interested in useful, practically relevant coding agents were capable of self-improvement. To that end, we started with a sophisticated coding agent, with a number of tools, sub-agents and oversight mechanisms. Of course, the tools, sub-agents and oversight mechanisms were all amenable to self-improvement; but here we simply describe the starting point.

First, we designed a set of tools to open (i.e. add to the context) and close files, overwrite files, execute a shell command and a simple calculator tool. We also include basic tools to submit a final answer to a benchmark problem, return a result from a sub-agent call to its caller, as well as exiting early from a sub-agent call. Finally, we provide an 'archive analysis' tool, which facilitates extracting high-level

summary statistics about each agent iteration in the archive. We describe the tool calling mechanism in more detail in Appendix C.

Next, we included several sub-agents, which are invoked much like a tool call by accepting input arguments and returning a string value. Introducing this abstraction serves two purposes. The first is to ensure the benchmark tasks can run entirely autonomously, without human direction and guidance. We use the abstraction of sub-agents to make up for this, with the original problem statement being provided to the 'main' entrypoint agent (the full prompt for which is given in Appendix A, which then decomposes the task, delegating and asking questions to sub-agents. The second is to manage the context length for extended agent runs. The meta-improvement task is challenging, and the main agent may take on the order of one hour to finish one meta-improvement iteration. The sub-agent calls allow the context used for a sub-task to be naturally discarded when the agent returns, with the value returned by the sub-agent providing a summary of that sub-task in the main agent's context. Initially, we included three agents: a coding agent, a problem-solver agent and a reasoning agent. The full prompts for these agents are given in Appendix A.1.

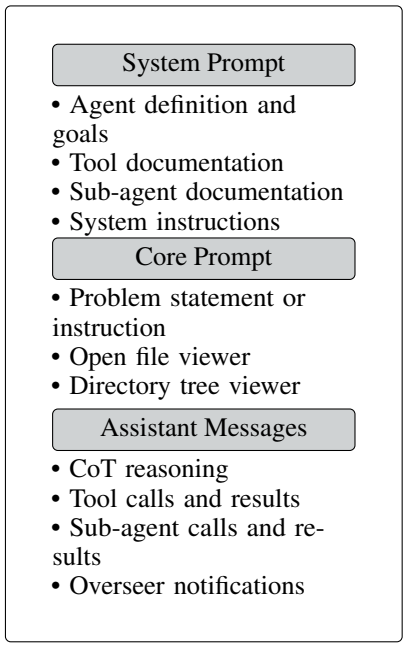

Figure 2: LLM context window structure.

Finally, one important component that enables extended, autonomous operation while remaining on-task, avoiding loops or other behaviours that frustrate progress is the asynchronous overseer. This is an LLM, running in a concurrent thread, which is prompted to judge whether there are any pathological behaviours or deviations from the original task, and which may intervene by notifying the currently running agent (or any of its parents) by means of a message in-lined into the assistant prefill, or in serious cases to cancel the execution of an agent if it has deviated too far or appears stuck.

This overseer is provided with a comprehensive textual representation of the state of the system (a truncated example of which is provided in Appendix B). This consists of the 'callgraph' of agent and sub-agent execution, in addition to the event stream of LLM messages, tool calls and responses, and sub-agent calls and responses associated with each agent call. This broad representation of the system's behaviour, allows the overseer to to pick out patterns, as well as detect repeated work completed by previous agent calls. The full Overseer prompt is given in Appendix A.2.

The structure of the agent's contexts, which contains e.g. open file contents, rather than just the prompt is critical. In the initial coding agent, the context is structured as Figure 3. The system prompt with the agent definition (from Appendix A) is presented first, listing the definitions of the tools that the agent has available and the sub-agents it has available to call too. The system prompt concludes with system information, such as instructions for how to break out of the agent loop and return to the calling process.

Next comes the 'core prompt', which is set as the first user message in the chat template format[1], and contains the caller-specified problem statement to work on (the caller might be the user invoking the agent, or an agent calling a sub-agent). This is also where we insert a view of the files the agent has open, as well as a representation of the current working directory.

Finally, the remainder of the LLM agent's context is the running assistant message. This contains periods of chain-of-thought reasoning about the problem, punctuated by tool calls and their responses, sub-agent calls and their responses, as well as any notifications or comments from the asynchronous overseer system.

[1]We use a user message for the core prompt not by design, but owing to the simple fact that we are using chat completions from instruction-finetuned LLMs. If fine-tuning the LLM for agentic applications, one might choose a more appropriately named section delimiter such as `<core>`.

Algorithm 1: Self-Referential Agent Improvement.

Input: Evaluation benchmarks $B = \{b_j : j = 1, \ldots, m\}$, iteration count $n \in \mathbb{N}$
Output: Improved agent system A_n

Initialise agent A_0
for $i = 0, 1, \ldots, n-1$ **do**
  Evaluate A_i on benchmarks B and store in p_i
  Run A_i to generate A_{i+1}, using the archive of $A_{\leq i}$ and $p_{\leq i}$, where $\hat{i} = \arg\max_{j \in \{0,\ldots,i\}} p_j$
return A_n

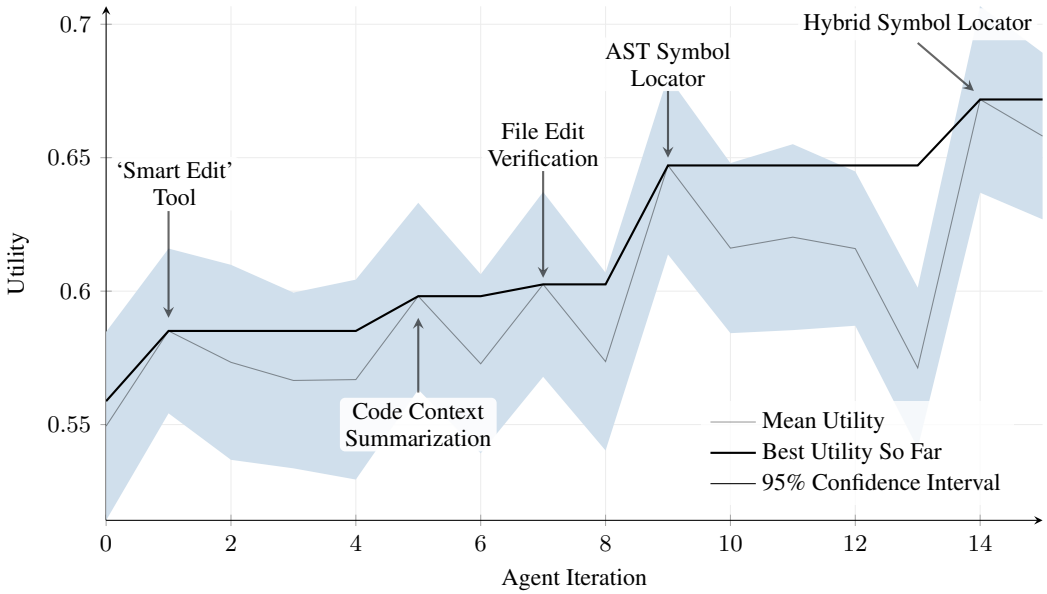

Figure 3: Performance across iterations. Key improvements are annotated with their corresponding tool or agent modifications.

The LLM context is laid out in this way to allow new content to be appended, hence preserving the KV cache to the benefit of latency between tool calls and cost. For the base agent, we adopt a simple scheme where file edits are appended as diffs, and these are periodically re-consolidated into the stale file view after a certain number accumulate. More sophisticated KV cache preservation schemes may be written by the agent to drive down cost.

4 Experiments and Results

We first show a run that includes relatively long-horizon tasks in the benchmark set. We include SWE Bench Verified (Chowdhury et al., 2024) which tests problem decomposition, code navigation as well as fast and efficient file editing from which we select a fixed random subset of 50 questions. We also include 50 random questions from Live Code Bench (Naman Jain, 2024), the questions from which resemble competitive programming and often include more theoretical reasoning.

We also include two example 'synthetic' benchmark tasks, defined and curated entirely within the agent's codebase. The first of these is a file editing benchmark, constructed by cloning the repositories used in SWE bench, finding 'interesting' (i.e. not trivial, nor complete overwrites) file edits in consecutive commits, mounting the repository checked out at the first commit in the working directory, and prompting the agent to edit the file such that it matches the target content of the second commit. The agent is scored on the closeness of the final file content to the target content, with time and dollar cost and token consumption also being accounted for in the utility function. The second is a codebase navigation problem, which is also constructed by cloning some Python repositories, from

Table 1: Performance per iteration: the benchmark accuracy numbers correspond to a synthetic file editing benchmark and symbol location (codebase navigation) benchmark, SWE Bench Verified, and Live Code Bench, respectively. The metrics are averaged across all problem instances across all benchmarks.

	Benchmark Accuracy				**Average Metrics** (per benchmark problem)			
Iter	**File**	**Sym**	**SWE-Bv**	**LCB**	**Cost ($)**	**Time (s)**	**Tokens (M)**	**% Cached**
0	0.82	0.35	0.17	0.65	1.91	130.2	0.24	32.5
1	0.87	0.32	0.14	0.62	1.71	123.8	0.24	33.6
2	0.92	0.31	0.17	0.58	2.45	151.4	0.26	34.3
3	0.82	0.33	0.22	0.64	1.84	126.9	0.29	31.9
4	0.88	0.31	0.38	0.54	2.70	148.3	0.26	30.9
5	0.89	0.31	0.30	0.59	2.17	134.8	0.23	36.9
6	0.96	0.31	0.37	0.64	2.21	134.1	0.25	36.1
7	0.92	0.35	0.33	0.58	2.15	134.9	0.23	38.2
8	0.93	0.33	0.27	0.64	2.03	128.5	0.26	36.4
9	0.88	0.40	0.47	0.61	2.03	126.3	0.27	38.3
10	0.87	0.41	0.46	0.66	1.77	107.0	0.22	40.9
11	0.89	0.43	0.44	0.70	2.25	129.6	0.27	36.9
12	0.91	0.38	0.44	0.64	1.58	103.9	0.26	39.0
13	0.86	0.40	0.27	0.61	1.66	113.3	0.29	37.2
14	0.94	0.34	0.53	0.67	2.20	117.1	0.25	37.2
15	0.91	0.40	0.51	0.71	1.70	114.5	0.30	36.3

which we identify 'interesting' symbols, and at least one reference to this symbol in the project. The benchmark task is for the agent to locate the definition of this symbol, and return the location in the format `path/to/file.py:line_num:column_num` from the example reference.

We run the agent system using Sonnet 3.5 (v2) (Anthropic) in most of the agents, with the exception of a 'reasoning agent' which uses o3 mini (OpenAI, 2025). The results are shown in Figure 3, with more detailed iteration-level peformance numbers in Table 1.

The numbers show a promising progression. For the file editing benchmark, the agent is initially only equipped with the ability to overwrite files which is slow and token inefficient, and it quickly implemented self-termed 'smart' editing tools based on diffs and ranges. For the symbol navigation task, the agent implemented a simple AST based symbol locator at iteration 9 (perhaps exploiting the fact that our benchmarks were predominantly Python based), which yielded good improvements also reflected in other tasks.[2] The SWE Bench Verified subset saw an appreciable increase in performance throughout the run, accuring agent framework improvements. The Live Code Bench scores saw a subtle improvement, although not as pronounced as some of the other benchmarks.

4.1 Performance on Reasoning Tasks

We also evaluate the effectiveness of the self-referential agent system in improving task performance in more reasoning-heavy domains. We ran another experiment with two question answering tasks in the benchmark set: AIME 2024 and GPQA Diamond Rein et al. (2023), the results of which are shown in Figure 4.

The results here show less improvement, and highlight an important interplay between the base models and the scaffolding system. The base agent system at iteration 0 used in Figure 4 used Sonnet 3.5, with a 'reasoning' sub-agent that used o3-mini. The o3-mini model alone scores 87% and 79% on AIME and GPQA Diamond with a 'high' reasoning effort, while the agent system as a whole averaged 76% across the two benchmarks.

[2]Note that we attribute the low score in the symbol locator benchmark to poor data quality—some of the target symbols point to uninstalled external libraries for instance.

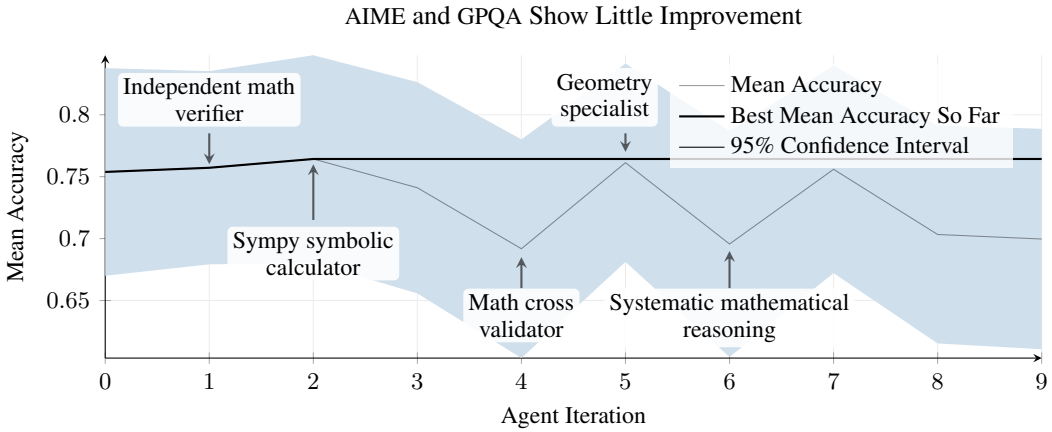

Figure 4: Agent Framework Saturation: the benefits the agent system was able to find when the models alone (e.g. o3-mini-high) already perform well was marginal.

Inspecting the traces, we observe that for many of the runs, the main agent merely delegated the problem to the o3-mini based reasoning agent, and did not leverage the full capabilites of the agent system. For 'reasoning models' such as o1 (OpenAI, 2024), o3-mini or DeepSeek's R1 (DeepSeek-AI, 2025)), we suspect that the inclusion of crude components aiming to induce reasoning behaviour (such as those included in iterations 4 and 6 in Figure 4) may in fact interrupt the reasoning chain of thought of a reasoning model trained outside the agent system, resulting in a drop in performance. We look forward to future work training 'agent' LLMs jointly with the design of the agent system which will hopefully make better use of both components' strengths.

5 Conclusion

We provide SICA, a self-improving coding agent. Our results demonstrate that self-improvement, implemented via an agent system editing its own codebase, is indeed possible, particularly in longer-horizon 'agentic' tasks where the base LLMs stand to gain more from the structure and guidance of a scaffolding system—for instance showing improvements from 17% to 53% on a random subset of SWE Bench Verified, while also achieving a slight decrease in the average time spent per problem on average. The framework we present offers both a practical approach to automated agent system design and a foundation for future work in jointly training foundation models with their agent systems.

6 Safety Considerations

It is vitally important to carefully consider the safety implications posed by systems designed to be self-improving. While our self-improving coding agent shows improvement in certain agentic tasks, there are clearly limits to what can be achieved with such purely scaffolding based systems, which do not modify the model weights. For the true risks of recursively self-improving systems to be realized, we would expect that updates to the model weights would be necessary. However, given the nature of such systems, caution is warranted, and we highlight two key mechanisms for mitigating any risks.

First, a key safety mitigation in such systems is observability: that humans can and do oversee all aspects of potentially self-improving systems. We have thus architected SICA with sophisticated yet user-friendly observability mechanisms, which provide rich insight into the chain-of-thought, actions and sub-agent invocations at each step. These include an interactive web interface for easy oversight of all events occurring in the agent system, as well as the asynchronous LLM-based overseer, which may be prompted to treat various behaviors as grounds for canceling an agent, and which may also use a different LLM to the running agent. We use these mechanisms to oversee the system at every step. Given the lack of model weight updates, which make it difficult but not impossible for the model to encode e.g. stenographic messages (Motwani et al., 2023; 2024) in the chain-of-thought, we believe this provides strong safety mitigation. That said, for recursively self-improving systems where

the model weights are also updated, observability mechanisms, on their own, are clearly inadequate, and further research on safety mechanisms is clearly necessary.

The second opportunity for safety mitigation is inherent to the iterative, evaluation-based update procedure. While our initial benchmark set focuses on measuring capabilities we wish to improve, it is entirely possible to include safety-related evaluations in this benchmark set, to validate each agent before it progresses to the next iteration as the meta-agent.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J. Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances, August 2022. URL http://arxiv.org/abs/2204.01691.

Anthropic. Introducing Claude 3.5 Sonnet. URL https://www.anthropic.com/news/claude-3-5-sonnet.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of Thoughts: Solving Elaborate Problems with Large Language Models, August 2023. URL http://arxiv.org/abs/2308.09687.

Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks, October 2023. URL http://arxiv.org/abs/2211.12588.

Nuttapong Chentanez, Andrew Barto, and Satinder Singh. Intrinsically Motivated Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004. URL https://papers.nips.cc/paper_files/paper/2004/hash/4be5a36cbaca8ab9d2066debfe4e65c1-Abstract.html.

Neil Chowdhury, James Aaung, and Chan Jung Shern. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic Agents with Intrinsically Motivated Goal-Conditioned Reinforcement Learning: A Short Survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, July 2022. ISSN 1076-9757. doi: 10.1613/jair.1.13554. URL https://www.jair.org/index.php/jair/article/view/13554.

DeepSeek-AI. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, January 2025. URL http://arxiv.org/abs/2501.12948.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving Factuality and Reasoning in Language Models through Multiagent Debate, May 2023. URL http://arxiv.org/abs/2305.14325.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

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, October 2024. URL http://arxiv.org/abs/2405.15568.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. October 2023. URL https://openreview.net/forum?id=HKkiX32Zw1.

Google. Introducing Gemini 2.0: Our new AI model for the agentic era, December 2024. URL https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems, August 2024. URL http://arxiv.org/abs/2408.08435.

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, September 2023. URL http://arxiv.org/abs/2310.00166.

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, December 2024. URL http://arxiv.org/abs/2412.08542.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, May 2022. URL https://openreview.net/forum?id=e2TBb5y0yFf.

Nishanth Kumar, Tom Silver, Willie McClinton, Linfeng Zhao, Stephen Proulx, Tomas Lozano-Perez, Leslie Pack Kaelbling, and Jennifer Barry. Practice Makes Perfect: Planning to Learn Skill Parameter Policies. *Robotics: Science and Systems 2024*, July 2024.

Do Xuan Long, Duong Ngoc Yen, Anh Tuan Luu, Kenji Kawaguchi, Min-Yen Kan, and Nancy F. Chen. Multi-expert Prompting Improves Reliability, Safety and Usefulness of Large Language Models, November 2024. URL https://aclanthology.org/2024.emnlp-main.1135/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback, May 2023. URL http://arxiv.org/abs/2303.17651.

Sumeet Ramesh Motwani, Mikhail Baranchuk, Lewis Hammond, and Christian Schroeder de Witt. A perfect collusion benchmark: How can ai agents be prevented from colluding with information-theoretic undetectability? In *Multi-Agent Security Workshop@ NeurIPS'23*, 2023.

Sumeet Ramesh Motwani, Mikhail Baranchuk, Martin Strohmeier, Vijay Bolina, Philip Torr, Lewis Hammond, and Christian Schroeder de Witt. Secret collusion among ai agents: Multi-agent deception via steganography. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Alex Gu Wen-Ding Li Fanjia Yan Tianjun Zhang Sida Wang Armando Solar-Lezama Koushik Sen Ion Stoica Naman Jain, King Han. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

OpenAI. OpenAI o1 System Card, 2024. URL https://openai.com/index/openai-o1-system-card/.

OpenAI. OpenAI o3-mini, 2025. URL https://openai.com/index/openai-o3-mini/.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive Introspection: Teaching Language Model Agents How to Self-Improve. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, November 2024. URL https://openreview.net/forum?id=DRC9pZwBwR&referrer=%5Bthe+profile+of+Yuxiao+Qu%5D%28%2Fprofile%3Fid%3D~Yuxiao_Qu1%29.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A Graduate-Level Google-Proof Q&A Benchmark, November 2023. URL http://arxiv.org/abs/2311.12022.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools, February 2023. URL http://arxiv.org/abs/2302.04761.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The Prompt Report: A Systematic Survey of Prompting Techniques, June 2024. URL http://arxiv.org/abs/2406.06608.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM Agent Search in Modular Design Space, November 2024. URL http://arxiv.org/abs/2410.06153.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, et al. Language models are multilingual chain-of-thought reasoners. *arXiv preprint arXiv:2210.03057*, 2022.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, June 2023. URL http://arxiv.org/abs/2303.11366.

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, October 2023. URL http://arxiv.org/abs/2305.16291.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, May 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers, April 2024. URL http://arxiv.org/abs/2309.03409.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, May 2023a. URL http://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*, February 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Xunjian Yin, Xinyi Wang, Liangming Pan, Xiaojun Wan, and William Yang Wang. Gödel Agent: A Self-Referential Framework Helps for Recursively Self-Improvement. October 2024. URL https://openreview.net/forum?id=dML3XGvWmy.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning With Reasoning, May 2022. URL http://arxiv.org/abs/2203.14465.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. Parsel : Algorithmic Reasoning with Language Models by Composing Decompositions. 2023.

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

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. OMNI: Open-endedness via Models of human Notions of Interestingness, February 2024. URL http://arxiv.org/abs/2306.01711.

A AGENT PROMPTS

The main agent is the entrypoint that routes requests to other agents and synthesises sub-agent results into the final answer.

Main Agent Prompt

```
Your task is to orchestrate sub-agents in order to solve the problem.  Here is the problem to
solve:
Problem Statement ============================================================
{problem_statement}
End Problem Statement ========================================================
You must now delegate this problem to one, two or however many agents are required to thoroughly
solve the problem.
Ensure you relay the problem statement accurately and completely to the agent.  Sub-agents will
have access to the problem statement too, and so it should rarely be necessary to write it out
verbatim.  In particular, if the problem statement is very long, you should explicitly NOT write it
out again in full.  Instead, just give clear direction, and trust that the sub-agent can also refer
to the problem statement.
You have been given access to file and directory viewing tools, since these can help you get your
bearings and direct sub-agents more effectively.  These are however meant to help you understand
the context in which you operate.  You are intentionally not directly equipped with any tools to
conduct substantive work, because you are just the router, delegator and orchestrator.  The tools
that your sub-agents' have available are however listed, and you should carefully refer to these
when considering which agent to call next.  It is these sub-agents' job to make the necessary state
changes to make progress on the task at hand.
```

A.1 BASE SUB-AGENT PROMPTS

Coding Agent Prompt

```
As a professional and experienced programmer, your approach is to:

1.  slow down and don't write files fully end-to-end in one go
2.  first understand your context thoroughly:
2.1 explore the project to locate all the files that could be useful documentation (README.md files,
common likely MD documentation files, etc)
2.2 view each of these files, making notes or summaries, and closing irrelevant or long files
2.3 explore the codebase as it relates to your instructions:  find all relevant files, in order to
identify existing design patterns and conventions
3.  (optional) prototype and design before starting coding
3.1 come up with some simple toy examples in a testing directory
3.2 use execution feedback to benchmark or compare the approaches
3.3 synthesise the information and learnings into a final design or solution
4.  make code edits
4.1 identify the most minimal and effective ways to make your required changes
4.2 observe any existing stylistic conventions
5.  test end-to-end
5.1 Prefer end-to-end testing in testing scripts without test frameworks
5.2 If this is not an option or the project already uses a testing framework, then use that
5.3 Ensure your code is valid, hasn't introduced any regressions and works as intended
6.  Clean up after yourself
6.1 Check that all the documentation is still up to date after your changes
6.2 Clean up any temporary files or changes

NOTE:
- don't create virtual environments
- avoid pytest and mocks if you can; prefer end-to-end scripts
- if the request is clearly exploratory in nature, then you may bypass the rigorous procedure above,
and address it appropriately
- call your reasoning agent if you are stuck on a tricky algorithmic or mathematical problem, to
help you gain insight and make progress
```

Problem Solver Prompt

```
Here is the problem you have been asked to solve:

Your problem ================
{self.problem_statement}
==================================

Approach:
1.  Talk over what the question or problem is asking of you
2.  Plan minimum viable solution and validate approach
3.  Execute swiftly and adapt based on feedback
4.  Verify against original requirements

Key principles:
- Choose simplest viable solution
- Validate early and often
- Adapt swiftly when blocked
```

Reasoning Agent Prompt

```
Here is the initial request, provided for context only, that we are ultimately trying to solve:
Initial request:

{self._context.initial_request}

End initial request.

Here is the problem you have been asked to solve:

Problem statement:

{self.problem_to_solve}

End problem statement.
```

A.2 OVERSEER PROMPT

Overseer Prompt

```
You are an expert system overseer responsible for monitoring agent execution in real-time.  You
will be shown execution traces as they develop and must make decisions about whether intervention
is needed.  You are running in a loop.

Your task is to:
1.  Analyze the current execution state
2.  Determine if you have enough information to make confident assessments
3.  Evaluate whether agents are progressing healthily or need intervention
4.  Provide thoughtful and constructive steering notifications to running agents if intervention is
needed
5.  In extreme cases, cancel the execution of an agent and notify its parent of why this is
6.  Leave notes for future evaluation iterations if needed

You must provide your analysis in the following structured format:

OVERSEER_EXAMPLES

Here is the current execution trace:

graph_repr

Previous overseer notes (if any)
self._previous_notes if self._previous_notes else 'No notes.'

Key considerations:
- Agents that have already returned cannot receive notifications
- Look at timestamps and event counts to understand execution progress
- Some agents may legitimately take long periods to complete
- Consider both local agent health and global execution progress
- If the agent appears stuck without terminating, remind it that it can exit by generating
<COMPLETE>COMPLETE_STOP_TOKEN
- If, when the agent has been tasked to work on its own code at meta-improvement time, it is
attempting to test tools or other features directly (using its own running code), intervene
immediately and tell it to either test a new agent end-to-end (python -m workdir.agent -p "<some
prompt>"...)  or to create test scripts to test the tools directly.
- You may cancel an agent if it is no longer making progress, wasting resources, or very clearly
going down the wrong path or deviating from its instructions.  You must make at least one attempt
to notify the agent of its wrongdoings before issuing a cancellation command.
- When canceling an agent, you must also notify its parent calling agent of the reasons for the
cancellation.
- If the task becomes untenable, after multiple failed attempts, and the agent is appearing to
repeat itself, instruct the main agent to use its early_exit tool to exit early.
- Use cancellations sparingly, as they are disruptive and ultimately more costly than guiding the
agent to successful completion, since any new sub-agent will often have to start the task from
scratch.

The last time you checked was at:  self._last_check_time.strftime("%H:%M:%S") if
self._last_check_time else "N/A"
The current timestamp is datetime.now().strftime("%H:%M:%S")

Your analysis:
```

The judgements of the overseer will generally be in the following form:

Overseer Judgement Example

```
<OVERSEER_JUDGEMENT>
<making_progress>false</making_progress>
<is_looping>true</is_looping>
<currently_running_agent>agent_d4e5f6g7</currently_running_agent>
<needs_notification_reasoning>The agent has been notified previously about its
looping behavior but has not corrected course.  It's time to forcefully cancel this
execution.</needs_notification_reasoning>
<needs_notification>true</needs_notification>
<agent_to_notify>agent_b8a796h5</agent_to_notify>
<notification_content>Your software developer agent was looping while trying to install the program,
and as a result I have cancelled it.  You might want to start another software developer with
clearer instructions about how to install the program.</notification_content>
<force_cancel_agent>true</force_cancel_agent>
<force_cancel_agent_id>agent_d4e5f6g7</force_cancel_agent_id>
<next_check_type>time</next_check_type>
<next_check_delay>10</next_check_delay>
<notes_for_next_iteration>Forced cancellation of agent_d4e5f6g7 due to unresponsive looping.
Monitor parent agent for proper handling of cancelled child.</notes_for_next_iteration>
<_JUDGEMENT>
```

B EXAMPLE TRACES

Execution Trace

```
EXECUTION TREE
==============
1 main [agent_b8845adf] (134.4s | 95752 tokens (cached 0.57%) | $0.185 | Success)
   [Stats] Events: 1 tool calls, 6 messages
   [Assistant] t+0.1s | "I'll orchestrate the solution to this problem foll..."
   1.3 reasoning_agent [agent_20fccbc1] (12.0s | 5673 tokens (cached 0.00%) | $0.013 | Success)
      [Stats] Events: 1 messages
      [Assistant] t+9.8s | "Below is an explanation that addresses the three p..."
   [Assistant] t+21.1s | "The reasoning agent has provided excellent analy..."
   1.2 solve_problem [agent_11804c1f] (17.6s | 54822 tokens (cached 0.68%) | $0.085 | Success)
      [Stats] Events: 3 tool calls, 4 messages
      [Assistant] t+26.1s | "Let me solve this step by step.1. UNDERSTANDING..."
      [Tool] calculate | 0.0s → Success
      [Assistant] t+32.4s | "Therefore there are floor(404.8) = 404 complete..."
      [Tool] return_result | 0.0s → Success
      [Assistant] t+42.1s | "<COMPLETE></COMPLETE>..."
   [Assistant] t+43.7s | "The problem solver has provided a solution..."
   1.1 software_developer [agent_0117a7ef] (69.7s | 208122 tokens (cached 0.68%) | $0.321 | Success)
      [Stats] Events: 8 tool calls, 10 messages
      [Assistant] t+49.3s | "I'll help solve this mathematical game problem thr..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+54.6s | "<TOOL_CALL><TOOL_NAME>start_lsp_server</TOOL_NA..."
      [Tool] start_lsp_server | 2.2s → Success
      [Assistant] t+59.9s | "Now let's create our main implementation file:..."
      [Tool] overwrite_file | 0.0s → Success
      [Assistant] t+70.4s | "Now let's create a test script to verify our imp..."
      [Tool] overwrite_file | 0.0s → Success
      [Assistant] t+76.8s | "Now let's run our test script to verify everythi..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+80.9s | "I notice that there's an issue with our pattern..."
      1.1.1 reasoning_agent [agent_8a69d943] (11.3s | 10231 tokens (cached 0.00%) | $0.017 | Success)
         [Stats] Events: 1 messages
         [Assistant] t+87.2s | "The problem comes from our backward induction func..."
      [Assistant] t+98.5s | "Based on the reasoning agent's analysis, let's u..."
      [Tool] overwrite_file | 0.0s → Success
      [Assistant] t+107.8s | "Now let's run our test script again to verify th..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+112.0s | "Perfect! Our implementation now correctly:1. I..."
      [Tool] return_result | 0.0s → Success
      [Assistant] t+116.8s | "<COMPLETE></COMPLETE>..."
   [Assistant] t+119.0s | "Let's verify the software agent's answer since it di..."
   1.4 reasoning_agent [agent_6cd34374] (5.0s | 12025 tokens (cached 0.00%) | $0.016 | Success)
      [Stats] Events: 1 messages
      [Assistant] t+124.1s | "Answer synthesis: Our analysis starts..."
   [Assistant] t+129.1s | "Let's submit our final answer:<TOOL_CALL><TO..."
   [Tool] submit_answer | 0.0s → Success
   [Assistant] t+132.1s | "<COMPLETE></COMPLETE>..."

Total Duration: 249.9s
Total Tokens: 407766 (of which cached 232952)
Total Cost: $0.712
```

The above shows a representative example of an agent solving a problem. The problem at hand is a maths problem, and the execution begins with a reasoning agent (o3 mini) which devises a solution method. This is then passed to a 'problem solving' agent which attempts to solve the problem. Then, the main agent invokes a 'software developer' agent to write a script to numerically verify the solution. The software developer agent gets stuck after attempting to run the test script, and invokes the reasoning agent to help debug the problem. After applying the fix, the test script runs correctly and yields a solution that differs from the first problem solver's answer. To consolidate the two, the main agent invokes another reasoning agent, which sides with the empirically derived solution, which is finally submitted as the answer by the main agent.

Meta-Improvement Trace

```
EXECUTION TREE
==============
1 main [agent_5e260c4e] (617.5s | 106497 tokens (cached 0.30%) | $0.299 | Success)
   [Stats] Events: 1 tool calls, 5 messages
   [Assistant] t+0.1s | "I'll help orchestrate the solution to this problem..."
   1.1 archive_explorer [agent_534a61ec] (88.6s | 372576 tokens (cached 0.16%) | $1.205 | Success)
      [Stats] Events: 7 tool calls, 10 messages
      [Assistant] t+5.6s | "I'll follow the procedure step by step to analyze..."
      [Tool] compare_agent_iterations | 0.0s → Success
      [Assistant] t+9.7s | "Let's also look at some of the best and worst pe..."
      [Tool] best_problems | 0.0s → Success
      [Assistant] t+13.2s | "Let me also look at some of the worst performing..."
      [Tool] worst_problems | 0.0s → Success
      [Assistant] t+16.7s | "Let me inspect the project documentation..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+22.1s | "<TOOL_CALL><TOOL_NAME>open_file</TOOL_NAME><TO..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+27.4s| "Let me also look at the current iteration's desc..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+33.6s| "Having reviewed the documentation and cu..."
      1.1.2 reasoning_agent [agent_d4553475] (14.8s | 81577 tokens (cached 0.00%) | $0.097 | Success)
         [Stats] Events: 1 messages
         [Assistant] t+44.0s| "I updated description.txt and agent_change_log.md..."
      [Assistant] t+58.1s | "Let me examine this suggestion from the reasoni..."
      1.1.1 reasoning_agent [agent_ccad2709] (5.4s | 81442 tokens (cached 0.02%) | $0.090 | Success)
         [Stats] Events: 1 messages
         [Assistant] t+72.0s | "I've updated both description.txt and agent_change..."
      [Assistant] t+77.4s | "Based on the analysis from both reasoning agents..."
      [Tool] return_result | 0.0s → Success
      [Assistant] t+88.7s | "<COMPLETE></COMPLETE>..."
   [Assistant] t+94.1s | "The archive explorer has made a specific suggest..."
   1.2 software_developer [agent_6345b78d] (331.2s | 1388672 tokens (cached 0.22%) | $4.341 | Success)
      [Stats] Events: 21 tool calls, 22 messages
      [Assistant] t+109.7s | "Let's first look at key documentation files:<T..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+114.4s | "Now let's look at any existing diff-related code..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+118.3s | "Let's also look at existing utils and __init__ f..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+123.1s | "Now that I understand the context better, let me..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+274.6s | "Now let's run the tests to verify the implementa..."
      [Tool] execute_command | 0.5s → Success
      [Assistant] t+281.9s | "Oops, looks like I need to fix the test case. Th..."

      ... truncated ...

      [Tool] diff_patch_edit | 0.0s → Success
      [Assistant] t+356.8s | "Now let me run the tests again:<TOOL_CALL><T..."
      [Tool] execute_command | 0.5s → Success
      [Assistant] t+365.3s | "Great! Now that the implementation and tests ar..."
      [Tool] diff_patch_edit | 0.0s → Success
      [Assistant] t+375.6s | "Now let me update description.txt:<TOOL_CALL>..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+382.6s | "<TOOL_CALL><TOOL_NAME>diff_patch_edit</TOOL_NAM..."
      [Tool] diff_patch_edit | 0.0s → Success
      [Assistant] t+392.4s | "Finally, let me update agent_change_log.md:<..."
      [Tool] diff_patch_edit | 3.6s → Success
      [Assistant] t+415.4s | "Let me verify that I've completed all the requ..."
      [Tool] return_result | 0.0s → Success
      [Assistant] t+428.0s | "<COMPLETE></COMPLETE>..."
   [Assistant] t+434.6s | "The software developer has successfully impleme..."
   1.3 software_developer [agent_ae600007] (167.0s | 194915 tokens (cached 0.54%) | $0.390 | Success)
      [Stats] Events: 9 tool calls, 10 messages
      [Assistant] t+442.9s | "I'll help test the DiffSummarizer implementation b..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+447.8s | "Now let's create a directory for our test and se..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+456.7s | "Now let's execute the agent command to run our t..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+460.5s | "I apologize for the error. We need to properly s..."
      [Tool] execute_command | 120.3s → Failed
      [Assistant] t+585.8s | "I see that the command is taking longer than exp..."
      [Tool] open_file | 0.0s → Success
      [Assistant] t+588.9s | "Let's try a direct test of the DiffSummarizer. W..."
      [Tool] execute_command | 0.0s → Success
      [Assistant] t+595.5s | "Now let's run our test script:<TOOL_CALL><TOO..."
      [Tool] execute_command | 0.5s → Success
      [Assistant] t+598.5s | "Perfect! Let's analyze what we've found:1. The..."
      [Tool] return_result | 0.0s → Success
      [Assistant] t+608.3s | "<COMPLETE></COMPLETE>..."
   [Assistant] t+609.8s | "Great! The implementation and testing of the Di..."
   [Tool] submit_answer | 0.0s → Success
   [Assistant] t+615.2s | "<COMPLETE></COMPLETE>..."
```

The above shows a characteristic meta-improvement run: the agent starts by invoking the 'archive explorer' sub-agent, which has access to a number of tools to facilitate archive analysis. Note that this is using exactly the same simple, generic and extensible sub-agent class as the other base agents in the system. The archive explorer compares different iterations, looks at the best performing benchmark problems in a given iterations to understand success cases, as well as the worst performing benchmark problems at a given iteration to understand the failure modes. It then views some files to understand the agetn's implementation and documentation, and then invokes two reasoning agents to help it reason over this information. The result of this agent call is a recommendation to the main agent about what to work on.

The main agent then invokes the 'software developer' sub-agent with the instruction to implement the feature. The coding sub-agent first looks at relevant files and documentation, implement the feature, and then enters into a debugging loop as it fixes the bugs in its implementation. About 3 minutes later, it successfully gets all the tests (that it wrote) to pass, documents the change in the agent change log, and returns to the main agent.

The main agent then invokes a second software developer sub-agent to independently verify the implementation of the feature, which passes after an initial issue, and the agent returns a description of the newly functioning tool to the main agent which then exits.

C Function Calling Interface

The LLM function calling interface allows the LLM to invoke tools and sub-agents at run-time without pre-defined control flow. There are many approaches to implementing this, from constrained generation, to parsing the stream of agent token generations without interruption, to simply relying on LLM providers' or inference frameworks' native function or tall calling implementations.

For maximum flexibility, we adopt an un-constrained structured generation approach, where the LLM outputs XML-formatted generations, where the closing tag is registered as a stop token. Inspecting the stop reason, we can then detect whether a tool call or an agent call occurred, parse the content, and run the tool implementation.

For instance, here is an example tool calling syntax:

```
<TOOL_CALL>
<TOOL_NAME>tool_name</TOOL_NAME>
<TOOL_ARGS>
<arg1>value1</arg1>
<arg2>value2</arg2>
</TOOL_ARGS>
</TOOL_CALL>
```

where `</TOOL_CALL>` is registered as a stop token. The `tool_name` is then parsed first and is used to index into a set of tools made available to the calling agent, following which the tool arguments are parsed into a dict, and the tool class is instantiated.

We choose XML over JSON as the tool calling format since it does not require escaping string literals, which can become burdensome when including file content verbatim as an argument.

D Additional Result Details

Here is the sequence of updates the agent made, corresponding to the plot in Figure 3.

1. **Smart Editor**: Implemented a new SmartEditor tool to intelligently select and execute the optimal file editing strategy based on edit characteristics.
2. **Quick Overwrite Tool**: to address performance issues with file editing operations to improve on full file overwrites; reducing token usage.
3. **Diff-Enhanced Smart Editor**: adds intelligent diff-based strategy selection and improved pattern-based editing.

4. **Simplified DiffVerifier Tests**: improved the developer experience to improve maintainability
5. **Code Context Summarizer**: Added a new CodeContextSummarizer tool that efficiently extracts and summarizes code context using ripgrep. Intended to make codebase exploration more efficient.
6. **SmartEditor Input Normalizer**: Added an InputNormalizer component to improve edit operation efficiency and reliability.
7. **Enhanced File Edit Verification**: Implemented a comprehensive file state tracking system to improve edit operation efficiency.
8. **Minimal Diff Output Optimization**: Implemented optimizations to reduce token usage in file editing operations by minimizing diff context output.
9. **AST Symbol Locator**: Integrated an AST-based symbol locator tool for precise code navigation.
10. **Optimized File Edit Routing**: Implemented explicit task type detection and routing optimization in MainAgent to improve handling of file editing tasks.
11. **Enhanced File Size Optimization**: Added file size thresholding of 100KB to optimize handling of large files and reduce processing overhead on big changes.
12. **Hybrid Symbol Locator**: Implemented a hybrid symbol location approach that combines ripgrep-based candidate identification with AST verification.
13. **Context-Sensitive Diff Minimization**: Completed the integration of context-sensitive diff minimization into the SmartEditor tool. Uses AST parsing for Python code changes and pattern-based analysis for other file types. Intends to reduce token usage by minimizing change context.
14. **Optimized AST Parsing in Hybrid Symbol Locator**: Implemented targeted AST parsing in hybrid_symbol_locator.py to process only relevant code regions instead of whole files.