SSEAL: Self-Supervised Explorative Agent Learning

Anonymous Authors[1]

Abstract

Large Language Models (LLMs) have concurrently demonstrated remarkable abilities to power autonomous, feedback-driven systems and learn complex patterns through in-context examples. Despite this progress, efficiently adapting LLM-based agents to ambiguous environments and new tasks remains challenging, and finetuning to each use case is impractical. We introduce Self-Supervised Explorative Agent Learning (SSEAL), a novel framework that enables LLM-based agents to autonomously explore, disambiguate, and learn their environment in a self-supervised manner. In a one-time **exploration phase**, the agent systematically generates and executes exploratory actions, and uses them to distill an optimized input prompt. This input includes (1) clarified task instructions, (2) updated environment context, and (3) few-shot trajectory examples, used in an **execution phase** to improve environment understanding. We evaluate SSEAL across the three diverse task domains of function-calling, robotics, and software engineering, and demonstrate that SSEAL improves performance on downstream tasks. Our work represents a step toward robust, self-supervised learning frameworks capable of improving performance in real-world, dynamic deployments. Our code and implementation is made available here.

1. Introduction

Recent developments in the transformer model architecture have shift the focus towards creating large, general models rather than small task specific models (Vaswani et al., 2023). For example Large Language Models (LLMs), have show incredible in-context learning and reasoning ability (Brown et al., 2020; Wei et al., 2023). Thus, LLMs have been applied to a wide variety of tasks. However, while LLM Agents have increasingly complex and general abilities, users still often deploy these models to specific use-cases and environments. Despite this, it is often too expensive or data-scarce to fine-tune these general models to optimize specific task performance. Therefore, we desire a method to adapt very general models to given specific environments and the tasks found in those environments – all in a fully self-supervised manner.

We consider the intuition that an agent's environment understanding can be separated from task execution. That is, instead of re-learning the environment at query time each time anew, an agent can first systematically explore and disambiguate an environment, retain this information, then execute on a set of queries. Moreover, we consider the strong in context learning and chain-of-thought capabilities of these general language models (Brown et al., 2020; Wei et al., 2023).

We introduce Self-Supervised Explorative Agent Learning (SSEAL), a framework to improve downstream agent performance by first employing a self-supervised exploration procedure to "fine-tune" the agent on a given task environment. Formally, consider some agent placed in environment context c in which it will execute actions on future input queries $\mathcal{Q}_c$. During SSEAL, we first let the agent explore environment context c, letting the agent learn how to operate on c successfully in an unsupervised manner. Then, this exploration should yields learning that increases subsequent performance on queries $\mathcal{Q}_c$. Importantly, SSEAL treats agents as a black box—we assume we cannot modify the weights of the model. Rather, SSEAL optimizes downstream performance by optimizing the input prompt $P := \{\rho_{\text{task}}, c, \xi\}$ where ρ_{task} is the task instructions, c is the defined environment context, and ξ are the few-shot examples, if any. Therefore, SSEAL allows for an agent to learn in-context in a self-supervised manner. The learned information is propagated to downstream queries for all time, optimizing long run task performance without supervision.

We show that the SSEAL framework can massively increase task performance— particularly in ambiguous environments. For example, on an adversarially perturbed task from Nexus-Bench, we seen SSEAL take GPT-4o's task accuracy from 5% to over 80%. Alternatively, on our more real world task

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

we see GPT-4o's performance increase from less than 20% to over 80%. Moreover, we show that stronger models can effectively transfer their learnings during SSEAL to other smaller and weaker models, allowing these weaker models to achieve similar task performance despite their smaller size.

In summary, our contributions are threefold:

1. We propose Self-Supervised Explorative Agent Learning (SSEAL), a novel learning algorithm for black-box agents with in-context learning ability.
2. We show examples where SSEAL can be effective deployed to increase performance in function calling, robotic manipulation, and software engineering tasks.
3. We demonstrate how SSEAL can be used to transfer performance from stronger models to weaker models at low cost.

2. Related Work

2.1. Agents

Large Language Models (LLMs) have demonstrated remarkable in-context learning abilities, allowing them to adapt to various tasks without requiring parameter updates (Brown et al., 2020). Additionally, chain-of-thought (CoT) prompting has been shown to significantly improve reasoning capabilities by guiding models to produce intermediate steps in problem-solving (Wei et al., 2023). Combining these insights, recent advancements have focused on developing reasoning-based agents.

ReAct (Reasoning and Acting) is a notable framework that leverages the CoT reasoning ability of LLMs to iteratively reason about their environment and act accordingly, leading to better performance in interactive tasks (Yao et al., 2022). These agents dynamically generate reasoning steps, observe the environment, and adjust their behavior, providing a strong foundation for adaptive problem-solving. While ReAct successfully integrates reasoning and action, its reliance on task execution during interaction limits its ability to pre-learn the environment context, which our SSEAL framework addresses.

More broadly, agent-based frameworks have been applied to tasks like tool use (Schick et al., 2023) and multi-step decision-making (Huang et al., 2022; Gautam et al., 2024).

2.2. Learning Through Exploration

Exploration-based methods have been widely used in reinforcement learning (RL) and autonomous agents to enable skill acquisition and environment understanding. Approaches like intrinsic motivation (Singh et al., 2004; Pathak et al., 2017) and self-play (Sukhbaatar et al., 2018) encourage agents to explore their environment by rewarding novel behaviors or self-improvement.

Voyager (Wang et al., 2023) demonstrates that LLMs can effectively explore and learn in open-ended environments, specifically within the Minecraft domain. Voyager employs exploration to acquire skills iteratively, enabling an agent to improve its performance over time. While similar in concept, SSEAL introduces a key distinction: it confines the exploration phase to the beginning of the task execution process for efficiency, thereby avoiding continuous exploration overhead. Moreover, we show SSEAL generalizes beyond a single domain and can be deployed across diverse task environments.

Recent work has also explored meta-learning techniques to enable agents to adapt quickly to new tasks (Finn et al., 2017). These methods aim to prepare agents for generalization by optimizing their learning processes during training.

3. SSEAL

Generally, given some environment and task, we will be given some preexisting environment context c, basic task instructions ρ_{task}, and an empty set of few-shot examples $\xi \leftarrow \{\}$. For complex environments, c and ρ_{task} will contain ambiguities. For example, consider a complicated API sent to a function calling agent. In this case, the API is c, and might not contain docstrings explaining how exactly to use the functions, or may under explain how functions interact. Likewise, the instructions to the model on how to call functions ρ_{task} might not properly explain exactly how the environment processes functions. The more complex an environment and task, the more difficult it is to define c and ρ_{task} such that no ambiguities are present. Thus, SSEAL uses self-supervised exploration to systematically identify and disambiguate the environment. While existing agent frameworks, such as ReAct (citation) employ chain-of-thought reasoning to overcome ambiguities, an agent must do so on a per-query basis. This means the agent needs may make the same mistake over and over again across many queries. With SSEAL, the exploration phase removes these issues before query time, and can execute on $\mathcal{Q}$ without error.

We detail the SSEAL in Algorithm 1. We let SSEAL start with some initial ρ_{explore}, ρ_{optimize}, ρ_{task}, where ρ_{explore}, ρ_{optimize} are engineered and assumed to be general enough to apply to the expected domain of input environment context $c \in \mathcal{C}$. The agent then explores the environment based on c, collecting a trajectory of actions (a) and observations (o) pairs $\vec{\tau}$. Then based on the collected trajectory of information, the agent constructs ρ_{clarify}, $\hat{c}$, and few-shot examples ξ, which are used to create the optimized prompt P^* for

query time downstream. After a high quality exploration procedure, we reason that subsequent performance on $\mathcal{Q}_c$ with using $P^* := \{\hat{\rho_{\text{task}}}, \hat{c}, \hat{\xi}\}$ will exceed the naive prompt $P := \{\rho_{\text{task}}, c, \xi\}$. We note that as $|\mathcal{Q}_c| \to \infty$ the additional cost of SSEAL is negligible.

Algorithm 1 SSEAL Algorithm

Initialize ρ_{explore}, ρ_{optimize}, ρ_{task}, c
$\vec{\tau} \leftarrow [\,]$
for each exploration iteration **do**
  $a \leftarrow \text{Agent}_{explore}(\rho_{\text{explore}}, c, \vec{\tau})$
  $o \leftarrow \texttt{execute}\,(a)$
  $\vec{\tau} \leftarrow \texttt{concat}\,(\vec{\tau}, [(a, o)])$
end for
$\rho_{\text{clarify}}, \hat{c}, \xi \leftarrow \text{Agent}_{explore}(\rho_{\text{optimize}}, c, \vec{\tau})$
$\hat{\rho_{\text{task}}} \leftarrow \rho_{\text{task}} \cup \rho_{\text{clarify}}$
$P^* \leftarrow \{\hat{\rho_{\text{task}}}, \hat{c}, \xi\}$
deploy $\text{Agent}_{execute}(P^*, q)$ for $q \in \mathcal{Q}_c$

4. Function Calling Environments

A common usage of LLM-as-agents is function calling, where an LLM is prompted to interact with some functional environment (Schick et al., 2023). This can vary from calling black box functions, to fully agent systems that can execute arbitrary code. Notably, successful function calling agent systems need to carefully tune prompts, function definitions, few-shot examples to optimize agent behavior, which is time consuming and costly. In the following section, we detail how we use SSEAL to optimize performance on function calling tasks.

4.1. Agent Framework

We use a ReAct framework to perform function calls (Yao et al., 2022). When attempting to solve a given user query, and agent is given the list of available functions in the environment (in context of SSEAL this is c), and any doc-strings that come with the functions. Additionally, the model is instructed to *"think step by step"* before output function calls to best satisfy the user query. The agent is then shown the result of those calls if any. Once the agent believes the user query has been satisfied, the agent ends the trajectory.

Eliciting Exploration: We elicit the SSEAL exploration loop through a meta ReAct loop before any queries are made. We prompt the model to explore the functional environment in order to understand how the environment should and could be used. We detailed the prompt below in Appendix Section A.1

In each exploration iteration, the ReAct agent reasons on which function calls it needs to make to further understand the environment. Then it submits the functions to the environment for execution. The environment then executes the functions and returns the results back to the model to reflect on function behavior. Based on the environment feedback, the agent can further submit more function calls to the environment.

Learning Distilation: While the exploration agent can learn the environment, the information learned during exploration needs to be efficiently passed to the downstream execution agent which will act upon the user queries. We cannot simply append the entire exploration trajectory to the execution agent because the exploration trajectory can be very long. This both stresses the context length of the agent and also may incur excessive input token costs. Therefore, in the function calling case, we propagate learned information via: (1) function doc-string modifications, (2) additional information/clarifications, (3) few-shot examples.

For a function calling task, our environment context c is designated by the available functions and their descriptions. To construct $\hat{c}$ after exploration during the SSEAL procedure, we prompt the exploration agent to rewrite the function doc-strings and argument types. The prompt is shown below in Appendix Section A.1.1.

To construct ρ_{clarify} we prompt the explore agent to document any clarifications surrounding the environments. For example, the agent may clarify that *"assignment statements are not allowed"* or "*always call* `ls` *to check which files are in the system before calling functions*". Finally, we construct $\hat{\rho_{\text{task}}}$ by appending ρ_{clarify} to ρ_{task}. We document the prompt for this task below in Appendix Section A.1.3.

To construct ξ we prompt the explore agent to generate few shot examples based on on its exploration trajectory $\vec{\tau}$. The examples allow the exploration agent to reason on possible down stream queries and how it should process them. This allows thr exploration agent to pass on this latent exploration information more readily. The prompt for generating examples is shown below in Appendix A.1.3.

4.2. Experiments

We test SSEAL in different environments.

1. **NexusBench:** Testing the SSEAL framework on function calling requires benchmarks with fully implemented function so that environment feedback is possible. As such, we utilize NexusBench's `LangchainRelational` task in which an agent satisfies user queries on a mock relational database. We additionally modified `LangchainRelational` into `LangchainRelational-NoSig` in which we remove the function signature to increase environment ambiguity. Finally, we construct `LangchainRelational-Adversarial` in which function names, argument names, and doc

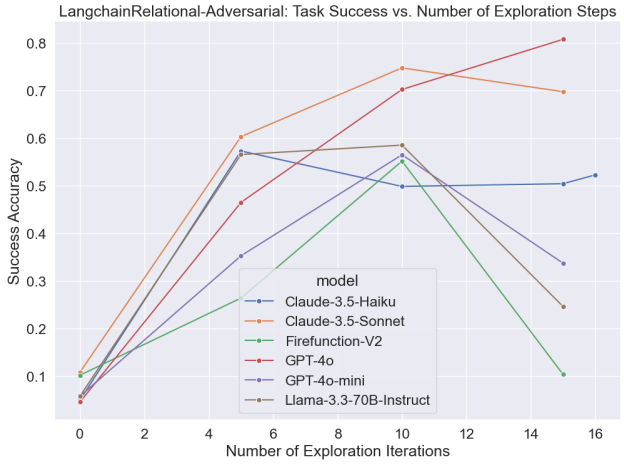

Figure 1. SSEAL effect on agent performance on `LangchainRelational-Adversarial`. 0 exploration iterations means SSEAL is not used.

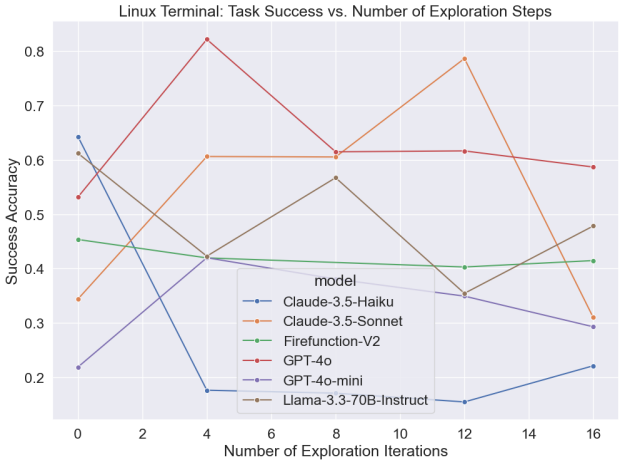

Figure 2. SSEAL on `LinuxTerminal` task: number of exportation iteration vs task execution accuracy.

strings are all obstructed, leaving only the argument types intact.

2. **Linux Terminal:** We additionally, construct a new task more representative of real use cases called `LinuxTerminal`, in which the agent needs to execute natural language queries on a simplified mock Linux filesystem environment, which altered function names. We construct the `LinuxTerminal` to stateful, requiring the end state to match the expected end state for the agent to be considered successful.

3. **Sports Data:** We construct an additional function calling task, `SportsData` where the agent is expected to execute a natural language query on a set of sports related dataframes using a provided customized python query API. Sports data is a particularly difficult task, and has many elements that go against the agent's presumptions of behavior.

We follow the function calling setup detailed in Section 4.1. To demonstrate adaptability to arbitrary function calling contexts, we use the same exploration and update instructions ρ_{explore} and ρ_{optimize}. We test our SSEAL framework on the above tasks.

4.3. Results

NexusBench:

We find that SSEAL is naturally very effective on `LangchainRelational-Adversarial`, with 5-8x performance improvements when utilizing exploration (Figure 1). In fact, GPT-4o and Claude-3.5-Sonnet, both the most expensive and capable models tests, can nearly reconstruct performance with respect to the non-adversarial case (Figure 7). Moreover, we find larger, more capable models are able to experience greater gain from SSEAL. Smaller models seem to show gain when the number of exploration steps is lower, but seem to "overfit" as exploration steps increase. We also find that on tasks that have very little ambiguity to begin with, such as `LangchainRelational`, SSEAL does not decrease performance when used on stronger models. However, smaller models may suffer from performance collapse.

Linux Terminal:

The `LinuxTerminal` task shows similar trends to the NexusBench tasks. In `LinuxTerminal` the task is much more complex compared to `LangchainRelational`. Moreover, `LinuxTerminal` requires the agent to manage a state, which may be different from the exploration state, causes further difficulties at test time. However, referencing Figure 2, stronger models such as GPT-4o and Claude-3.5-Sonnet show marked improvements after undergoing SSEAL. GPT-4o increases task accuracy from 53% to 82% with just 4 exploration steps. Claude-3.5-Sonnet increases score from 35% to 76% after 12 steps of SSEAL. Beyond the optimal steps for each model, SSEAL procedure begins to overfit. However, over-fitted performance is generally still near or above baseline for these models.

The increased difficulty of `LinuxTerminal` shows the limitations of using weaker models for SSEAL. None of the weak models are able to improve themselves. Notably, Claude-3.5-Haiku performance drops to effectively 0.

Sports Data:

Considering Figure 3, we notice that similar to

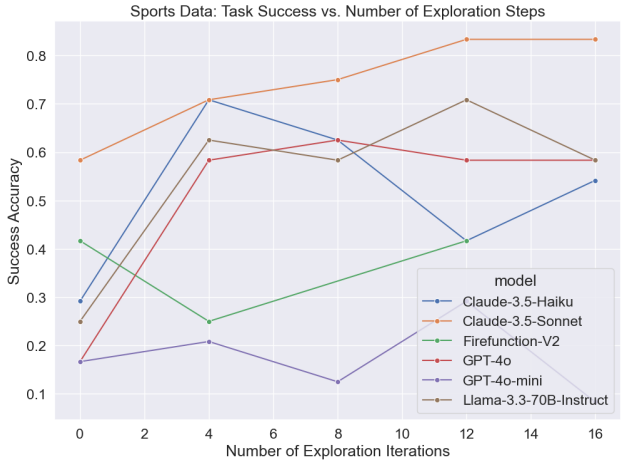

Figure 3. SSEAL on `SportsData` task: number of exportation iteration vs task execution accuracy.

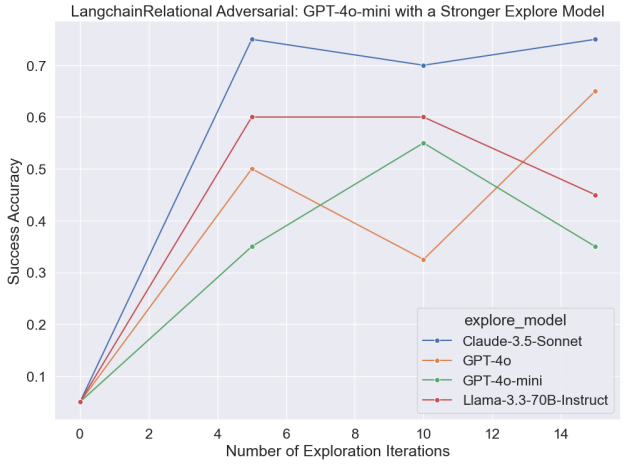

Figure 4. `LangchainRelational-Adversarial` task accuracy vs exploration steps for different exploration agents, while always using GPT-4o-mini as the agent during task execution.

`LinuxTerminal`, the more complex `SportsData` task makes improvement significantly more difficult for weaker models. Notably, Claude-3.5-Sonnet is able to achieve monotonically increases performance as it undergoes more SSEAL exploration steps. Other stronger models, GPT-4o and Llama-3.3-70B-Instruct show saturating performance increases, with little overfitting. Alternatively, weaker models such as Claude-3.5-Haiku, GPT-4o-mini, and Firefunction-V2 see overfitting behavior or little improvement to begin with.

4.4. Mixing Models

Before, we considered a simple setup where both $\text{Agent}_{explore}$ and $\text{Agent}_{execute}$ use the same underlying LLM. However, since $\text{Agent}_{explore}$ passes learned information through an optimized task prompt, $\text{Agent}_{explore}$ can use a different model that $\text{Agent}_{execute}$. We leverage the assumption that exploration is a more difficult task than query execution after exportation has already done the brunt of the reasoning work. Additionally, consider that $\text{Agent}_{explore}$ only needs to run once per environment, while $\text{Agent}_{execution}$ runs for all queries for all time. Therefore, we consider a setup where $\text{Agent}_{explore}$ is a stronger, more costly model, and $\text{Agent}_{execute}$ is a cheaper model. We demonstrate this setup can achieve performance similar to always using a strong model, and much better than always using a weak model.

In Figure 4, Figure 9, and Figure 5 we should the results of using stronger models during exploration while switching to GPT-4o-mini, the cheapest model, during execution on each task environment. We show that using a stronger model only during exploration can greatly increase task performance. In particular with Figure 4, we see that on the `LangchainRelational-Adversarial` environment, using Claude-3.5-Sonnet as the exploration model allows GPT-4o-mini to approximately recover similar performance compared to using Claude-3.5-Sonnet at execution time as well.

Appendix Figure 9 shows that even in more complex environments, the model mixing strategy gives massive performance improvements for weaker execution models, albeit with seemingly less stability with respect to the number of iterations. Using both Claude-3.5-Sonnet and GPT-4o yield Pareto frontier performance compared to just using GPT-4o-mini for exploration, with both strong models pushing GPT-4o-mini's performance from a previous best of 35% to around 70%— doubling task accuracy for negligible long run cost.

Again, in Figure 5 we see using a strong model for just exploration yields a huge performance increase. In particular, at its best, Claude-3.5-Sonnet is able to teach GPT-4o-mini to reconstructs its own full performance, increasing GPT-4o-mini's peak score by 280%.

This suggests that stronger models can successful transfer their learned information to smaller models through prompting alone. SSEAL provides a method to break problems into ambiguous and disambiguated sub-parts, allowing for the hierarchal usage of strong and weak models to handle exploration and execution, respective, thereby optimizing price against performance.

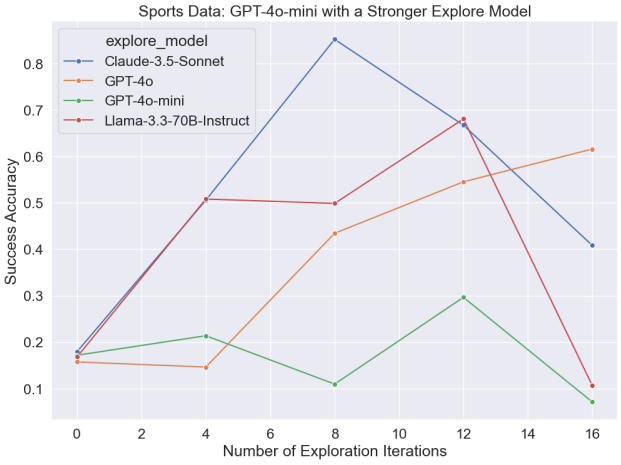

Figure 5. `SportsData` task accuracy vs exploration steps for different exploration agents, while always using GPT-4o-mini as the agent during task execution.

5. Towards Real World Deployment

Beyond basic function-calling tasks, we explore how SSEAL can be leveraged for two complex, real-world tasks.

5.1. Robot Agents

We first investigate the application of SSEAL to a robotic manipulation task, focusing on how an agent can learn to effectively utilize a Vision-Language-Action (VLA) model as a tool to complete more general robotic tasks. More specifically, we set up a central planning agent that can send language instructions to the VLA, observe the outcomes of the robotic trajectory executed by the VLA, and iteratively learn from such observations.

This environment poses two unique challenges. (1) The VLA's behavior, unlike traditional tools, cannot be easily documented or described, making it difficult for a planning agent to understand its capabilities. (2) The temporal horizon of action execution is unknown, meaning the agent is unsure how long to wait before evaluating the outcome of an action or sending a new action. Both of these challenges, which would be difficult for a traditional agent framework to handle, are what motivate us to apply autonomous exploration.

5.1.1. Setup

For this application of SSEAL, we deploy the agent-based system illustrated in Figure 6. The VLA operates at high frequency, executing instructions provided by the planning agent and writing third-person images of the environment to a shared memory buffer. Concurrently, a Vision-Language Model (VLM) periodically analyzes the trajectory of images

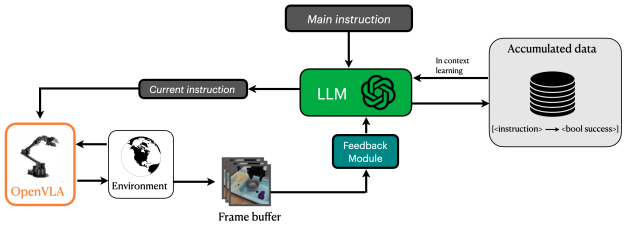

Figure 6. **VLA as a tool** We design a system where an agent sends instructions for the VLA to execute, observes the resulting robotic trajectory, and updates its memory of what the VLA is capable of.

and provides updates to the agent on whether the current instruction was successfully completed, failed, or is still in progress. The agent, upon receiving this feedback, decides what instruction to send to the VLA next.

We utilize the OpenVLA model (Kim et al., 2024), a state-of-the-art open-source VLA, and the simulated Libero environment and task suites (Liu et al., 2023). The Libero environment provides rendered third-person images as input to OpenVLA, as well as ground truth success information for each task.

5.1.2. SSEAL Implementation

We implement SSEAL by initializing an exploration phase where the agent interacts with the VLA to identify the space of instructions the model can reliably execute, and then use these learnings for downstream query execution.

The exploration proceeds as follows: (1) The agent, given an image of the scene, generates a set of exploratory tasks designed to probe the VLA's behavior. (2) The VLA executes the proposed tasks, while the VLM-powered feedback module classifies the success or failure of each task based on rendered images from the environment. (3) The agent receives the resulting task-success tuples, which it ultimately uses as an in-context learning prompt for downstream tasks.

We then evaluate the agent's ability to disambiguate complex instructions, and show that SSEAL offers a significant performance improvement.

5.1.3. Experiments

Our primary experiment evaluates whether the agent can learn to translate high-level instructions into prompts that the VLA can execute successfully. To do so, we conduct a three-part study:

Part 1: Evaluation. We first seek to evaluate how important prompting language is to the VLA. We take the original task suite of instructions, and use an LLM to generate a

rephrased dataset where each instruction gets mapped to a slightly rephrased version of itself. For example, the original task *"put the wine bottle on the rack"* might get rephrased to *"locate the wine bottle and transfer it to the wine rack"*. We evaluate the VLA on this dataset and find that the success rate drops from 77.5% to 40.6%. The prompting language proves to be extremely important, and therefore we see a unique opportunity for autonomous learning and improvement. We use this rephrased dataset as a baseline and measure the agent's ability to prompt the VLA successfully.

Part 2: Privileged Exploration. Next, we investigate whether an agent can learn to prompt the VLA if given privileged information about the success outcome of each instruction. We sample instruction-success tuples collected in the previous experiment to create an in-context learning prompt, that asks the agent to analyze these outcomes and output an updated instruction. With this method, the average success rate across tasks improves significantly, with a **17% increase** over the baseline instructions. However, this method is privileged in two regards: (i) the success values are ground-truth labels provided by the simulator, which is not possible in a real-world deployment, and more importantly (ii) the agent did not choose to execute these instructions, they were chosen in a supervised manner.

Part 3: Autonomous Exploration. To apply our SSEAL framework, we remove the assumption of ground-truth success information from the simulator and ask the agent to explore the instruction space of the VLA on its own. Importantly, the agent starts off with absolutely no information of what the VLA is capable of, what tasks it was trained on, or what a VLA even is. As described in Section 5.1, the agent first generates a wide range of exploratory tasks, sends them to the VLA to execute, and the VLM feedback module classifies them as successful or not. Similar to part 2, these explored instruction-success tuples are used for in-context learning, and we evaluate the agent's ability to output a successful instruction. As seen in Section 1, SSEAL leads to a **9% increase** in success rate compared to baseline instructions.

Method	Success Rate
Baseline Instructions	40.6%
Privileged Exploration	57.5%
Autonomous Exploration (with SSEAL)	49.4%

Table 1. Success rates for different instruction methods. Autonomous exploration leads to a significant improvement over baseline instructions.

The results highlight the efficacy of SSEAL in systematically learning to use VLAs as tools. By autonomously disambiguating the instruction space through exploration, SSEAL can overcome some of the limitations of VLAs' sensitivity to language variations. While the privileged supervision experiment serves as an upper bound, the autonomous exploration results demonstrate that meaningful gains can be achieved without privileged information. Furthermore, simply asking the agent *"what is this robotic model capable of?"* after the SSEAL process leads to a well-defined and accurate response that would likely not be possible without exploration.

However, we do observe that the VLM feedback module introduces occasional inaccuracies, particularly when evaluating edge-case tasks proposed by the agent. This suggests that future improvements to off-the-shelf VLMs could further enhance the performance of SSEAL in robotic environments.

5.2. Software Engineering Agents

Recent works in AI for code have moved past autocompletion-based models and introduced end-to-end code-editing systems with strong performance (Gauthier, 2024; Wang et al., 2024; Arora et al., 2024; Chen et al., 2024; Brown et al., 2024). However, similar to the other agent scenarios we explore for function calling, these systems are often initialized with an understanding of the task they will perform through user generated examples in-context or previous trajectory demonstrations (Yang et al., 2024; Dehghani et al., 2021; Kapoor et al., 2024). We aim to remove this dependency by extending SSEAL to SWE-agent; we choose SWE-agent due to its high open-source scores on various benchmarks (Yang et al., 2024). We evaluate on SWE-bench-lite, the most commonly used bug-fixing benchmark for software engineering agents (Jimenez et al., 2024).

Due to cost limitations, we run baselines using GPT-4 Turbo with 1 exploration iteration, where the list of function APIs is given to the agent, and it tests each one as many times as the model sees fit in a testbed repository. We share prompts and more details in Appendix Section A.1.4. SSEAL improves performance over having no demonstration and is comparable to a successful trajectory in the context window of an agent[1].

Demonstration Type	Resolution Rate
No Demonstration	16.33%
Demonstration	18.00%
SSEAL	**18.67%**

Table 2. SWE-agent scores on SWE-Bench-lite with ablations on the in-context demonstration.

We find that exploration critically helps in the function call-

[1]We use numbers as reported in (Yang et al., 2024).

ing behavior of an agent. Specifically, previous work has acknowledged the difficulty of editing steps for coding agents, and we find that using editing tools beforehand and communicating that understanding through an in-context example limits the percent of trajectories with failed edit steps from 51.7% to 44.6%.

5.2.1. Adapting to Environment Changes

SSEAL, by functioning as a replacement to the in-context example, empirically teaches SWE-agent new explorative behaviors. To test this, we also explore the introduction of new agent tools during the execution of a task. Building on related work about environment policies that communicate updates to deployed agents, we similarly pass text to the agent that a new tool is available for use by the agent. We empirically observe that our agents can generalize to new functions and trigger a SSEAL-like exploration iteration. We leave it to future work to explore online tool learning in extended setups and benchmark similar empirical findings.

6. Conclusion

In this work, we present a novel framework enabling LLM agents to autonomously adapt to ambiguous environments without fine-tuning or human interventions. SSEAL leverages self-supervised exploration to clarify task instructions, refine environment contexts, and generate effective few-shot examples. Importantly, this information is efficiency distilled into a single prompt for all future use cases.

Experiments across function calling, robotics, and software engineering consistently demonstrate that SSEAL significantly boosts performance on downstream tasks. Furthermore we show SSEAL with a stronger model can improve downstream performance of a weaker model, a powerful paradigm for balancing cost with performance. We open-source our implementation to facilitate further research into agent learning and adaptive systems. SSEAL represents a significant step towards realizing self-adaptive agents capable of thriving in dynamic real-world settings.

7. Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Arora, D., Sonwane, A., Wadhwa, N., Mehrotra, A., Utpala, S., Bairi, R., Kanade, A., and Natarajan, N. Masai: Modular architecture for software-engineering ai agents, 2024. URL https://arxiv.org/abs/2406.11638.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Chen, D., Lin, S., Zeng, M., Zan, D., Wang, J.-G., Cheshkov, A., Sun, J., Yu, H., Dong, G., Aliev, A., Wang, J., Cheng, X., Liang, G., Ma, Y., Bian, P., Xie, T., and Wang, Q. Coder: Issue resolving with multi-agent and task graphs, 2024. URL https://arxiv.org/abs/2406.01304.

Dehghani, M., Tay, Y., Gritsenko, A. A., Zhao, Z., Houlsby, N., Diaz, F., Metzler, D., and Vinyals, O. The benchmark lottery, 2021. URL https://arxiv.org/abs/2107.07002.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks, 2017. URL https://arxiv.org/abs/1703.03400.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Moghaddam, R. Z. Refactorbench: Evaluating stateful reasoning in language agents through code. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Gauthier, P. Aider: Ai-powered coding assistant, 2024. URL https://github.com/paul-gauthier/aider.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022. URL https://arxiv.org/abs/2201.07207.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., and Narayanan, A. Ai agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

Kim, M. J., Pertsch, K., Karamcheti, S., Xiao, T., Balakrishna, A., Nair, S., Rafailov, R., Foster, E., Lam, G., Sanketi, P., et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Liu, B., Zhu, Y., Gao, C., Feng, Y., Liu, Q., Zhu, Y., and Stone, P. Libero: Benchmarking knowledge transfer for lifelong robot learning, 2023. URL https://arxiv.org/abs/2306.03310.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction, 2017. URL https://arxiv.org/abs/1705.05363.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Singh, S. P., Barto, A. G., and Chentanez, N. Intrinsically motivated reinforcement learning. In *NIPS*, pp. 1281–1288, 2004. URL http://papers.nips.cc/paper/2552-intrinsically-motivated-reinforcement-learning.

Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. Intrinsic motivation and automatic curricula via asymmetric self-play, 2018. URL https://arxiv.org/abs/1703.05407.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. Opendevin: An open platform for ai software developers as generalist agents, 2024. URL https://arxiv.org/abs/2407.16741.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

A. SSEAL Prompts

A.1. Function Calling Prompts

A.1.1. Exploration Prompt

```
You are an AI agent tasked with exploring a function calling environment.  Your
goal is to analyze the provided functions, understand their potential uses, and
propose function calls to resolve any ambiguities or missing information.  This
process is similar to running unit tests to better understand the function calling
environment.
Here is the list of functions you have access to:

<functions>
{{FUNCTIONS}}
</functions>

Please follow these steps:
1.  Analyze the provided functions:
- Review each function's name, description, arguments, and return types (if
provided).
- Identify any missing or ambiguous information, such as unspecified return types or
unclear argument formats.
2.  Propose function calls:
- For each function, suggest at least one function call that would help clarify its
behavior or resolve ambiguities.
- If a function's return type is missing or unclear, propose a call with example
input to observe the return type.
- If a function's argument format is ambiguous, propose multiple calls with different
input formats to determine the correct usage.
Please structure your outputs as follows:

<exploration_summary>
<function_analysis>
For each function, describe the function call you propose that will help clarify its
behavior, and what you hope to learn from the results
</function_analysis>
<function_list>
A series of your chosen function calls, in python syntax, separated by newlines.
For example
f(1)
g()
h(4, 'a')
</function_list>
</exploration_summary>
```

A.1.2. Optimization Prompt: Context

```
Modify the list of function contexts provided.  Specifically add/improve doc-strings
and argument types and return types.  You should be very detailed, consider what
ambiguities gave you trouble at the beginning.  Give example argument inputs,
example function outputs, and observed error cases.  Make sure it is absolutely
clear when and how to use each function.  Make sure to include all functions.
```

A.1.3. Optimization Prompt: Examples

```
<examples>
Give reasonable example user queries on the enviornment.  Then show can they can be
answered step by step through calling functions in the environment, in the same way
you have done so above.  Be detailed on the process and reasoning.
<examples>
```

A.1.4. Optimization Prompt: Clarifications

```
Any clarifications, learnings, guides etc.  Focus on how to interact with the
environment.  Explain to a future agent put in the same environment how they might
go about answering user queries sent to the environment.  The future agent will also
only be able to submit functions in the <function_list> tags and receive environment
feedback.
```

A.1.5. SWE-agent prompt changes

```
Here's how I can use some of the commands.  I have to keep the DISCUSSION and tool
call sections exactly as below.

Here's an example of using the ls function:

DISCUSSION
Let's see what files are in the current directory.
```
ls -a
```

...

...

...

Here's an example of using the edit function:

DISCUSSION
Let's try editing filename.py.
```
edit <start_line>:<end_line>
<replacement_text>
end_of_edit
```
```

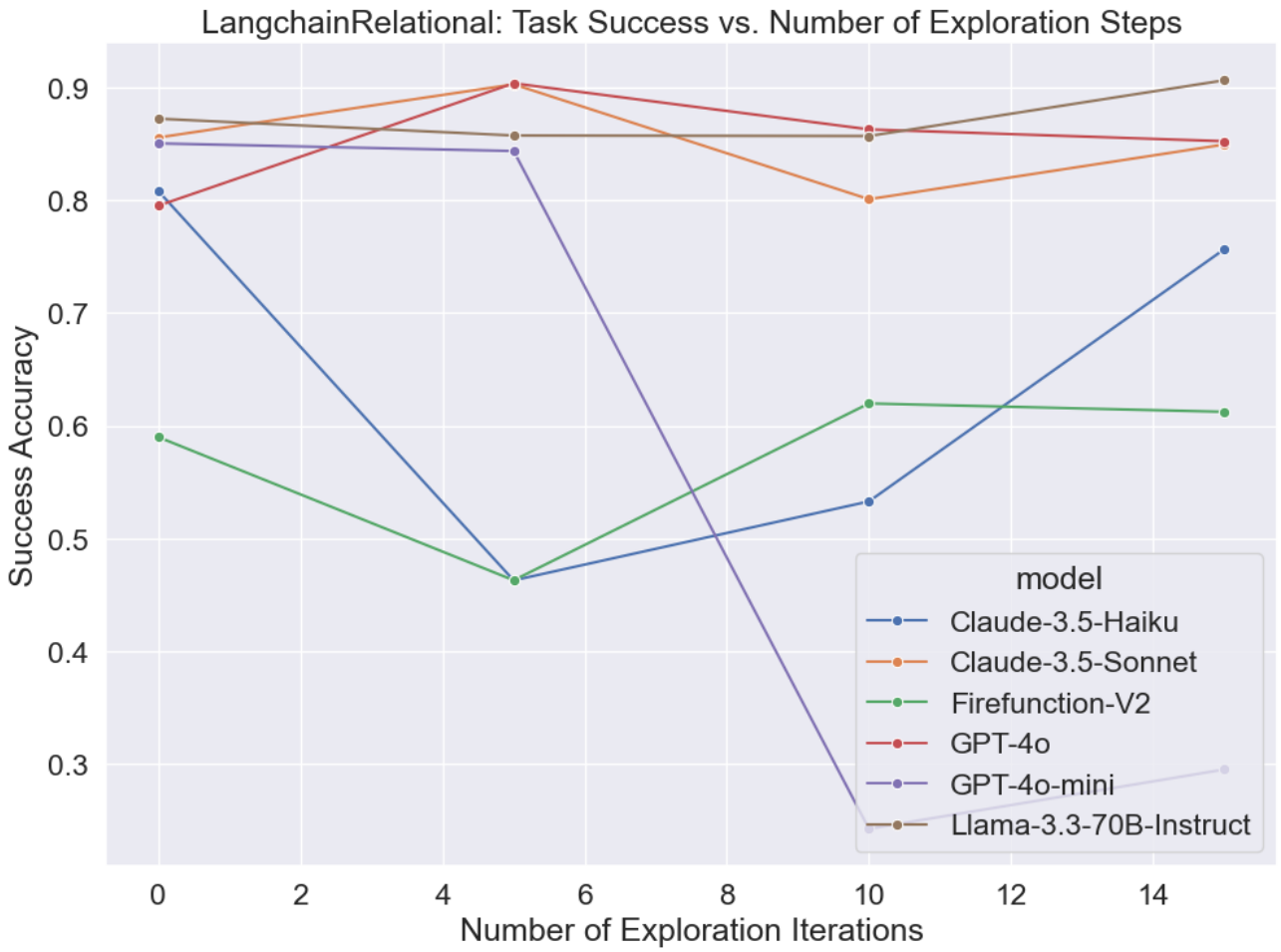

Figure 7. SSEAL affect on agent performance on `LangchainRelational`, a simple environment with little ambiguity.

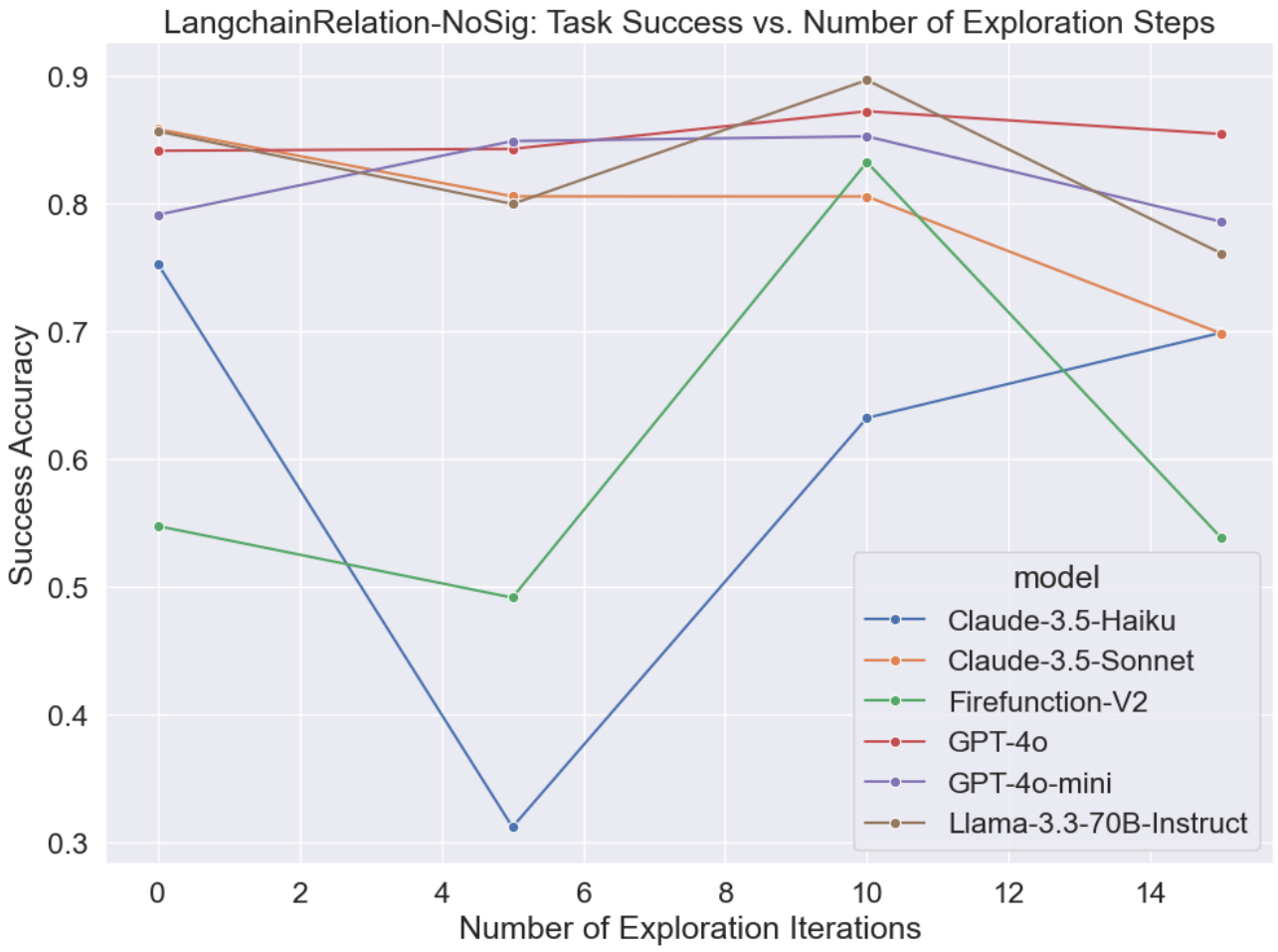

Figure 8. SSEAL affect on agent performance on `LangchainRelational-NoSig`.

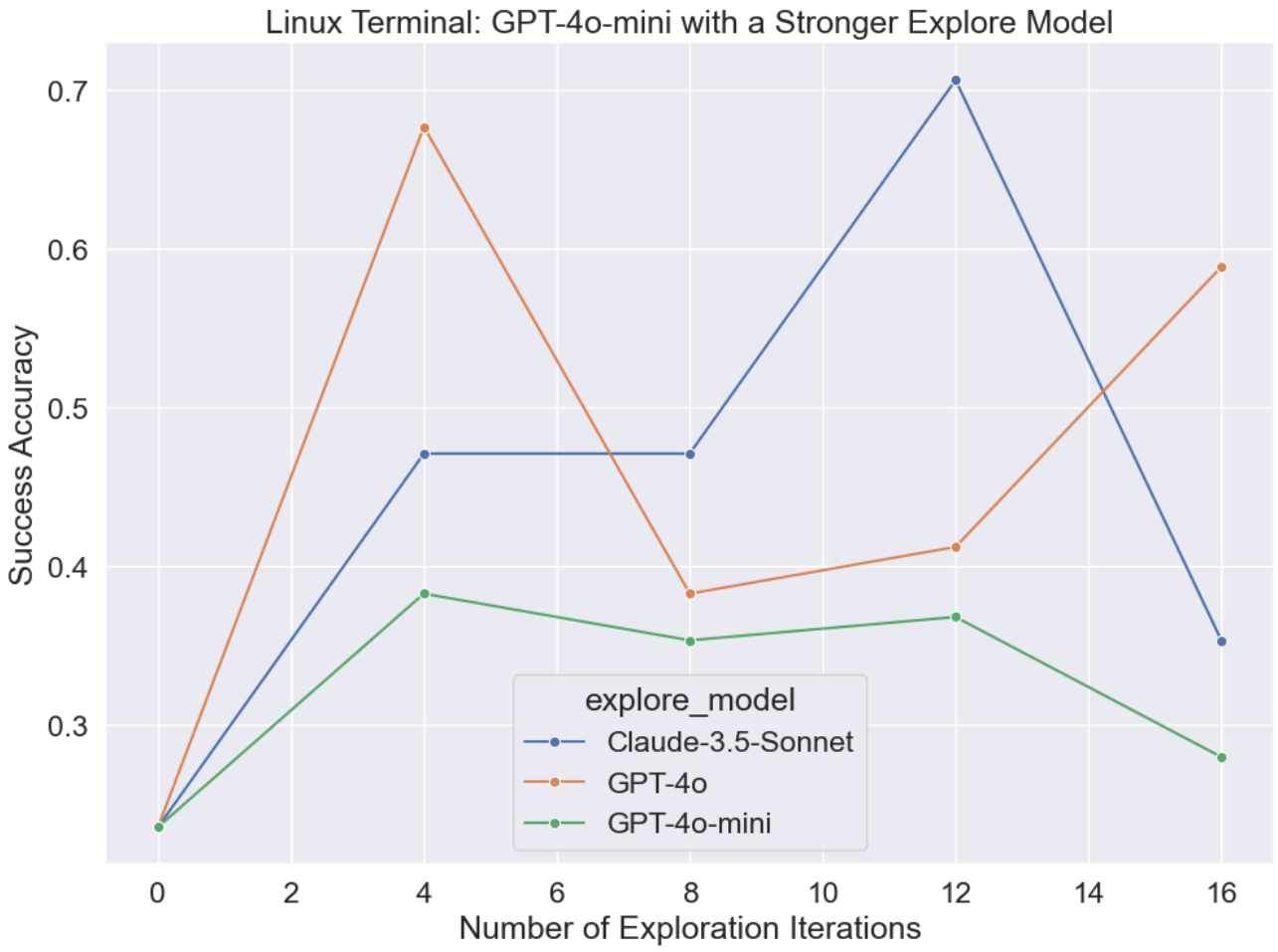

Figure 9. `LinuxTerminal` task accuracy vs exploration steps for different exploration agents, while always using GPT-4o-mini as the agent during task execution.