ProST: Progressive Sub-task Training for Pareto-Optimal Multi-agent Systems Using Small Language Models

Biddut Sarker Bijoy[1] **Mohammad Saqib Hasan**[1] **Pegah Alipoormolabashi**[1]
Avirup Sil[2 *] **Aruna Balasubramanian**[1] **Niranjan Balasubramanian**[1]
[1]Stony Brook University [2]Oracle Applied Science
[1]{bbijoy, mdshasan, palipoormola, arunab, niranjan}@cs.stonybrook.edu, [2]avi.sil@oracle.com

Abstract

Multi-agent systems with smaller language models (SLMs) present a viable alternative to single agent systems powered by large language models (LLMs) for addressing complex problems. In this work, we study how these alternatives compare in terms of both effectiveness and efficiency. To study this trade-off, we instantiate single and multi-agent systems for the complex problems in the AppWorld environment using different sized language models.

We find that difficulties with long-trajectory learning in smaller language models (SLMs) limit their performance. Even when trained for specialized roles, SLMs fail to learn all subtasks effectively. To address this issue, we introduce a simple progressive sub-task training strategy, which introduces new sub-tasks progressively in each training epoch. We find that this novel strategy, analogous to instance level curriculum learning, consistently improves the effectiveness of multi-agents at all configurations. Our Pareto analysis shows that fine-tuned multi-agent systems yield better effectiveness-efficiency trade-offs. Additional ablations and analyses shows the importance of our progressive training strategy and its ability to reduce subtask error rates.

1 Introduction

Solving complex problems requires a wide-range of capabilities including planning sub-tasks, acting in an environment via coding, reasoning about the actions, recovering from errors, as well as tracking and managing the execution of sub-tasks. LLMs with their strong coding and reasoning abilities and advances in handling long contexts have shown impressive breakthroughs on such complex problems (e.g., `AppWorld` (Trivedi et al., 2024) and `SWE-Bench` (Jimenez et al., 2024)). However, these large models incur high computational and API costs for both training and inference. Another viable alternative is to use multi-agent solutions with smaller language models[1] (SLMs). The core idea is to identify specialized roles in solving these complex tasks (e.g., orchestration, code writing, and critiquing) and use separate SLMs (or SLM calls) for each role. Belcak et al. (2025) argue that SLMs are much more cost-effective and practical for training, adapting, and deploying multiple specialized experts for different agentic routines. Shen et al. (2024) show these multi-agent solutions with SLMs can be effective for complex problems requiring tool use via APIs.

We consider two challenges that arise in this context. First, for SLMs, even specialized roles can be difficult to learn. Complex problems often have multiple subtasks resulting in long trajectories. Despite their improving capabilities, SLMs still struggle to learn all subtasks under standard fine-tuning. Second, while the per-token compute depends on the size of the models, the overall computational cost also depends upon the nature of their solutions (e.g., their length). Therefore, it is important to explicitly consider both the effectiveness and efficiency of these solutions to better understand the trade-offs.

We address these challenges using `AppWorld` benchmark. We design a multi-agent solution with three agents: an Orchestrator that plans subtasks incrementally, an Executor that interacts with the environment via code to solve subtasks, and a Critic agent that critiques the Executor output. We make two specific contributions on this multi-agent setup:

1) Progressive Sub-task Training To address the inconsistent performance on large trajectories, we introduce a progressive training strategy. We find that SLMs are unable to solve some subtasks of these complex problems, even in role-

*Work done while at IBM Research AI.

[1]Small and large are subjective terms. For this paper we use SLM for models with fewer than 40B parameters.

Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 3357–3375

specialized learning settings. We propose a simple curriculum-style learning strategy, where we introduce sub-tasks progressively in training epochs. This allows the small capacity SLMs to gradually expand their learning, covering all aspects of the problem trajectories.

2) Pareto Analyses We compare different instantiations of single and multi-agent systems. In particular, we plot the efficacy of different solutions against inference-time `FLOPs`. The Pareto curve for these solutions helps choose the best solution at different compute budgets.

We evaluate these ideas using `Qwen-2.5 Coder`, `Llama-3.1` and `Phi-4` models. We distill training trajectories for a single agent from a frontier model (`Claude-3.7-Sonnet`). Then, we translate these into role-specific multi-agent training trajectories for three agents, namely, an Orchestrator (planner), an Executor (coding) agent, and a Critic agent. Our evaluations yield multiple interesting observations `AppWorld` tasks: (i) While larger models tend to have higher effectiveness, multi-agent solutions can achieve better efficiency versus effectiveness trade-offs. (ii) Progressive Subtask Training yields a better Pareto-front compared to standard fine-tuning. (iii) Stronger (higher compute) agents for planning are more effective (and efficient) than using stronger coding and critic agents when fine-tuning using Progressive Subtask Training.[2]

2 Related Work

SLM-based Agents SLMs are being used as agents due to lower computational cost, but struggle with complex planning and tool use (Shen et al., 2024), skills essential for agents. Hence, many strategies are proposed: separation of roles to only tackle small, concrete tasks (Shi et al., 2024; Qiao et al., 2024); distilling training data from powerful LLMs(Shi et al., 2024), Direct Policy Optimization (DPO)(Feng et al., 2025); online learning by generating data from unsuccessful attempts(Qi et al., 2025), masking loss from erroneous steps(Fu et al., 2025); learning only from critical steps (Chen et al., 2025d). Contrary to these works, we formulate `ProST` which optimizes SLMs, as part of multi-agent design, by initiating with important parts of the trajectory and iteratively increasing steps, a strategy never tried for agents.

Multi-Agent Optimizations Many works propose techniques to make multi-agents more robust. These techniques include dynamically reducing redundant agents and messages (Wang et al., 2025c; Zhang et al., 2024a), iterative answer refinement through feedback(Chen et al., 2025b), optimizing collaboration strategies(Wang et al., 2025a; Zeng et al., 2025) and communication protocols (Xiao et al., 2025), and test-time search for effective tool planning (Chen et al., 2025a). Recent works propose using cost-effective SLMs for multi-agent setups: Erdogan et al. (2025) uses smaller`32-B` models as planning and execution subagents; Belcak et al. (2025) argue for SLM multi-agents based on their improved capabilities, cost, ease of adaptability, and deployment. Our work introduces `ProST`, a training algorithm for SLMs in multi-agent setups that iteratively trains on subtasks. Unlike previous works, we show how `ProST` produce SLMs effective at solving tasks as well as computationally efficient.

Efficiency of Multi-agents Recent works have sought to assess and improve the computational efficiency of multi-agent systems. For example, Wang et al. (2025c) use token consumption as a metric to prune redundancies in multi-agent collaborations. Yue et al. (2025) show, via Pareto fronts, that their language model routing strategy in multi-agent systems reduces dollar cost while improving accuracy. Wang et al. (2025a) use *token accuracy ratio* to compare multi-agent collaboration strategies. In contrast, we analyze computational cost against task accuracy to measure *Pareto optimality*. Based on this, we design `ProST`, which gives better performance accuracy with higher computational efficiency (more *Pareto optimal*).

3 Multi-Agent Design for AppWorld

Multi-agent systems using SLMs instead of LLMs have emerged as an alternative for better cost-performance trade-off (Erdogan et al., 2025; Belcak et al., 2025). However, optimizing SLMs for long and complex reasoning tasks remains challenging(Shen et al., 2024). In this work, we focus on `AppWorld` (Trivedi et al., 2024), an agent environment of complex tasks requiring coding and ability to interact with APIs. Our method is based on two considerations: (i) SLMs are less effective at essential agent skills like coding, long-context and recovering from errors; (ii) overall computation cost is based on both solution length and low

[2]Code and data available at https://github.com/StonyBrookNLP/prost-multiagents

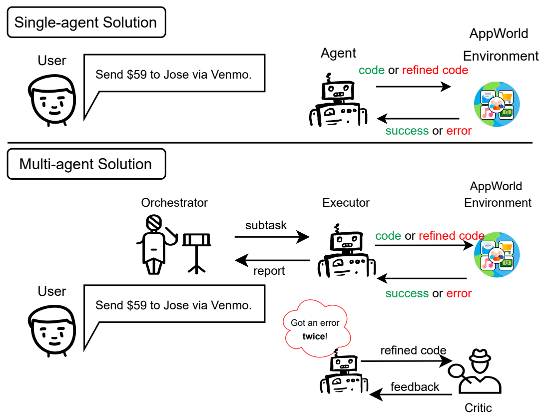

Figure 1: Comparison of single-agent and multi-agent architectures for solving complex tasks. The single-agent approach (top) employs a monolithic LLM that directly processes user given task and interacts with the AppWorld environment by generating code and processing the output. The multi-agent approach (bottom) decomposes this functionality into three specialized components: an Orchestrator decomposes the user query into subtasks (one subtask at a time), an Executor that solve the given subtask step by step (one step at a time) by interacting with environment, and a Critic that evaluates and provides natural language feedback on the Executor's outputs when required.

per-token cost of SLMs.

To address these challenges we: (i) use a tri-agent system for AppWorld (subsection 3.1), (ii) introduce Progressive Subtask Training, a novel training loss that improves SLMs on complex long trajectory reasoning (section 4), and (iii) use Pareto analysis to understand the cost-performance trade-offs in agent systems (section 5).

3.1 Multi-agent Architecture

Solving AppWorld tasks require identifying subtasks that can solve each complex problem via writing code and self-reflection (Madaan et al., 2023). Agents in multi-agent systems work in collaboration to solve such tasks, where each has its specific responsibilities. Such architectures have been used in Qiao et al. (2024) (plan, tool, and reflect agents) and Shi et al. (2024) (grounding, execution, and review agents). Accordingly, our multi-agent system (refer to Figure 1) also comprises of similar elements: an Orchestrator that generates and delegates sub-tasks, an Executor that writes code to interact with AppWorld, and a Critic that provides feedback to Executor (see prompt in Figure 10).

Orchestrator This agent decomposes the task into subtasks with specific goals and execution plan. Orchestrator operates iteratively, executing one subtask after another and handling execution failures by *dynamic decomposition strategy*,i.e., further decomposing the subtask (Prasad et al., 2024).

Executor Agent responsible for solving each subtask generated by Orchestrator. It does so by writing code according to the plan, executing in AppWorld and observing the results (see prompt in Figure 11). Each subtask may need multiple iterations of such cycle. In case of error, Executor refines code. If error still persists it revises the code with feedback from Critic. Execution stops when either task is complete or maximum number of steps is reached.

Critic Agent that acts when Executor fails during refinement. When prompted by Executor (see Figure 12), Critic reviews the main task, subtasks, and the full trajectory of the Executor so far, as well as the Executor's refinement. Critic then provides natural language feedback on the Executor's suggested resolution.

4 Progressive Subtask Training

We fine-tune SLMs using multi-agent training trajectories derived from LLMs (described in subsection 5.2). Standard fine-tuning trains each agent to maximize likelihood of the role relevant gold trajectory. However, as we metioned, the trajectories remain long, even with role specialization. SLMs' inability to process long trajectories (Chen et al., 2025d) limits training them for their respective roles. In particular, we observe that standard finetuning leads to higher error rates for the more difficult middle subtasks compared to first and last subtasks. This observation motivates our proposed solution, drawing from curriculum learning.

Our approach is to use incremental curriculum on subtasks of each training instance. This ensures models train on subtasks subset in each iteration. We achieve this by randomly focusing on a set of subtasks at each epoch. However, later subtasks often depend on outputs of earlier ones, thereby requiring either understanding those tasks or accessing their output. For example, in task to send text message, selecting contacts presupposes retrieval of contact list. This dependency suggests that introducing subtasks into the curriculum in their natural order within the tasks is more effective, where the models learn the earlier subtasks

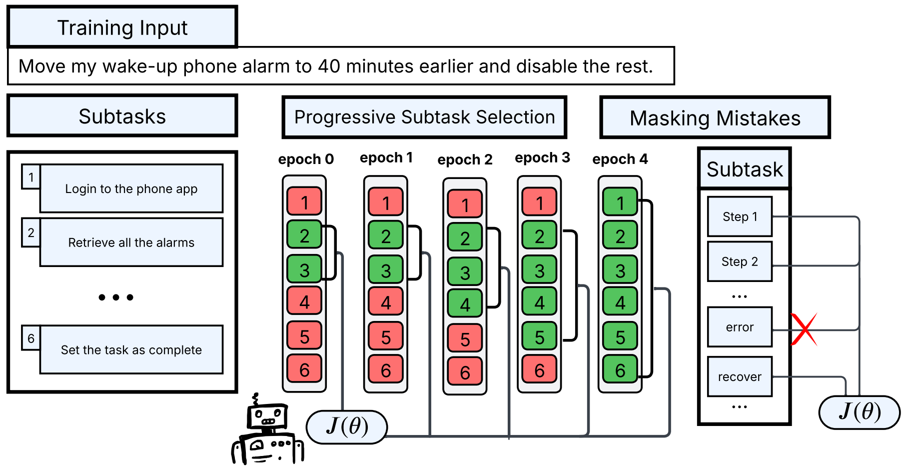

Figure 2: The figure shows how ProST trains models by progressively increasing the subtasks (highlighted in green) in training trajectory. The agent is trained on subtasks 2 and 3 in epoch 0, ignoring subtask 1 as it is not important(login tasks in AppWorld). With each increasing epoch, subtasks presented to the model for training is increased from subtask 2 onwards. In the final epoch, the full trajectory with all the subtasks are presented. Any step of a subtask that is erroneous is ommitted during training. In this way, model learns to: i) to solve the most important subtasks first, ii) sequentially learn to solve the larger problem at hand, and iii) avoid optimizing for solving subtasks erroneously.

first, followed by the later ones.

$$J_O^{(e)}(\theta, x) = \sum_{i=1}^{M_x} \mathbb{I}(i \in S^{(e)})\ log(\pi_\theta(s_i | I, \{s_j, r_j\}_{j<i})) \quad (1)$$

$$J_O^{(e)}(\theta) = \mathbb{E}_{x \sim D_O}\left[J_O^{(e)}(\theta, x)\right] \quad (2)$$

This increment, however, may not be equal at each iteration as number of subtasks may not be perfectly distributable across the iterations. Hence, our algorithm tries to optimize subtask increment at later epochs. This is highlighted in Figure 2 where first two epochs have no increment in subtasks but epochs 2 to 4 do, as 6 subtasks cannot be divide into 5 epochs for equal increment across epochs. (Further details about how subtasks are expanded progressively in Appendix B.3). Below, we formally describe this progressive subtask learning strategy.

Orchestrator Training The ReAct-style training instances for the Orchestrator ($x \sim D_{Orch}$) include either (1) task descriptions as inputs paired with their corresponding initial subtasks as outputs, or (2) the Executor agent's latest response as input, with the resulting subtasks and associated plans as the action outputs. Suppose the entire trajectory for the Orchestrator consists of m steps (i.e., ReAct turns) for task x, denoted by $(\{\phi\}, s_1)$, $(\{(s_1, r_1)\}, s_2)$, $\cdots$, $(\{(s_1, r_1), \cdots, (s_{m-1}, r_{m-1})\}, s_m)$. For each turn i, s_i denotes the subtask generated at step i, and the input to this step is the history of subtasks along with Executor's final reports $\{(s_j, r_j)\}_{j<i}$ from previous steps. Here, r_j represents the final report about the success/failure of the subtask s_j. Given $S^{(e)}$ the set of subtasks that the model is allowed to train on at epoch e under the progressive training strategy, we can formally state the training objective for the Orchestrator (see eq 2). While standard fine-tuning trains over the entire trajectory at all epochs, our approach is to only train on progressively increasing parts of this trajectory.

Executor and Critic Training The Executor agent is trained to take as input a subtask and the corresponding plan s_i generated by the Orchestrator and is expected to generate the sequence of turns of code. Additionally, the trajectory for the Executor agent includes environment feedbacks and interactions with the Critic. Suppose the entire trajectory for the overall task x consists of n steps (i.e. ReAct steps) and is given by $(\{\phi\}, (T_1, A_1, O_1))$, $(\{(T_1, A_1, O_1)\}, (T_2, A_2, O_2))$, $\cdots$, $(\{(T_1, A_1, O_1), \cdots, (T_{m-1}, A_{m-1}, O_{m-1})\}, (T_m, A_m, O_m))$. For a given subtask s_i at each step t, the input is the history of the conversation so far: $\{(T_j, A_j, O_j)\}_{j<t}$. Here T, A, and O

denote thoughts, actions, and observations.

In this learning paradigm, we train agents using both correct steps and self-corrected steps. This helps develop a comprehensive understanding of both successful subtask completion and error correction patterns. Following (Fu et al., 2025) we compute the loss function $J_E^{(e)}$ (eq. 4), $J_C^{(e)}$ (eq. 5) only on correct and self-refined steps for Executor and Critic, respectively. In equation 3, N_x is the total number of turns in a given task x.

$$J_{EC}^{(e)}(\theta, x) = \sum_{t=1}^{N_x} \mathbb{I}(A_t).\mathbb{I}(s_t \in S^{(e)}).$$
$$log(\pi_\theta(T_t, A_t | I, subtask, \{T_j, A_j, O_j\}_{j<t})) \quad (3)$$
$$J_E^{(e)}(\theta) = \mathbb{E}_{x \sim D_E}\left[J_{EC}^{(e)}(\theta, x)\right] \quad (4)$$
$$J_C^{(e)}(\theta) = \mathbb{E}_{x \sim D_C}\left[J_{EC}^{(e)}(\theta, x)\right] \quad (5)$$

5 Experimental Setup

We evaluate the performance of Progressive Subtask Training (§5.4), on `AppWorld`, a popular agentic benchmark. We first *data synthesis* to generate training trajectories for `AppWorld` and then train and evaluate models and baselines using `AppWorld` established metrics.

5.1 AppWorld

`AppWorld` offers a testbed of day-to-day digital tasks that test agent abilities to solve complex problems using interactive coding, often over long trajectories. `AppWorld`'s **execution engine** simulates 9 day-to-day apps: Amazon, Spotify, Venmo, Gmail, Todoist, SimpleNote, Splitwise, Phone, and FileSystem. `AppWorld`'s **benchmark** is a dataset of 750 realistic task instructions across 250 different scenarios/use-cases, divided into 105 train, 60 dev, 105 in distribution test and 417 out-of-distribution test sets[3]. `AppWorld` uses **state-based programmatic evaluation** by checking engine's final database state with manually-written assertions (details in Appendix C).

5.2 Data Synthesis

`AppWorld` only provides training pairs (input prompts, output state). Hence, to obtain training trajectories, we design a *data synthesis* pipeline similar to (Erdogan et al., 2025) that first generates `ReAct` (Yao et al., 2023)-style trajectories for single agents and then uses a powerful LLM to translate them to multi-agent trajectories. First, we use `Llama-3.3-70B-Instruct` (Meta, 2025) to generate single-agent trajectories (see the prompt in Figure 13 in Appendix E) from `AppWorld` train and dev sets via rejection sampling at different temperatures (Chen et al., 2025c), obtaining $2,844$ gold trajectories. Then, for each agent (Orchestrator,Executor and Critic) in our multi-agent setup, we prompt `Claude 3.7-Sonnet` (Anthropic, 2025) to convert the single agent trajectories into appropriate trajectories for each agent (See the prompt in Figure 14 in Appendix E). These trajectories were then validated using `AppWorld`. We gathered $2,708$ such trajectories for training (see Appendix A.1 and A.2 for details).

5.3 Models & Setup

We use `Qwen2.5-Coder` variants (`7B`,`14B`, and `32B`) for our experiments due to their strong coding abilities and `Llama-3.1-8B and Phi-4` for their strong reasoning abilities (Hui et al., 2024), skills essential for agents. All models are finetuned using LoRA (Hu et al., 2021) (see Appendix B.1 for more training and evaluation details). We use the same `ReAct` (Yao et al., 2023) prompt from Trivedi et al. (2024) during inference, but we customize it to be agent-specific (see Appendix E). For each variant, we evaluate five different baselines: single-agent (`SA`), and fine-tuned version (`[FT]`), our multi-agent (`MA`) system proposed in Section 3 and its fine-tuned version (`[FT]`) and multi-agent system trained with Progressive Subtask Training (`[ProST]`). We denote agents in multi-agent configurations using the (Orchestrator-Executor-Critic) format, where each placeholder indicates the agent size in billions of parameters (e.g., `7-7-7`). We also evaluate the multi-agent system with different sized Orchestrator models, such as a `14B` Orchestrator with `7B` Executor and Critic (`14-7-7`), and a `7B` Orchestrator with `14B` Executor and Critic (`7-14-14`).

5.4 Cost-Performance Pareto Analysis

Performance `AppWorld` provides two metrics to measure performance: 1) **Task Goal Completion** (`TGC`) measures percentage of tasks where agent passes all the human written unit tests for that task. 2) **Scenario Goal Completion** (`SGC`) is the percentage of scenarios where the agent passes all the unit tests of each task in that scenario. It measures both

[3]We found only 90 train and 57 dev tasks are available in the appworld when we load the datasets.

consistency and performance across similar tasks.

Computational Cost We use Floating Point Operations (FLOPs) to measure computational cost[4]. We approximate FLOPsper instance as a function of the model parameters and tokens involved in the computation. We use the formula from Kaplan et al. (2020) which is $2\times\#\text{params}\times(\text{input tokens}+\text{output tokens})$.

Our evaluations plot these effectiveness metrics against FLOPs to show the Pareto curves (Yue et al., 2025; Pimentel et al., 2020) of different agent setups, thereby providing Pareto-optimal choices at different cost-performance trade-offs.

6 Results

6.1 Effectiveness

Table 1 shows the effectiveness metrics (TGC and SGC) for single and multi-agent setups instantiated with the different size variants of Qwen-2.5-Coder discussed in Section 5.3. We observe two findings:

1) Multi-agent systems are more effective than their single-agent counterparts We expect multi-agent systems with specialized SLMs than their corresponding single agent versions. This is observed in Table 1 and 3 as the multi-agent system (MA) is better than the corresponding single-agent baseline (SA) for each parameter variant (e.g., MA 14-14-14 [ProST] attains TGC score of 42.3, while SA 14 [FT] TGC scores 34.5). Furthermore, we find that multi-agent systems can even be on par with large single agents (e.g., GPT-4o) in terms of both effectiveness as well as efficiency[5]. For example, the Phi-4 14B-14B-14B multi-agent gets 46.4, which is close to GPT-4o gets 48.8 on Test-Normal TGC) (Trivedi et al., 2024).

2) Finetuning with ProST creates more effective multi-agent systems To solve AppWorld tasks, we need models that are good at both complex reasoning as well as coding. Multi-agents powered by code specialist SLMs as base models (e.g., Qwen-2.5-Coder) improve when trained with ProST. However, as shown in Table 1 and 3, when trained with models that are good at both coding and complex reasoning, such as Llama-3.1 and Phi-4 base models, we find that ProST improves

Model	TGC(%)	SGC(%)
Qwen-2.5-Coder-7B		
SA 7	3.6	0
SA 7 [FT]	24.4	8.9
MA 7-7-7	5.9	0
MA 7-7-7 [FT]	25.6	8.9
MA 7-7-7 [ProST]	**30.4**	**19.6**
Qwen-2.5-Coder-14B		
SA 14	14.9	3.6
SA 14 [FT]	34.5	21.4
MA 14-14-14	20.2	8.9
MA 14-14-14 [FT]	35.7	26.8
MA 14-14-14 [ProST]	**42.3**	**26.8**
Qwen-2.5-Coder-32B		
SA 32	36.3	16.1
MA 32-32-32	39.9	19.6

Table 1: TGC and SGC scores on AppWorld Test-Normal benchmark for different baselines and our models. We observe that multi-agent systems trained with ProST gives best performance across each variant, even better than much base models of much larger variants. SA = Single Agent system, MA = Multi-Agent system, FT = Standard finetune, ProST = Finetune with Progressive Subtask Training.

SGC scores on three out of four models, demonstrating its utility for improving robustness as well. Compared to standard fine-tuning, ProST yields 18.8% and 18.5% relative gains in TGC scores for the 7B and 14B coding models, respectively. Using Llama-3.1 and Phi-4, the relative gains are even larger at 30.8% and 34.5%, respectively.

6.2 Pareto Front Comparisons

Figure 3 shows the computational efficiency (measured in FLOPs) against task accuracy (TGC) of a subset of our systems using Qwen-2.5-Coder series models. It shows three *Pareto fronts* for non-finetuned, finetuned, and ProST fine-tuned models. The figure supports three key findings:

1) ProST improves Pareto optimality of multi-agent systems Pareto-front is worst for non-finetuned models. Fine-tuning gives more Pareto-optimal models, observed by Pareto fronts (2) and (3), shifting rightwards, i.e., reduces computational cost while improving effectiveness. However, models trained using ProST have the best tradeoff between accuracy and computational efficiency. This shows the effectiveness of ProST in training more effective models with lower computation costs.

2) Optimality is not dependent on total parameter size alone One would expect a system with a larger number of overall parameters to have a

[4] As we discuss in the limitations section section 8, we only focus on computational cost and not on memory considerations.

[5] Here we mean efficiency in terms of overall FLOPs and not accounting of engineering overheads that can impact both settings.

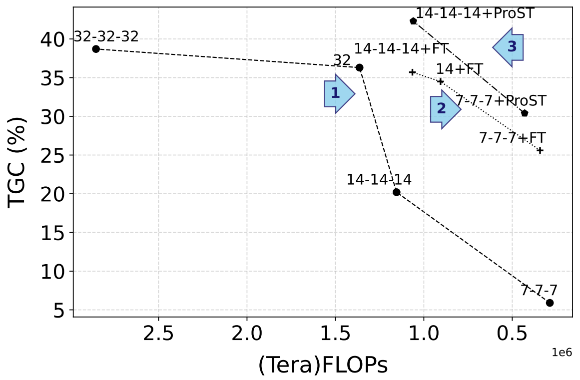

Figure 3: Pareto fronts for three classes of systems: (1) Non-finetuned models, (2) Standard finetuned models, and (3) `ProST`-tuned models.

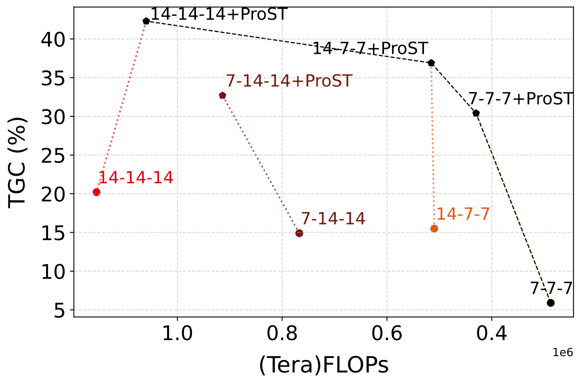

Figure 4: Stronger Orchestrators trained by Progressive Subtask Training achieve better Pareto optimality in terms of computation and accuracy. Points closer to the top-right indicate better trade-offs.

higher computational cost and to perform better in general. However, when comparing the non-fine-tuned 32B single-agent setup to the 14-14-14 ProST multi-agent setup, we observe that the multi-agent system has more parameters (32B vs 42B), yet computational cost is lower (1.36 vs 1.06 TeraFLOPs). This is due to the overall computational cost depending on the overall length of trajectories produced by the different systems, which in turn also depends to some degree on their effectiveness. Notably, the fine-tuned 14-14-14 ProST multi-agent setup achieves a higher score than both the non-fine-tuned 32B single-agent setup and the much larger 32-32-32 multi-agent setup (96B parameters), highlighting that effectiveness is not solely determined by total parameter size.

3) Stronger Orchestrator is a more optimal choice Recent work shows that powerful planning/Orchestrator agents can improve task effectiveness (Erdogan et al., 2025). We study whether allocating higher capacity to the Orchestrator leads to more Pareto-optimal solutions with ProST (see Figure 4.) To this end, we create two system variants: (i) 7-14-14, which uses a weaker Orchestrator (7B), and (ii) 14-7-7, which uses a stronger Orchestrator (14B).

We find that multi-agent systems with a strong Orchestrator are more optimal than ones with a weak Orchestrator of the same parameter count (see Table 5 in Appendix). As shown in the figure, 14-7-7 + ProST has higher TGC and is also more efficient than 7-14-14 + ProST. Also, ProST training reduces the overall FLOPs for all MAS systems with strong Orchestrators (i.e. 14B) while improving effectiveness, when compared to their non-fine-tuned versions. Whereas, ProST training a weak Orchestrator (i.e. 7B) improves effectiveness but does not reduce overall FLOPs. This is likely because poor orchestration (i.e. planning and guidance) causes more trial and error before success, leading to longer trajectories.

Selection Strategies	TGC	SGC
Qwen-2.5-Coder-14B MAS [ProST]		
- All	35.7	26.8
- Random	34.5	23.2
- Decrement	39.9	26.8
- Ours	**42.3**	**26.8**

Table 2: Ablation: Effect of different subtask selection strategies in ProST.

7 Analysis

Importance of progressive subtask selection ProST can be seen as training for a subset of tasks in each epoch using a specific progressive inclusion strategy. Here we test the necessity of the subset selection by comparing against including all subsets of tasks in each epoch (ALL), and the necessity of our specific progressive strategy by comparing against both (i) random subset selection at each epoch (Random)[6], and (ii) an inverse strategy that includes all subtasks (full task) first, and removes subtasks as training progresses(Decrement).

Table 2 shows that random selection yields the worst performance, whereas progressively learning more subtasks either via (Decrement or Ours) provides gains over training on all subtasks from the start. However, Ours, the progressive curriculum

[6]Note that this strategy is constructed to ensure that all subtasks are shown to the model during the training. See Appendix B.2 for details.

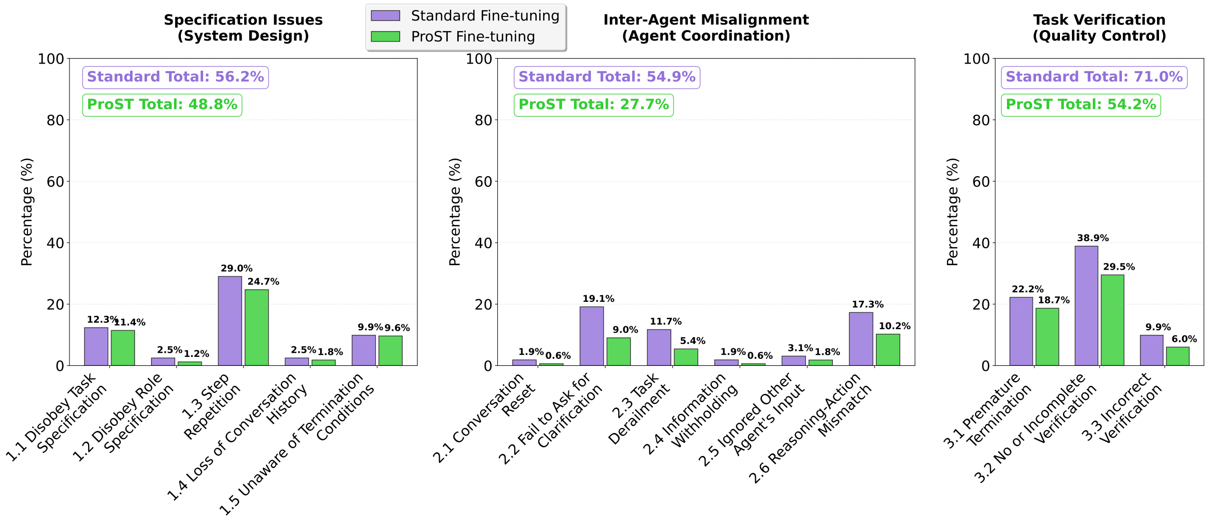

Figure 5: Failure mode analysis using the MAST (Cemri et al., 2025) taxonomy (Specification Issues, Inter-Agent Misalignment, Task Verification) and the LLM-as-a-Judge pipeline (`Llama-3.3 70B` as the judge). Progressive Subtask Training (`ProST`) consistently reduces error modes across all categories.

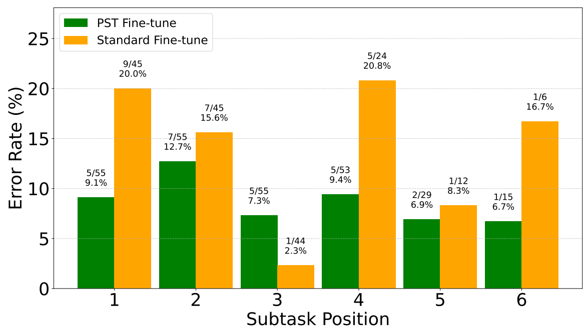

Figure 6: Error rates per subtask position for `ProST` Fine-tuning (green) and Standard Fine-tuning (orange), computed only on successful tasks. The error rate is defined as (number of successful tasks with at least one error at subtask i / total number of successful tasks that reached subtask i) $\times$ 100%. We consider only the subtasks which has more than 5 successful tasks from both settings. `ProST` Fine-tuning shows lower error rates across most subtasks, while Standard Fine-tuning exhibits significantly higher error rates.

whereby we learn the task-specific subtasks first, is still substantially better than its inverse `Decrement`, likely because it better reflects the natural dependencies between the subtasks. However, we see that `SGC` scores are the same for `All`, `Decrement`, and `Ours`, showing that selection has no affect on solving tasks across same scenarios.

`ProST` reduces error rates in subtasks and MAST taxonomy One of the main motivations for `ProST` was our observation that SLMs are unable to learn all subtasks effectively under standard fine-tuning. `ProST` aims to address this by training models to incrementally learn more subtasks. Figure 6 compares the rates of error at each subtask position from successful completion during inference for standard finetuning and `ProST` by counting the number of error messages returned by the `AppWorld` environment. We find that `ProST` has a lower error rate than standard fine-tuning at most subtask positions. This shows that, with `ProST`, SLMs learn subtasks more effectively.

To analyze the errors further, we adopt MAST (Multi-Agent System Failure Taxonomy) (Cemri et al., 2025), an empirically grounded taxonomy for categorizing errors in multi-agent systems, which categorizes failures into *Specification Issues*, *Inter-Agent Misalignment*, and *Task Verification*. We use their validated LLM-as-a-Judge pipeline (with `Llama-3.3 70B` as the judge) to classify errors across these categories. As shown in Figure 5, Progressive Subtask Training (`ProST`) consistently reduces error rates across all categories and subcategories, with highest improvements in inter-agent coordination failures.

How do trajectory lengths compare? We compare the trajectory lengths of single and multi-agent systems. Figure 7 shows the distribution of tokens for the `14B` based systems in successfully completed test set tasks compared to the unsuccessful ones (see Figure 9 in the Appendix for the full results across 7B, 14B, and 32B models) . The trajectory lengths, i.e, the average number of tokens, for unsuccessful tasks is higher than for successful ones for all systems with higher variance. Unsuccessful tasks are likely to have more trial-and-error behavior and more errors, which add to the tra-

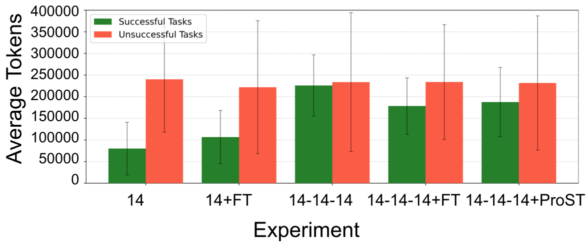

Figure 7: Comparison of total tokens across successful and unsuccessful task completion. Green bars indicate tokens processed in successful tasks, and red bars represent tokens processed in failed tasks. Percentage labels above each bar show the ratio of processed tokens that resulted in completing tasks.

Model	TGC(%)	SGC(%)
`Llama-3.1-8B`		
`SA 8`	6.6	1.8
`SA 8 [FT]`	26.2	8.9
`MA 8-8-8`	5.9	0
`MA 8-8-8 [FT]`	25	7.1
`MA 8-8-8 [ProST]`	**32.7**	**19.6**
`Phi-4-14B`		
`SA 14`	15.5	3.6
`SA 14 [FT]`	32.1	14.3
`MA 14-14-14`	30.9	14.3
`MA 14-14-14 [FT]`	34.5	19.6
`MA 14-14-14 [ProST]`	**46.4**	**28.6**

Table 3: TGC and SGC scores on `AppWorld` `Test-Normal` benchmark for different baselines and models (Llama-3.1-8B and Phi-4-14B). We notice similar performance tradition between Qwen series models and Llama-3.1-8B & Phi-4-14B. We observe that multi-agent systems trained with `ProST` have the best performance in all settings. SA = Single Agent system, MA = Multi-Agent system, FT = Standard finetune, `ProST` = Finetune with Progressive Subtask Training.

jectory length. Multi-agent systems have longer trajectories for successful tasks compared to single-agent systems in part because of the overheads involved in separating the roles and the resultant communication. Fine-tuning, standard, and `ProST` have shorter trajectories for successful tasks, likely due to reduced subtask error rates.

Does `ProST` generalize to models of different capabilities? We also used `ProST` to train two other non-coding models with different generative capabilities, `Llama-3.1-8B` and `Phi-4-14B` and evaluated them on both `Test Normal` (see Table 3) and `Test Challenge` (see Table 4) sets. For `Test Normal`, Llama-3.1-8B and Phi-4-14B yield significant performance improvements. Phi-4-14B performed even better than Qwen-2.5-Coder-14B. Phi-4 `14-14-14 [ProST]` shows 46.4% and 28.6% in TGC and SGC in Test Normal, respectively, and is the highest among the different sized models.

Model	TGC	SGC
`Qwen-2.5-Coder-14B`		
`SA 14`	4.3	2.2
`SA 14 [FT]`	**16.8**	**5.8**
`MA 14-14-14`	10.1	3.6
`MA 14-14-14 [FT]`	13.7	5.0
`MA 14-14-14 [ProST]`	14.1	**5.8**
`Phi-4-14B`		
`SA 14`	6.2	2.2
`SA 14 [FT]`	14.1	3.6
`MA 14-14-14`	12.2	4.3
`MA 14-14-14 [FT]`	12.5	7.2
`MA 14-14-14 [ProST]`	**17.8**	**8.6**

Table 4: Performance comparison in single-agent and multi-agent settings on *Test Challenge* (Test-C) set for `Qwen-2.5-Coder-14B` and `Phi-4-14B`.

Can `ProST` generalize to out-of-distribution data (OOD)? `AppWorld` also provides an out-of-distribution `Test-Challenge` test set, with tasks requiring more API calls and use at least one unseen application. We evaluate `ProST` on this test set, presenting results for `Qwen-2.5 Coder 14B` and `Phi-4 14B` in Table 4. We find that `Phi-4-14B` with `ProST` outperforms other fine-tuning and multi-agent baselines with 17.8% in `TGC` and 8.6% in `SGC`. While `Qwen-2.5-Coder-14B`'s fine-tuned single-agent achieves the best performance in its model family, its `ProST` variant has higher accuracy than the corresponding fine-tuned multi-agent setup. These results show that `ProST`, with appropriate base models, can help improve OOD generalization. More results for this test set are present in Appendix Table 6.

8 Conclusion

SLM based multi-agents present a viable alternative for solving complex problems. However, the limited capacities of SLMs prevent them from learning all subtasks in long trajectory problems. In this work, we introduced a new Progressive Subtask Training algorithm that helps address this challenge. Progressively introducing subtasks during training allows the limited capacity SLMs to more effectively learn the subtasks resulting in improved overall effectiveness. To better understand the cost-effectiveness trade-offs between single and multi-agent setups, we conduct pareto analyses which yield a holistic view of the performance of these systems. Our experiments on `AppWorld`, a popular agentic benchmark demonstrates the utility of `ProST` for addressing complex problems.

Limitations

Generalizability We use AppWorld as a test bed for complex reasoning problems. While our design for the agents i.e., their roles, and ProST training paradigm are broadly applicable to many agentic frameworks, and is not AppWorld specific (detailed explanation in Appendix D.1), some aspects of our modeling and agentic design are influenced by specifics of AppWorld. For example, we showed through ablation studies that the order of introducing subtasks in ProST does indeed matter. However, the choice of introducing the first and last subtasks at the end of training is specific to AppWorld, and it may not generalize well to other benchmarks.

Training Data The training dataset is LLM generated. While we only use successful trajectories, the overall quality of these trajectories (e.g., diversity) has not been evaluated. Also, some parts of the multi-agent training instances are created using a closed-source LLM (Claude-3.7-Sonnet). LLM generation of synthetic data using frontier (non open-source) models, while widely used (Saqib et al., 2025; Wang et al., 2025b; Li et al., 2023; Tang et al., 2023), has its limitations in terms of reproducibility. We will release these instances to ensure replicability and comparisons against other methods for training agents. However, the standard issues of using closed-source LLM will make it difficult to compare against other means of generating training data, unless the method is replicated. We will release code and data to support replicability as much as possible.

Inference-time Computation Cost Progressive Subtask Training optimizes agents in multi-agent architectures such that models perform better at similar inference time cost, measured in FLOPs. However, our analysis does not include memory considerations. This can be a significant bottleneck as we load multiple instantiations of SLMs, leading to a large GPU overhead. For instance, when loading the 14-14-14 multi-agent system, we need one H100 GPU (80GB each) for each agent, totaling around 240GB of memory. While parameter count is taken into account during inference computation cost, we do not account actual cost of loading the whole. We thus carefully scope our claims to only runtime costs as measured by FLOPs.

Out-of-domain Aspects: Since we finetune models on training data with limited scope, models may have learned short-cuts, or overfit to the distribution of the training data. The mixed results on out-of-distribution test data in table 4 and 6 and suggests scope for future work.

Acknowledgements

This work is supported in part by a SUNY-IBM Artificial Intelligence Collaborative Research Alliance grant, an Amazon Research Award 2023, and a NSF NAIRR compute credits grant #240140. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of Amazon.

References

Anthropic. 2025. Claude 3.7 sonnet. Accessed: 2025-04-28.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small language models are the future of agentic ai.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail?

Junzhi Chen, Juhao Liang, and Benyou Wang. 2025a. Smurfs: Multi-agent system using context-efficient DFSDT for tool planning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3281–3298, Albuquerque, New Mexico. Association for Computational Linguistics.

Justin Chen, Archiki Prasad, Swarnadeep Saha, Elias Stengel-Eskin, and Mohit Bansal. 2025b. MAgICore: Multi-agent, iterative, coarse-to-fine refinement for reasoning.

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. 2025c. Reinforcement learning for long-horizon interactive llm agents.

Zhixun Chen, Ming Li, Yuxuan Huang, Yali Du, Meng Fang, and Tianyi Zhou. 2025d. Atlas: Agent tuning via learning critical steps.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks.

Zihao Feng, Xiaoxue Wang, Bowen Wu, Weihong Zhong, Zhen Xu, Hailong Cao, Tiejun Zhao, Ying

Li, and Baoxun Wang. 2025. Empowering llms in task-oriented dialogues: A domain-independent multi-agent framework and fine-tuning strategy.

Dayuan Fu, Keqing He, Yejie Wang, Wentao Hong, Zhuoma Gongque, Weihao Zeng, Wei Wang, Jingang Wang, Xunliang Cai, and Weiran Xu. 2025. Agentrefine: Enhancing agent generalization through refinement tuning.

Alexander Golubev, Sergey Polezhaev, Karina Zainullina, Maria Trofimova, Ibragim Badertdinov, Yury Anapolskiy, Daria Litvintseva, Simon Karasik, Filipp Fisin, Sergey Skvortsov, Maxim Nekrashevich, Anton Shevtsov, Sergey Abramov, and Boris Yangel. 2024. Leveraging training and search for better software engineering agents. *Nebius blog*. Https://nebius.com/blog/posts/training-and-search-for-software-engineering-agents.

Shivanshu Gupta, Sameer Singh, Ashish Sabharwal, Tushar Khot, and Ben Bogin. 2025. Leveraging in-context learning for language model agents. *arXiv preprint arXiv:2506.13109*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

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

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention.

Zhuoyan Li, Hangxiao Zhu, Zhuoran Lu, and Ming Yin. 2023. Synthetic data generation with large language models for text classification: Potential and limitations. *arXiv preprint arXiv:2310.07849*.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma swe-gpt: An open development-process-centric language model for automated software improvement.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Meta. 2025. Llama-3.3-70b-instruct. Also available at: https://huggingface.co/unsloth/Llama-3.3-70B-Instruct. Accessed: 2025-04-20.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with swe-gym.

Tiago Pimentel, Naomi Saphra, Adina Williams, and Ryan Cotterell. 2020. Pareto probing: Trading off accuracy for complexity. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3138–3153, Online. Association for Computational Linguistics.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2024. Adapt: As-needed decomposition and planning with language models.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. 2025. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning.

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Jiang, Chengfei Lv, and Huajun Chen. 2024. AutoAct: Automatic agent learning from scratch for QA via self-planning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3003–3021, Bangkok, Thailand. Association for Computational Linguistics.

Mohammad Saqib, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. 2025. Teaching an old llm secure coding: Localized preference optimization on distilled preferences. *arXiv preprint arXiv:2506.00419*.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. Small llms are weak tool learners: A multi-llm agent.

Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. 2024. Learning to use tools via cooperative and interactive agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10642–10657, Miami, Florida, USA. Association for Computational Linguistics.

Ruixiang Tang, Xiaotian Han, Xiaoqian Jiang, and Xia Hu. 2023. Does synthetic data generation of llms help clinical text mining? *arXiv preprint arXiv:2303.04360*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL*.

Haochun Wang, Sendong Zhao, Jingbo Wang, Zewen Qiang, Bing Qin, and Ting Liu. 2025a. Beyond frameworks: Unpacking collaboration strategies in multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21361–21375, Vienna, Austria. Association for Computational Linguistics.

Yubo Wang, Xiang Yue, and Wenhu Chen. 2025b. Critique fine-tuning: Learning to critique is more effective than learning to imitate. *arXiv preprint arXiv:2501.17703*.

Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. 2025c. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration.

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

Sibo Xiao, Zixin Lin, Wenyang Gao, Hui Chen, and Yue Zhang. 2025. Long context scaling: Divide and conquer via multi-agent question-driven collaboration.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. 2025. MasRouter: Learning to route LLMs for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15549–15572, Vienna, Austria. Association for Computational Linguistics.

Yuting Zeng, Weizhe Huang, Lei Jiang, Tongxuan Liu, XiTai Jin, Chen Tianying Tiana, Jing Li, and Xiaohua Xu. 2025. S^2-MAD: Breaking the token barrier to enhance multi-agent debate efficiency. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9393–9408, Albuquerque, New Mexico. Association for Computational Linguistics.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. 2024a. Cut the crap: An economical communication pipeline for llm-based multi-agent systems.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement.

A Data Preparation and Trajectory Transformation

A.1 Single-Agent Trajectory Generation

AppWorld (Trivedi et al., 2024) provides training pairs consisting of input prompts and output states. Specifically, it offers 90 training and 57 development tasks, each with its corresponding output state (i.e., Python solution). We use all 147 tasks to create a fine-tuning dataset for both single-agent and multi-agent setups. To increase the dataset size and introduce solution diversity for each task, we use Llama-3.3-70B-Instruct (Meta, 2025) as the Re-Act agent across 20 different temperature settings. We start at a temperature of 0.05 and increment by 0.05 each time, up to 1.0. To reflect practical scenarios, we first attempt to solve all $147 \times 20 = 2,940$ tasks without providing the output state to the agent. Our base Llama-3.3-70B-Instruct model successfully solves 1,034 out of 2,940 candidate trajectories across the different temperature settings. For the remaining tasks, we provide abstract instructions instead of direct code. We generate these instructions in natural language using Llama-3.3-70B-Instruct. By feeding these abstract instructions to the agent for the 1,906 unsolved candidates, we obtain 1,396 new gold trajectories. For the final 510 candidates, we convert the Python code solutions into pseudocode using Llama-3.3-70B-Instruct. We avoid giving direct Python code as hints because the agent tends to reproduce the code verbatim. Assisting in pseudocode format yields 414 additional gold trajectories. In total, we collect 2,844 gold trajectories. This approach ensures diverse solutions for each task across different temperature settings. When our agent fails to solve tasks in the first phase (i.e., limit exceeds), we use our curated abstract assistance. We did not generate new training tasks for AppWorld, since creating and validating new tasks requires specific environment support, which is challenging to set up.

A.2 Transformation to Multi-Agent Trajectories

We ensure that each agent in the multi-agent system is trained effectively to handle its specific role while maintaining overall system coherence. To achieve this, we have two options at this point to prepare agent-specific training data. One option is direct distillation. First, we use Llama-3.3-70B-Instruct for this purpose. However, open-source models such as Llama-3.3-70B-Instruct are not trained on such long-horizon tasks for multi-agent setting, leading to fragile plans and overall less informative trajectories. On the otherhand, direct distillation of multi-agent trajectories from frontier models is both prohibitively expensive and has fundamental technical limitations. The costs of direct distillation exceed well beyond typical research budgets—our estimate worked out to between $20 - 25,000$ USD just to obtain the trajectories. To obtain around 3000 trajectories we will have to get around 12000 sample generations from the frontier model. With each generation costing around 2, this works out to $24,000$ dollars.

In contrast, our cost-efficient trajectory transformation approach leverages existing successful single-agent solutions as a foundation, then systematically decomposes them into coordinated multi-agent workflows. We leverage in-context learning capabilities of LLMs by providing the teacher LLM (Claude-3.7-Sonnet) (Anthropic, 2025) with successful single-agent trajectories and prompting it to decompose them into structured multi-agent orchestrations. Given a single-agent trajectory that successfully completes a task in the AppWorld environment, we prompt the LLM to partition the sequence of actions into logical subtasks while preserving the exact execution steps that led to success. Our transformation approach maintains a critical constraint: each executor step (both the reasoning and code) must be identical to the original single-agent trajectory steps. Crucially, we do not allow the model to invent new code or modify existing execution logic - it can only redistribute the original successful steps across subtasks and add exit commands to signal subtask completion. The teacher LLM successfully produces valid 2,708 trajectories from 2844 single agent trajectories. This approach ensures that the generated multi-agent trajectories align with the real execution environment and we further validated these transform actions again by running into the AppWorld environment. This approach proved both economically feasible (cost was only 300 USD) and effective for our purposes.

While the impact of different distillation strategies can add useful knowledge this is somewhat orthogonal to our key contributions of pareto comparisons of single/multiagent systems, improving multiagents using SLMs through ProST training. The idea of using synthetically distilled dataset is a part of our work but not our main focus. We further analyze error rates for each subtask of the multi-agent trajectories (depict in Figure 8). It shows our dataset covers error correction patterns across all subtasks. We finetune specialized agents of different sizes (7B and 14B), as well as single agents as baselines. Agents in the multi-agent systems are trained on their respective datasets in two ways: (1) standard supervised finetuning, and (2) Progressive Subtask Training (`ProST`) (details in Section 4.)

A.3 Pipeline Rationale and Verification

The primary reason we used the specific pipeline of `Llama-3.3-70B` to generate single-agent trajectories and frontier model (i.e., `Claude-3.7-Sonnet`) to translate into multi-agent trajectories is cost. While our approach enables cost-effective scaling, it also introduces variability. These imperfections can affect fairness by biasing agents toward particular solution styles, and they limit reproducibility since other researchers may obtain different trajectories if different LLMs or prompts are used. To mitigate this, we will release all generated trajectories and prompts, enabling others to train under identical conditions. Understanding how the LLMs structure reasoning in natural language is an interesting study that can shed further light on different reasoning patterns, but one that is beyond the scope and goals of our work.

Notably, both the single-agent and the multi-agent trajectories are fully verified. They are gold trajectories in the sense that using them will solve the corresponding tasks. Both versions share a bulk of their trajectories, with the added parts in multi-agent trajectories being mostly content specific to multi-agent aspects of the problem. Our generated trajectories are fully verified — they are actually validated via execution in the AppWorld environment. We only retain trajectories that actually work for the given task. We conducted further sanity checks on a limited set for prompt engineering to ensure that there are no formatting or structural errors and that the steps remain consistent with the intended task goals. There were no errors in the sub-task goals or the description, although we found some cases where the description can be improved in small ways. The reason for the low-levels of noise here is that the original single agent trajectories we use are gold trajectories – actually are verified. Together, this combination of automatic environment-based validation and manual spot-checking shows that our dataset is both high-

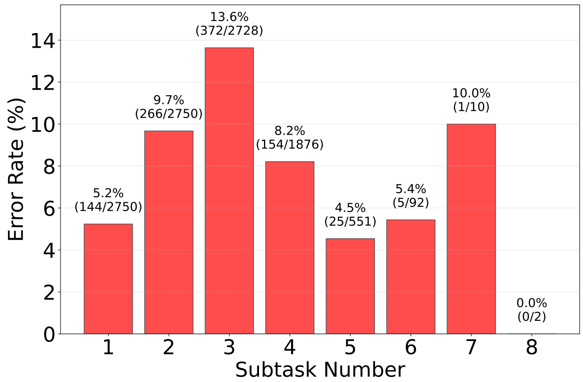

Figure 8: Error rate percentage for each subtask position across all training tasks. It shows what percentage of tasks have at least one error in each subtask position (1st subtask, 2nd subtask, 3rd subtask, etc.). We define the error rate function as (number of tasks that have at least one error in i^{th} subtask position / total number of tasks that have i^{th} subtask position) $\times$ 100%

quality and task-valid.

We use heterogeneous teachers because generating high-quality action traces and plan-level abstractions requires models with different strengths. This approach mirrors Erdogan et al. (2025), which similarly separates plan and act supervision, employs open-source models for execution traces, and frontier models for plan generation and expansion. Both approaches use synthetic data generation, annotating ground-truth trajectories with feasible plans to strengthen the planner – a method shown to improve planner quality in. Fairness is further preserved by applying the same evaluation protocols and training pipeline components where applicable across settings, leaving the agent paradigm (single-agent versus multi-agent) and the multi-agent specific learning method (`ProST`) as the sole intended variables. As reflected by consistent Pareto and `TGC`/`SGC` gains across model scales, our observed effect comes from `ProST` and role specialization, not from differences in teacher models.

B Implementation Details

B.1 Training and Evaluation Configuration

Our base models are finetuned with Low-Rank Adaptation (LORA) (Hu et al., 2021). We use rank (r=16), alpha (α=32) and dropout = 0.05. We use a learning rate of 2.0e-4 with a cosine scheduler. To stabilize training, we set the gradient clipping to 1.0 and the weight decay to 0.01. We train the model for 5 epochs. We use a warm-up ratio of 0.1. The maximum sequence length is set to 20,480 tokens, as our training trajectories are long. All of our experiments are conducted on 4 NVIDIA H100 GPUs with 80GB of memory each. We use Accelerate integrated with DeepSpeed (ZeRO-3) for multi-GPU training. This setup provides efficient memory management and optimization. For both training and evaluation, each agent took 8 GPU hours. For evaluation, we use vLLM (Kwon et al., 2023) servers. For both single-agent and multi-agent setups, we perform inference at T=0.1 and T=0.2 for *Test Normal* set and *Test Challenge*, respectively. We set the maximum sequence length to 65536 and enable prefix caching. For the single-agent setting, we allow up to 50 turns for evaluation. For the multi-agent, Orchestrator can plan up to 12 subtasks, and each time Executor allows max 15 turns (for the *Test Challenge* set, we set max 20 turns) to complete each subtask. A task is considered failed if an agent calls the task completion API with a fail status or doesn't complete the task within the maximum allotted limits.

B.2 Random Subtask Selection Strategy

To create training data for progressively random subtask(s) selection finetuning strategy, we employ a randomized epoch assignment mechanism to ensure comprehensive exposure of all subtasks during model training. For each subtask within a task, we generate two random integers between 0 and 4, then create an inclusive range of epochs from the minimum to the maximum value. This approach guarantees that every subtask appears in at least one epoch (when both random numbers are the same) and can span multiple consecutive epochs (when the numbers differ). This strategy ensures that all subtasks receive adequate exposure throughout the 5-epoch training process.

B.3 `ProST` Details

We always start with at least 2 subtasks. We progressively add task-specific subtasks, primarily during epochs 0–2, and include the first subtask (login in `AppWorld`) and task completion subtasks in epochs 3–4 if the total number of subtasks is at most 5. If there are 6 or more subtasks, we place greater emphasis on task-specific subtasks in epochs 0–3 and add non-task-specific subtasks in epoch 4. In general, we do not strictly assign subtasks to specific epochs; instead, our approach adds subtasks when there are enough available subtasks to add at each epoch. Otherwise, we add subtasks after every k epochs (e.g., $k = 2, 3$). For each task, we make sure that all subtasks are seen during the full 5-epoch training.

C Appworld Details

`AppWorld` (Trivedi et al., 2024) is a framework designed to assess the ability of autonomous agents in solving real-world problems by interacting with an environment of everyday applications. The framework provides the user with two components: the **`AppWorld` Execution Engine** and **`AppWorld` Benchmark**.

`AppWorld` Execution Engine provides an execution environment where agents can interact with abstract versions of the following nine commonly used applications: Amazon, Spotify, Venmo, Gmail, Todoist, SimpleNote, Splitwise, Phone, and FileSystem. These applications are simulated using 457 different APIs and 101 database tables with $370K$ rows that represent around 100 users. The execution engine is designed to create a safe and well-tested (98% coverage with $1,780$ unit tests) environment for testing agents' ability to interact with applications by writing code to invoke APIs.

`AppWorld` Benchmark provides a dataset consisting of 750 realistic task instructions across 250 different scenarios of app usage in the `AppWorld` engine. The tasks are divided into 105 train-set 60 validation set[7], 168 in-distribution *Test Normal* test set, and 417 out-of-distribution *Test Challenge* testbed with unseen apps and APIs. The tasks are created to utilize, on average, 1.8 apps or make 9.8 API calls by writing ~ 50 lines of code. The task is evaluated using **State-Based Programmatic Evaluation**, where the final state of the database is checked against several manually written assertions that must pass if the task was completed correctly.

D Extended results

D.1 Generalizability of `ProST`

The core components of our approach—role specialization, progressive subtask exposure, and Pareto-aware optimization—are not tied to AppWorld's specific applications. The Orchestrator-Executor-Critic architecture is a generalizable decomposition pattern. Our training strategy, `ProST`, targets a universal challenge: SLMs' inability to learn long trajectories under standard fine-tuning. Our ablation studies confirm that the order of subtask introduction matters, suggesting that curricu-

[7]We found only 90 train and 57 dev tasks are available in the appworld when we load the datasets.

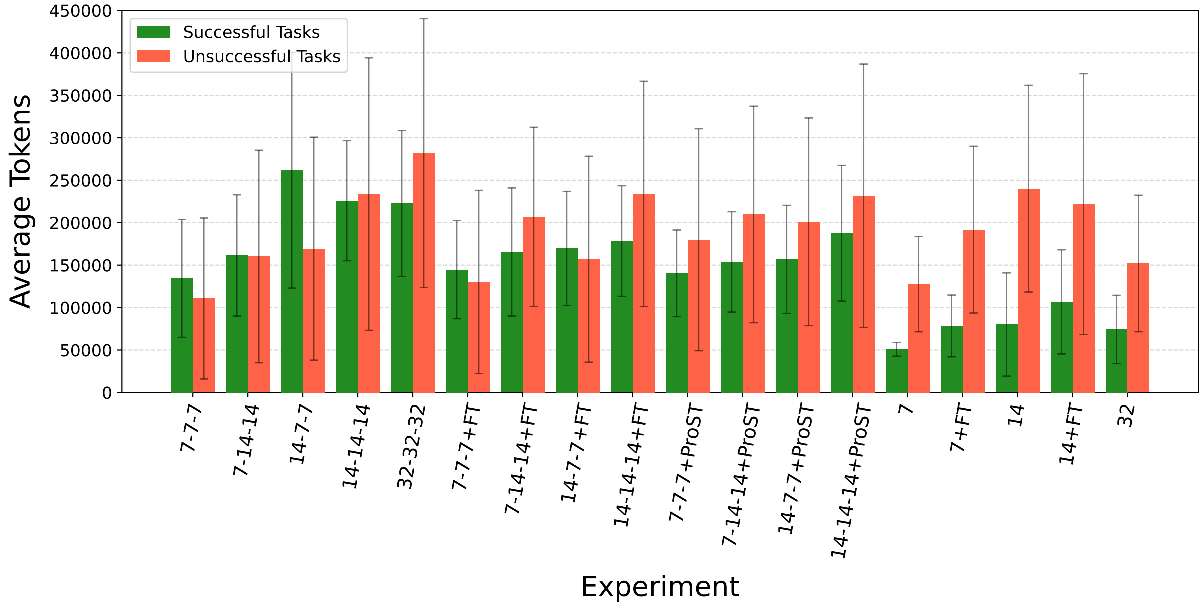

Figure 9: Comparison of average tokens across successful and unsuccessful tasks for all experiments. Green bars indicate average tokens spent on successful tasks, and red bars represent average tokens spend on each failed task.

Model	TGC(%)	SGC(%)
Qwen-2.5-Coder		
MA 14-7-7	15.5	3.6
MA 7-14-14	14.9	3.6
MA 14-7-7 [FT]	30.4	17.9
MA 7-14-14 [FT]	26.8	12.5
MA 14-7-7 [ProST]	36.9	26.8
MA 7-14-14 [ProST]	32.7	16.1

Table 5: Extended results table with different model sizes multi agent system. TGC and SGC scores on AppWorld Test-Normal benchmark for different baselines and our models. MA = Multi-Agent system, FT = Standard finetune, ProST = Finetune with Progressive Subtask Training.

lum design is a critical lever—not just an artifact of AppWorld's tasks. Furthermore, we successfully apply ProST to non-coding models (Llama-3.1-8B, Phi-4-14B), showing consistent gains across architectures, and observe performance improvements on out-of-distribution tasks, where even modest gains indicate robustness. Using a single, high-quality benchmark for comprehensive evaluation is standard practice for complex problems due to a range of reasons. Firstly, working on complex task domains often requires significant engineering effort and environment setup cost. Many prominent works in software engineering and agent evaluation use single long horizon domains for this reason. For instance, the widely-cited SWE-bench (Jimenez et al., 2024) paper has become the gold

Model	TGC(%)	SGC(%)
Qwen-2.5-Coder-7B		
SA 7	0.2	0.0
SA 7 [FT]	8.9	2.9
MA 7-7-7	1.7	0.0
MA 7-7-7 [FT]	7.2	3.6
MA 7-7-7 [ProST]	**10.5**	**4.3**
Qwen-2.5-Coder-14B		
SA 14	4.3	2.2
SA 14 [FT]	**16.8**	**5.8**
MA 14-14-14	10.1	3.6
MA 14-14-14 [FT]	13.7	5.0
MA 14-14-14 [ProST]	14.1	**5.8**
Llama-3.1-8B		
SA 8	2.2	0.0
SA 8 [FT]	8.9	2.2
MA 8-8-8	2.2	0.0
MA 8-8-8 [FT]	8.9	**4.3**
MA 8-8-8 [ProST]	**10.8**	3.6
Phi-4-14B		
SA 14	6.2	2.2
SA 14 [FT]	14.1	3.6
MA 14-14-14	12.2	4.3
MA 14-14-14 [FT]	12.5	7.2
MA 14-14-14 [ProST]	**17.8**	**8.6**

Table 6: Performance comparison in single-agent and multi-agent settings on *Test Challenge* (Test-C) set for Qwen-2.5-Coder-7B, Qwen-2.5-Coder-14B, Llama-3.1-8B, and Phi-4-14B.

standard for evaluating coding agents and notable works (Yang et al., 2024; Zhang et al., 2024b; Ma et al., 2024; Pan et al., 2025; Golubev et al., 2024; Xia et al., 2024; Xie et al., 2025) attempt to resolve programming tasks by interactive coding agents.

Making progress on AppWorld represents useful advances in the complex problem space. The complex tasks in AppWorld are defined over everyday apps in the AppWorld benchmark, which is comparable to SWE-bench in scope and difficulty (see AppWorld details in Appendix C). This is a challenging benchmark and there are other recent works that focus exclusively on improving performance on this benchmark as well (Chen et al., 2025c; Gupta et al., 2025). A recent multi-agent analysis paper also showed that AppWorld tasks pose substantial reasoning and coding challenges (Cemri et al., 2025). Another recent work (Qi et al., 2025) evaluate solely their method on a long-horizon Web benchmark WebArena due to the challenges of complexity of configuring a second long horizon benchmark. Another limiting factor in exploring other complex tasks benchmarks is scacity of training data for multi agent systems. We spent a significant amount of time creating multi step reasoning training data (i.e for AppWorld) following rigorous processes. Thus, we argue that applying any other long-horizon benchmark's tasks to the `ProST` training paradigm would also yield significant performance improvements compared to standard fine-tuning.

E Prompts Details

When creating subtasks, follow these guidelines:

- Provide a clear description of the subtask, including the goal. Add every small detail from the main actual task.
- List the steps in natural language format to achieve the subtask's goal.
- Specify the apps that can be used to achieve the subtask's goal. **Do not include any API names**, as the orchestrator does not have knowledge of API names.
- Instruct the Executor agent to find relevant APIs from the API list of the specified app.
- Suggest checking the detailed documentation of the relevant API using `apis.api_docs.show_api_doc(app_name, api_name)` to understand its arguments and output structure.
- Suggest calling the API using `apis.app_name.api_name` with the required parameters to get the desired information or perform the action. Mention detailed substeps (how to implement the logic) in natural language to achieve the subtask's goal.
- If the API requires authentication, suggest using the app's access token obtained from the previous authentication step.

Figure 10: Prompt for the Orchestrator agent. Here we show only the extended part. We customized the original Re-Act prompt shown in Figure 13.

Key instructions:

- For each step you take to solve the subtask, first write your thought process, then include code inside a `<code> ... </code>` block.
- Once you have completed the given subtask, include a summary of what you accomplished in your response.
  - Mention all the variables' exact names you used to store information and describe what they contain.
  - Must end with the exit command in a `<code>exit</code>` block to signal completion. If you cannot solve the subtask, include a message explaining why and add a `<code>exit</code>` block. Do not use any other method to signal completion of the subtask.

Figure 11: Prompt for the Executor agent. Here we show only the extended part. We customized the original Re-Act prompt shown in Figure 13.

Key instructions:

- **REVIEW SCOPE:** Analyze the most recent step's code and reasoning and previous steps in the conversation. Look for:
  - Logical flaws that would cause incorrect results with respect to the subtask.
  - Whether the code snippet uses any undefined variables. If the variable is defined in previous steps, then we can use it.
  - Whether the APIs are using correct parameter(s) according to the API documentation.
  - Whether the code uses valid email addresses, access tokens, and variables from the actual conversation context, not placeholders.
- When reviewing code, check if variables from previous steps are being used correctly. Variables defined in earlier code blocks should be available in subsequent blocks.
- Ensure the code properly utilizes the `"supervisor"` app for account information and the `"phone"` app for contacts/friends/family data when needed.
- Check that the code examines API specifications using `apis.api_docs.show_api_doc` before making API calls, and verify the API calls match the documented parameters exactly.
- Do not suggest code enhancements or optimizations. Focus solely on correctness and adherence to the subtask requirements.
- For APIs that return paginated results, verify the code properly loops through all pages using `page_index` or a similar mechanism.
- Your role is to review code and provide feedback, not to write code. Focus on identifying issues as defined in **REVIEW SCOPE** and provide brief actionable feedback in natural language.

Figure 12: Prompt for the Critic agent. Here we show only the extended part. We customized the original Re-Act prompt shown in Figure 13.

USER:
I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously. To do this, you will need to interact with app/s (e.g., spotify, venmo, etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

To explore available APIs and functionality, following are the key commands:

To get a list of available apps
print(apis.api_docs.show_app_descriptions())

To get the the list of available APIs in a specific app, e.g. supervisor
print(apis.api_docs.show_api_descriptions(app_name=‘supervisor’))

To get the specification of a particular api, e.g. supervisor app's show_account_passwords
print(apis.api_docs.show_api_doc(app_name='supervisor', api_name='show_account_passwords'))

To call a particular API from an app
print(apis.app_name.api_name(args))

To call the task completion API
print(apis.supervisor.complete_task(answer=<answer>))

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task. Let's start with the task

[Re-Act style example trajectory placeholder]

USER:
Congratulations, we solved the example task successfully. Now, before going to the next task, I want you to know the key instructions about solving the subtasks of the next task.

Key instructions:

- Remember that the email addresses, access tokens and variables (e.g. spotify_password, spotify_access_token) in the example above are not valid anymore.
- For each step you take to solve the subtask, first write your thought process, then include code inside a <code> ... </code> block.
- Only generate valid code blocks, i.e., do not put them in “‘...“‘ or add any extra formatting.
- Remember you can use the variables in your code in subsequent code blocks.
- Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
- The provided Python environment has access to its standard library. But modules and functions that have a risk of affecting the underlying OS, file system or process are disabled. You will get an error if do call them.
- Any reference to a file system in the task instructions means the file system *app*, operable via given APIs, and not the actual file system the code is running on. So do not write code making calls to os-level modules and functions.
- To interact with apps, only use the provided APIs, and not the corresponding Python packages. E.g., do NOT use ‘spotipy‘ for Spotify. Remember, the environment only has the standard library.
- The provided API documentation has both the input arguments and the output JSON schemas. All calls to APIs and parsing its outputs must be as per this documentation.
- Many APIs return items in "pages". Make sure to run through all the pages by looping over ‘page_index‘. For that use while true loop and check if the API returns any items. If it does, process them and increment the ‘page_index‘ by 1. If it does not return any items, break the loop.
- If no direct API exists for the information you are looking for, examine all available APIs (relevent APIs first).
- Maintain variable names that are meaningful and relevant to the context of the task. Avoid using generic names like ‘result‘, ‘data‘, or ‘temp‘. Instead, use descriptive names that reflect the content or purpose of the variable, such as ‘song_list‘, ‘spotify_login_result‘, ‘song_details‘, ‘gmail_password‘, etc.
- To obtain current data or time, use Python functions like ‘datetime.now()‘ or obtain it from the phone app. Do not rely on your existing knowledge of what the current date or time is.
- For all temporal requests, use proper time boundaries, e.g., if I ask for something that happened yesterday, make sure to consider the time between 00:00:00 and 23:59:59. All requests are concerning a single, default (no) time zone.
- Any reference to my friends, family or any other person or relation refers to the people in my phone's contacts list.
- All my personal information, and information about my app account credentials, physical addresses and owned payment cards are stored in the "supervisor" app. You can access them via the APIs provided by the supervisor app.
- Once you have completed the task, call ‘apis.supervisor.complete_task()‘. If the task asks for some information, return it as the answer argument, i.e. call ‘apis.supervisor.complete_task(answer=<answer>)‘. For tasks that do not require an answer, just skip the answer argument or pass it as None.
- The answers, when given, should be just entity or number, not full sentences, e.g., ‘answer=10‘ for "How many songs are in the Spotify queue?". When an answer is a number, it should be in numbers, not in words, e.g., "10" and not "ten".
- You can also pass ‘status="fail"‘ in the complete_task API if you are sure you cannot solve it and want to exit.
- You must make all decisions completely autonomously and not ask for any clarifications or confirmations from me or anyone else.

USER:
Using these APIs, now generate code to solve the actual task:

Supervisor name is: <first_name> <last_name>. Email is <email> and phone number is <phone_number>.

Task: <task_instruction>

Figure 13: Prompt for single Re-Act agent inference

I am your supervisor and you are a highly intelligent AI Assistant. Your task is to transform a single-agent trajectory into a multi-agent trajectory for tasks in the AppWorld environment. The single-agent trajectory sucessfully resolved the task, so you need to ensure that the multi-agent trajectory also resolves the task successfully if I run the steps by order in the environment.

AppWorld Environment Overview
AppWorld is a simulated environment with 9 day-to-day apps that mimic real-world applications. This environment provides these following apps through the 'apis' object:

- amazon: Shopping and order management
- spotify: Music streaming and playlist management
- gmail: Email communication and management
- todoist: Task and to-do list management
- simple_note: Note-taking and organization
- venmo: Person-to-person payments
- splitwise: Expense tracking and settlement
- file_system: File operations and management
- phone: Calling and messaging functionality

Additionally, there are 2 helper applications:

- api_docs: Provides interactive documentation lookup for all apps
- supervisor: Provides access to personal information (addresses, payment cards, account passwords)

Key API Commands

- 'apis.api_docs.show_app_descriptions()': List available apps
- 'apis.api_docs.show_api_descriptions(app_name=<app_name>)': List APIs for an app
- 'apis.api_docs.show_api_doc(app_name=<app_name>, api_name=<api_name>)': Get API details
- 'apis.app_name.api_name(args)': Call an API
- 'apis.supervisor.complete_task(answer=<answer>)': Complete task

Multi-Agent Framework
Your transformation will involve two agents:

1. **Orchestrator Agent**:

- Reviews the task and determines the next logical subtask
- Provides detailed subtask descriptions to the Executor
- Receives completion reports from the Executor

2. **Executor Agent**:

- Performs the subtasks defined by the Orchestrator
- Not aware of the overall task, only focused on the current subtask
- Works in a Python REPL environment executing code step-by-step to accomplish the subtask
- Reports back to the Orchestrator upon subtask completion

Transformation Guidelines
Subtask Design

- Create meaningful, logical subtasks that progress toward the overall goal
- Each subtask should be a discrete step toward completing the task
- Supervisor is the user, so use 'user' when talking about the who is using the app, for example "Find the song user liked" rather than "Find the song 'I', or 'you' liked"
- Authentication/login to an app should always be a separate subtask
- Must ensure the final subtask involves calling the task completion API. Orchestrator agent should instruct to call 'apis.supervisor.complete_task()'. If the task requires information, should ask to return it using the answer parameter: 'apis.supervisor.complete_task(answer=<answer>)'.

Subtask Description Format

- Begin with a brief description of the subtask's goal
- List all possible logical steps in details by order to accomplish the subtask. The order of the steps should be the same as the order of the steps in the original single-agent trajectory
- For authentication related subtasks, include these specific steps:
  - Add supervisor name, email, and phone number to the description as this info will be helpful for the Executor agent for authentication
  - Suggest to explore the API documentation of the app using 'apis.api_docs.show_api_descriptions (app_name=<app_name>)' to find the authentication-related API
  - Suggest to check the detailed documentation of the authentication API using 'apis.api_docs.show_api_doc (app_name=<app_name>, api_name=<api_name>)' to understand its arguments and output structure.
  - For username, suggest to use username (e.g., email address, phone number, etc.) given in the subtask description
  - For password, suggest to find the account's password retrieval API from the "supervisor" app and call the API to retrieve the password
  - Suggest to call the login API using 'apis.<app_name>.<login_api> (username=<username>, password=<password>)' with the collected username and password. And store the token for future use
- For other subtasks,
  - List the steps in natural language format
  - Specify the possible apps that can be used to achieve the subtask's goal. Don't include any API names as orchestrator don't have knowledge of the API names
  - **Don't include actual API names in the description, instead suggest to explore the API documentation using 'apis.api_docs.show_api_descriptions (app_name=<app_name>)' to find the relevant API. Then suggest to findout relevent APIs**
  - Suggest to check the detailed documentation of the relevent API using 'apis.api_docs.show_api_doc (app_name=<app_name>, api_name=<api_name>)' to understand its arguments and output structure.
- In the subtask description, include actual path, file or people names, if mentioned in the task description. For example, mention actual path instead of saying "...the specified path..."
- For final subtask, **must include a note to call the task completion API using 'apis.supervisor.complete_task(answer=<answer>)' to signal that the overall task has been completed. If the task requires information, should ask to return it using the answer parameter: 'apis.supervisor.complete_task(answer=<answer>)'**
- End with instruction to report back upon completion

Executor Steps Format

- Each step (thought and code) must be identical to the original step in the single-agent trajectory
- Must include both the thought and the code exactly as in the original step
- Must enclosed on code in <code> </code> tags
- Upon subtask completion, add a summary and exit command: <code>exit</code>. Summary should include what executor agent has accomplished in the subtask and signal to the Orchestrator agent that the subtask is complete.

Output Requirements

- IMPORTANT: You must respond ONLY with valid JSON. Do not include any explanatory text, introductions, or markdown formatting outside of the JSON object. Your entire response must be parseable as JSON.
- Follow the exact JSON structure shown in the below example. The example provided below is for illustrative purposes only. In the given example- the App, and API names were only for demonstration.
- Do not exclude any original step or code
- Do not invent new code except for the final exit command
- Preserve all original steps; it's thought as well as the code
- Divide into logical subtasks with authentication always as a separate subtask
- Define thought first and then format only code within <code> tags and end each subtask with <code>exit</code>
- Do not include actual API names in the description, instead suggest to explore the API documentation using 'apis.api_docs.show_api_descriptions(app_name=<app_name>)' to find the relevant API. Then suggest to findout relevent APIs. Then check the detailed documentation of the relevent API using 'apis.api_docs.show_api_doc(app_name=<app_name>, api_name=<api_name>)' to understand its arguments and output structure.
- Use the placeholder '<app_name>' and '<api_name>' in the subtask description, because the orchestrator agent doesn't know the API names
- In final subtask, include a note to call the task completion API using 'apis.supervisor.complete_task(answer=<answer>)' to signal that the overall task has been completed. If the task requires information, should ask to return it using the answer parameter: 'apis.supervisor.complete_task(answer=<answer>)'

Your goal is to transform a single-agent trajectory into a multi-agent trajectory following the JSON schema below:

```
{
  "subtasks": [
    {
      "subtask_number": <integer>,
      "subtask_description": <string>,
      "executor_steps": [
        {
          "subtask_number": <integer>,
          "step_number": <integer>,
          "plan_and_code": <string and <code> code </code> >
        },
      ]
    },
  ]
}
```

For example, consider the following task and the multi-agent trajectory solution where two agents are solving the task.

[Example Multi-Agent Trajectory Placeholder]

Now translate the actual single-agent trajectory into a multi-agent trajectory:

Actual task: <task_description>
Supervisor name is: <first_name> <last_name>. Email is <email> and phone number is <phone_number>.

The single-agent trajectory is as follows: <single_agent_trajectory>

Figure 14: Prompt for Single-agent to multi-agent trajectory transformation