# LITEEXEC: SCALING DOWN THE EXECUTOR IN LONG-HORIZON AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Long-horizon tasks involve different capability requirements, from high-level planning and task decomposition to direct execution and tool interaction. Monolithic agents apply a single model throughout, potentially misallocating model capacity and increasing inference costs. In this work, we investigate whether these differences can be exploited by pairing a large-model planner with a smaller executor. With this role separation, we find that scaling down model size for the executor under a large-model planner leads to a smaller performance drop than scaling down a monolithic agent, suggesting that execution is more amenable to scaling down. Additionally, the planner consumes less than $4\%$ of total tokens, leaving most token usage to smaller executors. However, off-the-shelf executors remain limited in performance and token efficiency due to ineffective collaboration with the planner. To address these limitations, we propose Executor GRPO, a simple yet effective reinforcement learning approach that trains a small executor using end-to-end task rewards with a frozen planner. The resulting model, LiteExec, exhibits emergent collaborative behaviors during training, including more informative feedback and more frequent reporting of execution difficulties. Experimental results demonstrate that pairing large-model planners with LiteExec improves performance and substantially reduces estimated inference costs relative to off-the-shelf executors, with improvements generalizing across planners and from coding to search tasks. Compared with the corresponding monolithic large-model agents, large-model planners paired with LiteExec-9B achieve competitive scores of $70.0$ (vs. $73.0$) on SWE-bench Verified and $76.0$ (vs. $78.7$) on SWE-bench Multilingual, while reducing large-model token usage by approximately $88\%$ and $90\%$ and estimated total inference costs by $24\%$ and $71\%$, respectively.

## 1 INTRODUCTION

Large language model (LLM) agents autonomously interleave reasoning, tool-mediated actions, and feedback from external environments over multiple steps (Yao et al., 2023; Shinn et al., 2023). Recent advances in LLM agents have led to substantial progress on long-horizon tasks across diverse domains, including software engineering (Yang et al., 2024; Deng et al., 2025), multi-source information seeking (Wong et al., 2026; Gupta et al., 2026), and automated AI research workflows (Lu et al., 2026; Starace et al., 2025). These advances have been driven by continued growth in both model scale and interaction horizons, with recent systems reaching the trillion-parameter regime and enabling million-token contexts (DeepSeek-AI et al., 2026; Kimi Team, 2026), resulting in higher inference costs for long-horizon tasks (Chen et al., 2024; Ong et al., 2025).

A primary contributing factor to high inference costs is the mainstream monolithic agent design, which applies a single model throughout the trajectory while overlooking the heterogeneous capability requirements across steps in long-horizon tasks. In particular, planning steps involve task decomposition and long-range coordination, requiring higher-capability models, whereas execution focuses on more localized tool interactions, such as web search and sandbox use, which may be handled by smaller models (Wu et al., 2026; Liu, 2026; Lee et al., 2026). This asymmetry raises our central question: whether pairing a large planner with a smaller executor can reduce inference costs on long-horizon tasks while maintaining competitive performance?



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To address this question, we first compare agents pairing a large-model planner with smaller executors against monolithic agents across different model sizes on long-horizon tasks (Section 2). We find that planner guidance improves the performance of the 9B and 35B-A3B models over their monolithic counterparts (Figure 1(a)). Moreover, scaling down only the executor incurs a smaller performance drop than scaling down the monolithic agent, suggesting that execution is more amenable to smaller models under planner guidance. Meanwhile, the planner accounts for only $1.9\%$–$3.6\%$ of total token usage, allowing smaller models to handle most of the token-intensive execution (Figure 1(b)). However, directly pairing off-the-shelf executors with a large-model planner still yields lower performance and higher estimated inference costs than the monolithic large-model agent (Figure 1(a),(c)). The cost gap reflects substantially higher executor token usage, which can outweigh the savings from lower per-token prices (Chen et al., 2026a). Effective collaboration requires feedback on execution evidence and unresolved difficulties; incomplete reports and unproductive attempts can impair coordination and increase token usage. These limitations motivate targeted executor adaptation for more effective and efficient collaboration with the planner.

To this end, we introduce Executor GRPO (Section 3), a simple and effective reinforcement learning (RL) approach that adapts a small model to the executor role using end-to-end task rewards under a fixed large-model planner. As illustrated in Figure 2, each complete plan–execution rollout receives a group-relative advantage shared across all executor rounds within that rollout, and only the executor policy is updated. During RL training, LiteExec exhibits emergent collaborative behaviors, providing more informative execution reports and more readily communicating difficulties to the planner, despite receiving no explicit rewards for these behaviors.

Experiments demonstrate that LiteExec improves task performance while substantially reducing estimated inference costs (Table 1). Pairing a large-model planner with LiteExec-9B instead of the off-the-shelf executor improves scores from $65.6$ to $70.0$ on SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) and from $61.3$ to $67.3$ on SWE-bench Multilingual (Yang et al., 2025), while reducing estimated costs by approximately $69\%$ and $63\%$, respectively. Compared with monolithic large-model agents, large-model planners paired with LiteExec-9B achieve competitive performance with substantially lower large-model token usage and estimated total inference costs. These plan–execution agents achieve $70.0$ on SWE-bench Verified and $76.0$ on SWE-bench Multilingual, compared with $73.0$ and $78.7$ for the corresponding monolithic agents, while reducing large-model token usage by approximately $88\%$ and $90\%$ and estimated total inference costs by $24\%$ and $71\%$, respectively. Despite being trained exclusively on coding tasks, LiteExec generalizes to search, outperforming the off-the-shelf executor on both BrowseComp (Wei et al., 2025) and DeepSearchQA (Gupta et al., 2026) at lower estimated costs (Table 2). Further analysis reveals more active planner–executor collaboration, characterized by more informative feedback and more frequent reporting of execution difficulties (Section 4.4). Our contributions are threefold:

- We study pairing large-model planners with smaller executors to reduce inference costs on long-horizon tasks, showing the potential of executor scaling and the limits of off-the-shelf executors.
- We introduce Executor GRPO, a simple and effective RL approach that trains a small executor under a fixed planner, with collaborative behaviors developing during training.
- We demonstrate that large-model planners paired with LiteExec outperform those using off-the-shelf executors and approach monolithic large-model performance at lower costs.

## 2 Empirical Study of Scaling Down the Executor

### 2.1 Problem Setting

Long-horizon tasks interleave *high-intelligence* steps, including long-term planning and task decomposition, with *low-intelligence* steps centered on direct execution, such as web search and sandbox interaction. Conventional monolithic agents apply a single model throughout the trajectory, ignoring this asymmetry and potentially misallocating model capacity. In this paper, we investigate whether these roles can be separated, with a larger model handling planning and task decomposition and a smaller model handling execution, to solve long-horizon tasks more efficiently.

**Agent harness.** To examine this question, we compare two agents: a monolithic agent and a plan–execution agent. The monolithic agent solves each task in a single model context with direct access

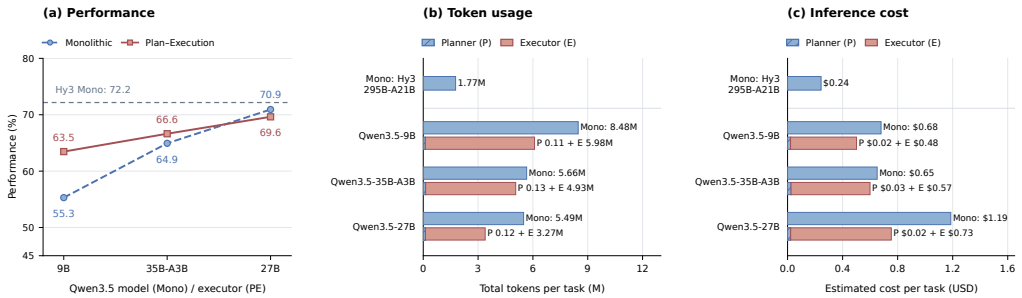


Figure 1: **Performance and inference efficiency under executor scaling.** Qwen3.5 models are evaluated as monolithic agents and as executors paired with Hy3-295B-A21B. Results are averaged over SWE-bench Verified and SWE-bench Multilingual. (a) Task performance. (b) Total input and output tokens per task. (c) Estimated inference cost per task.

to environment tools. The plan–execution agent (Figure 2(a)) comprises a tool-restricted planner and an executor that collaborate over multiple rounds to solve each task. At each round, the planner issues a subtask. The executor interacts with the environment and returns a report; accordingly, the planner either delegates another subtask or produces the final answer. The planner and executor prompts are provided in Appendix A.

**Evaluation.** To test whether separating planning from execution enables more efficient long-horizon agents, we study a large-model planner guiding a smaller executor. We use Hy3, a 295B MoE model with strong reasoning and agentic capabilities, as the planner (Tencent Hy Team, 2026). As executors, we evaluate Qwen3.5 models at three scales: 9B, 35B-A3B, and 27B (Qwen Team, 2026). We also evaluate all four models as monolithic agents. We report average performance across two software engineering benchmarks: SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) and SWE-bench Multilingual (Yang et al., 2025). More details are provided in Section 4.1.

### 2.2 Results and Analysis

**Execution Is More Amenable to Scaling Down.** Figure 1(a) compares monolithic and plan–execution performance across model sizes, with Hy3 as the planner. Relative to their monolithic counterparts, the 9B and 35B-A3B models paired with Hy3 improve from $55.3\%$ to $63.5\%$ and from $64.9\%$ to $66.6\%$, respectively, while reducing estimated inference costs (Figure 1(c)). Across the tested range, plan–execution performance is also less sensitive to scaling down model size. Reducing model size from Qwen3.5-27B to Qwen3.5-9B decreases monolithic performance by $15.6$ percentage points ($70.9\%$ to $55.3\%$), compared with $6.1$ percentage points ($69.6\%$ to $63.5\%$) when scaling down the executor under a large-model planner. These results suggest that retaining a large planner can mitigate the performance loss from using a smaller executor. Meanwhile, the planner accounts for only $1.9\%$–$3.6\%$ of total token usage, leaving $96.4\%$–$98.1\%$ to the smaller executor (Figure 1(b)). These findings suggest that pairing a large planner with a smaller executor is promising for efficient long-horizon tasks and motivate further executor adaptation.

**Limitations of Off-the-Shelf Executors.** Despite these benefits, performance and cost gaps remain relative to the monolithic large-model agent. All three plan–execution configurations score below monolithic Hy3 ($72.2$), including $63.5$ with the 9B executor. Their estimated per-task costs of $\$0.50$–$\$0.75$ exceed $\$0.24$ for monolithic Hy3 (Figure 1(c)). This cost gap reflects the dominant token consumption of the executor: while the planner uses few tokens, the executor consumes substantially more tokens than the stronger monolithic model. Our empirical analysis identifies ineffective planner–executor interaction as a key bottleneck. In particular, executors tend to solve tasks independently, even when facing difficulties, rather than adopt a collaborative approach, such as reporting key findings or blocking issues to the planner, leading to unproductive token consumption and lower task performance. We hypothesize that this behavior stems from mainstream monolithic

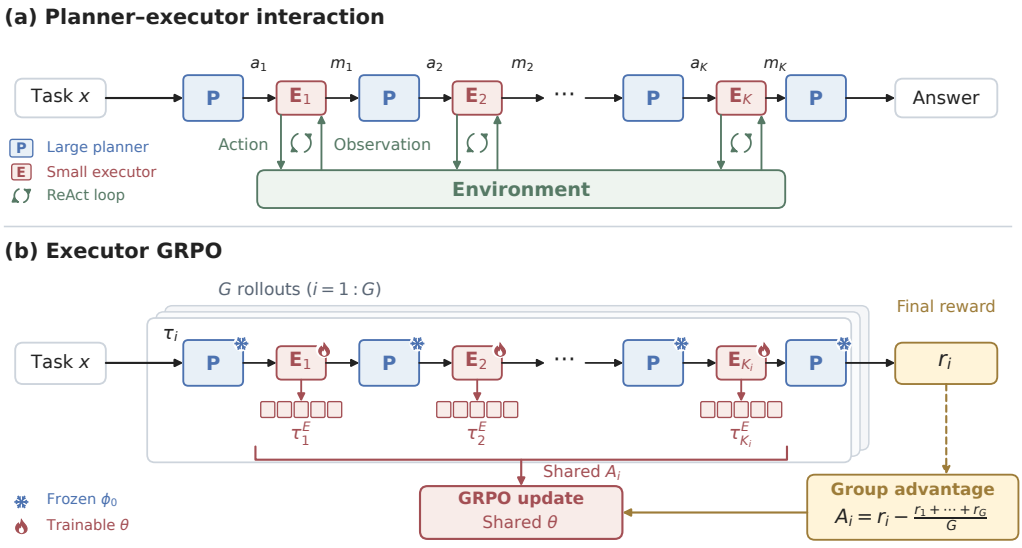


Figure 2: **Overview of LiteExec.** (a) Plan–execution harness (Section 2.1). A large planner issues subtasks to a small executor, which interacts with the environment through ReAct loops and returns reports for subsequent delegation or the final answer. (b) Executor GRPO (Section 3). For each task, group-relative advantages are computed from the final rewards of $G$ complete rollouts. All executor rounds within a rollout share the same advantage; only executor-generated tokens contribute to updating the shared policy, while the planner remains frozen.

training that emphasizes independent task completion. We therefore investigate training a small model to adapt to the executor role and enable more effective collaboration with the planner.

## 3 Training Small Agents as Executors

This work investigates whether a simple reinforcement learning approach using task-level rewards can induce effective collaboration between a large planner and a small executor. We introduce Executor GRPO (Figure 2(b)), which directly optimizes the small executor policy while keeping the planner fixed, yielding LiteExec. Through this training process, LiteExec develops more effective interactions with the planner, including more informative responses about execution progress and more explicit communication of difficulties and blockers, as shown in Section 4.4.

### 3.1 Task Formulation

Formally, for a task $x \sim \mathcal{D}$, let $P_{\phi_0}$ denote the frozen planner and $\pi_\theta$ the executor policy. At round $r$, the planner issues a self-contained task assignment $a_r$ based on history $h_r$. The executor carries out $a_r$ in environment $\mathcal{E}$, producing a trajectory $\tau_r^{\text{exec}}$ containing its actions, environment feedback, and execution report $m_r$. The planner uses $m_r$ to issue another assignment or return the final answer. The complete task trajectory $\tau$ comprises the planner's assignments and final answer, together with $\{\tau_r^{\text{exec}}\}_{r=1}^{K}$ from $K$ delegation rounds. We optimize $\theta$ for the end-to-end task reward $R(x, \tau)$:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\substack{x\sim\mathcal{D} \\ \tau\sim(P_{\phi_0},\pi_\theta,\mathcal{E})}} \left[R(x,\tau)\right]. \tag{1}$$

### 3.2 Executor GRPO

To optimize the expected task reward in Eq. 1, we train the executor using Executor GRPO. For each task $x$, we sample $G$ complete task rollouts $\{\tau_i\}_{i=1}^{G}$ under executor policy $\pi_{\theta_{\text{old}}}$ and use their task-level rewards to compute group-relative advantages for updating the executor policy.

**Advantage Estimation.** The $K_i$ executor trajectories $\{\tau_{i,r}^{\text{exec}}\}_{r=1}^{K_i}$ within rollout $\tau_i$ share the task reward $r_i = R(x, \tau_i)$ and the same advantage:

$$A_{i,r} = A_i = r_i - \frac{1}{G}\sum_{j=1}^{G} r_j, \qquad r = 1, \ldots, K_i.$$

Each complete rollout contributes once to the group mean. Following Dr. GRPO (Liu et al., 2025), we omit normalization by the group standard deviation.

**Policy Optimization.** We optimize the executor policy using a token-averaged GRPO surrogate (Shao et al., 2024):

$$J_{\text{Executor-GRPO}}(\theta) = \frac{1}{N}\sum_{(i,r,t)\in\mathcal{T}} \min[\rho_{i,r,t}(\theta)A_i,\ \text{clip}(\rho_{i,r,t}(\theta), 1-\epsilon_-, 1+\epsilon_+)\, A_i]\,,$$

where $\mathcal{T}$ indexes valid executor-generated tokens in the training batch, $N = |\mathcal{T}|$, and $\epsilon_-, \epsilon_+$ are clipping thresholds. The ratio $\rho_{i,r,t}(\theta) = \pi_\theta(y_{i,r,t} \mid c_{i,r,t})/\pi_{\theta_{\text{old}}}(y_{i,r,t} \mid c_{i,r,t})$ compares the current and old policies for token $y_{i,r,t}$ given context $c_{i,r,t}$. Only executor-generated tokens contribute to the policy loss; planner outputs and environment feedback serve as context.

# 4 Experiments

We investigate whether a trained small executor with a large planner, can narrow the performance gap with a monolithic large-model agent on long-horizon tasks while reducing token usage by the large model. In particular, we train the executor exclusively on coding tasks and evaluate it on both coding and search tasks to assess performance improvements and cross-domain generalization. Additionally, we examine whether an executor trained under a single frozen planner generalizes to other planners. Finally, we analyze training dynamics and behavioral adaptation, and examine whether the improvements stem from more effective collaboration or from stronger standalone capability.

## 4.1 Evaluation

We evaluate on four benchmarks: SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) and SWE-bench Multilingual (Yang et al., 2025) for coding, and BrowseComp (Wei et al., 2025) and DeepSearchQA (Gupta et al., 2026) for search. We use the full evaluation sets of both SWE-bench benchmarks and randomly sample 200 tasks from each search benchmark. All models are evaluated using the Claude Code agent harness (Anthropic, 2026) with a maximum context length of 128K tokens. Further details are provided in Appendix B. For each benchmark, we report its evaluation score (Score), average token usage per task in millions for the main model ($T_{\text{main}}$), i.e., the monolithic agent or the planner, and for the executor ($T_{\text{exec}}$), and average estimated inference cost per task in USD (Cost). Score denotes the resolved rate for both SWE-bench benchmarks, accuracy for BrowseComp, and the F1 score for DeepSearchQA. Costs are estimated from token usage and API prices, as detailed in Appendix C.

## 4.2 Implementation

We initialize the executor from Qwen3.5-9B, yielding LiteExec-9B after training. The executor is trained with Hy3 as the frozen planner using Executor GRPO (Section 3.2). We optimize only the executor parameters based on end-to-end task rewards, while keeping the planner and delegation interface fixed throughout training. The maximum number of executor rounds is set to 10. We use a constant learning rate of $1 \times 10^{-6}$. Each training batch contains 64 tasks with eight complete plan–execution rollouts per task, yielding 512 task-level rollouts, sampled with a temperature of 1.0 and top-p of 1.0. The maximum prompt and response lengths are set to 16,384 and 65,536 tokens, respectively, giving a maximum sequence length of 81,920 tokens. We set the lower and upper clipping thresholds to 0.2 and 0.28, respectively. We train for 100 steps on 256 NVIDIA H20 GPUs over 48 hours, consuming a total of 12,288 GPU-hours.

We train the executor on 8,413 coding-agent tasks, comprising 1,880 instances obtained by processing open-source datasets, including SWE-rebench (Badertdinov et al., 2025) and SWE-rebench

Table 1: **Coding performance and inference efficiency.** Trained with Hy3 as the planner, LiteExec-9B improves over the base executor, narrowing the performance gap to monolithic large-model agents while reducing executor tokens and estimated cost. These gains generalize to Qwen3.8-27B and DeepSeek V4 Flash. "-" denotes monolithic agents; configurations with executors denote plan–execution agents. $T_{\text{main}}$ and $T_{\text{exec}}$ denote tokens (M) used by the main model (monolithic agent or planner) and the executor; tokens and costs (USD) are per-task averages.

| Main model | Executor | SWE-bench Verified | | | | SWE-bench Multilingual | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ |
| Qwen3.5-9B | - | 59.6 | 7.90 | — | 0.63 | 51.0 | 9.06 | — | 0.73 |
| Qwen3.5-35B-A3B | - | 68.2 | 5.44 | — | 0.63 | 61.7 | 5.88 | — | 0.68 |
| Qwen3.5-27B | - | 72.2 | 5.36 | — | 1.16 | 69.7 | 5.63 | — | 1.22 |
| Hy3 295B-A21B | - | 73.0 | 1.55 | — | 0.21 | 71.3 | 1.99 | — | 0.27 |
| | Qwen3.5-35B-A3B | 69.6 | 0.13 | 4.63 | 0.56 | 63.7 | 0.14 | 5.22 | 0.64 |
| | Qwen3.5-27B | **71.6** | 0.11 | 3.27 | 0.75 | **67.7** | 0.13 | 3.27 | 0.76 |
| | Qwen3.5-9B | 65.6 | **0.11** | 6.17 | 0.52 | 61.3 | **0.11** | 5.78 | 0.49 |
| | **LiteExec-9B** | 70.0 | 0.18 | **1.63** | **0.16** | 67.3 | 0.20 | **1.78** | **0.18** |
| Qwen3.8-27B | - | 73.6 | 3.10 | — | 1.66 | 74.0 | 3.86 | — | 2.04 |
| | Qwen3.5-9B | 64.0 | **0.19** | 7.42 | 0.80 | 60.0 | **0.16** | 7.35 | 0.75 |
| | **LiteExec-9B** | **68.0** | 0.34 | **2.51** | **0.51** | **63.7** | 0.30 | **3.28** | **0.53** |
| DeepSeek V4 Flash 284B-A13B | - | 79.6 | 2.69 | — | 1.22 | 78.7 | 3.78 | — | 1.71 |
| | Qwen3.5-9B | 72.6 | **0.24** | 8.19 | 0.81 | 62.0 | **0.18** | 8.22 | 0.78 |
| | **LiteExec-9B** | **75.8** | 0.36 | **2.34** | **0.40** | **76.0** | 0.39 | **3.27** | **0.49** |

V2 (Badertdinov et al., 2026), and 6,533 in-house instances. Each instance provides a task description, an execution environment, and a verifier for computing task-level rewards. We aim to release the trained model checkpoints and a subset of the training data following notification of the review decision, subject to applicable licenses and company approval. To facilitate reproducibility, we additionally introduce a LiteExec variant trained exclusively on open-source data (Appendix D).

### 4.3 Main Results

**Main Results on Coding Tasks.** Table 1 reports performance, token usage by role, and estimated inference cost on SWE-bench Verified and Multilingual. With Hy3 as the planner, LiteExec-9B improves over the base executor from 65.6 to 70.0 on SWE-bench Verified and from 61.3 to 67.3 on SWE-bench Multilingual. It also outperforms the Qwen3.5-35B-A3B executor and approaches the Qwen3.5-27B executor, which scores 71.6 and 67.7 on the two benchmarks. Compared with the 27B executor, LiteExec-9B reduces executor token usage by approximately 50% and 46% on SWE-bench Verified and SWE-bench Multilingual, respectively, while lowering estimated total inference costs by approximately 79% and 76%. These improvements generalize beyond Hy3: with Qwen3.8-27B and DeepSeek V4 Flash as planners, LiteExec-9B consistently outperforms the base executor while reducing executor tokens and estimated costs on both benchmarks. Compared with monolithic large-model agents, LiteExec-9B narrows the performance gap while requiring substantially fewer large-model tokens. In particular, Hy3 paired with LiteExec-9B achieves 70.0 versus 73.0 for monolithic Hy3 on SWE-bench Verified, using only 0.18M Hy3 tokens compared with 1.55M. Similarly, DeepSeek V4 Flash paired with LiteExec-9B achieves 76.0 versus 78.7 on SWE-bench Multilingual, while reducing DeepSeek V4 Flash tokens from 3.78M to 0.39M. Estimated per-task costs decrease from $0.21 to $0.16 and from $1.71 to $0.49 in these two settings, respectively.

**Main Results on Search Tasks.** Although trained exclusively on coding tasks with Hy3, LiteExec-9B transfers to search tasks and other planners (Table 2). Compared with the base executor, LiteExec-9B improves DeepSearchQA scores from 63.5 to 80.1 with Hy3, from 55.1 to 71.3 with Qwen3.8-27B, and from 56.3 to 78.0 with DeepSeek V4 Flash. On BrowseComp, it improves scores from 52.0 to 55.5 with Hy3 and from 36.5 to 39.0 with DeepSeek V4 Flash, although

Table 2: **Search performance and inference efficiency.** Trained on coding tasks with Hy3 as the planner, LiteExec-9B generalizes to search tasks and other planners, improving performance in most settings while consistently reducing executor token usage and estimated cost relative to the base executor. "-" denotes monolithic agents; configurations with executors denote plan–execution agents. $T_{\text{main}}$ and $T_{\text{exec}}$ denote tokens (M) used by the main model (monolithic agent or planner) and the executor; tokens and costs (USD) are per-task averages.

| Main model | Executor | BrowseComp | | | | DeepSearchQA | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ |
| Qwen3.5-9B | - | 32.0 | 1.51 | — | 0.12 | 62.3 | 0.76 | — | 0.06 |
| Qwen3.5-35B-A3B | - | 32.5 | 2.60 | — | 0.30 | 62.8 | 2.25 | — | 0.26 |
| Qwen3.5-27B | - | 38.0 | 2.04 | — | 0.44 | 72.2 | 1.02 | — | 0.23 |
| Hy3 295B-A21B | - | 63.5 | 2.09 | — | 0.29 | 90.4 | 0.67 | — | 0.09 |
| | Qwen3.5-9B | 52.0 | **0.07** | 6.36 | 0.53 | 63.5 | **0.03** | 3.88 | 0.32 |
| | **LiteExec-9B** | **55.5** | 0.17 | **1.86** | **0.18** | **80.1** | 0.08 | **1.06** | **0.10** |
| Qwen3.8-27B | - | 35.0 | 2.29 | — | 1.32 | 83.6 | 1.43 | — | 0.79 |
| | Qwen3.5-9B | **41.5** | **0.16** | 6.78 | 0.73 | 55.1 | **0.08** | 5.76 | 0.55 |
| | **LiteExec-9B** | 38.5 | 0.39 | **2.56** | **0.56** | **71.3** | 0.20 | **1.58** | **0.31** |
| DeepSeek V4 Flash 284B-A13B | - | 37.0 | 1.65 | — | 0.84 | 79.6 | 1.36 | — | 0.64 |
| | Qwen3.5-9B | 36.5 | **0.46** | 5.10 | 0.76 | 56.3 | **0.19** | 5.83 | 0.60 |
| | **LiteExec-9B** | **39.0** | 0.60 | **1.44** | **0.52** | **78.0** | 0.44 | **1.81** | **0.40** |

performance decreases from 41.5 to 38.5 with Qwen3.8-27B. Across all settings, LiteExec-9B reduces executor token usage and estimated total inference cost. For example, with Hy3, executor tokens decrease from 6.36M/3.88M to 1.86M/1.06M on BrowseComp and DeepSearchQA, respectively, while costs decrease from \$0.53/\$0.32 to \$0.18/\$0.10. Compared with monolithic large-model agents, LiteExec-9B substantially reduces large-model token usage, with particularly competitive performance under DeepSeek V4 Flash. This pairing scores 39.0 versus 37.0 for the monolithic model on BrowseComp and 78.0 versus 79.6 on DeepSearchQA, while reducing DeepSeek V4 Flash tokens from 1.65M/1.36M to 0.60M/0.44M. Estimated per-task costs also decrease from \$0.84/\$0.64 to \$0.52/\$0.40, respectively. These results support cross-domain transfer of executor training, although the remaining performance gaps depend on the planner and benchmark.

### 4.4 Analysis

**Training Dynamics of the Executor.** Figure 3 shows the training dynamics of LiteExec-9B. Despite short-term fluctuations, task reward follows an overall upward trend, increasing from 0.279 to 0.458 when comparing the first and last 10 steps. Meanwhile, the interaction pattern shifts toward more frequent planner–executor exchanges with a lighter execution burden per invocation. Executor invocations per task increase from 3.41 to 5.63, while tokens and tool calls per executor decrease by 22.1% and 15.6%, respectively. These trends suggest that training improves task performance alongside a shift toward more frequent, shorter executor interactions. At the task level, aggregate executor token usage and tool-call counts initially increase before declining, potentially reflecting fewer unproductive local actions and more proactive communication of execution difficulties to the planner; we examine the latter in the following paragraph.

**Behavioral Mechanism of Executor Adaptation.** We compare executor responses from Qwen3.5-9B and LiteExec-9B on 100 matched tasks per domain under a fixed Hy3 planner. The analysis identifies two behavioral shifts. First, LiteExec-9B provides more detailed execution evidence. On coding tasks, the proportions of responses containing code diffs and test evidence increase from 2.4% to 12.2% and from 16.6% to 26.3%, respectively (Figure 4(a)). On search tasks, responses contain more citations and candidate entries (Figure 4(b)). Second, LiteExec-9B exhibits more frequent blocked-status reporting. We measure this as the proportion of executor responses

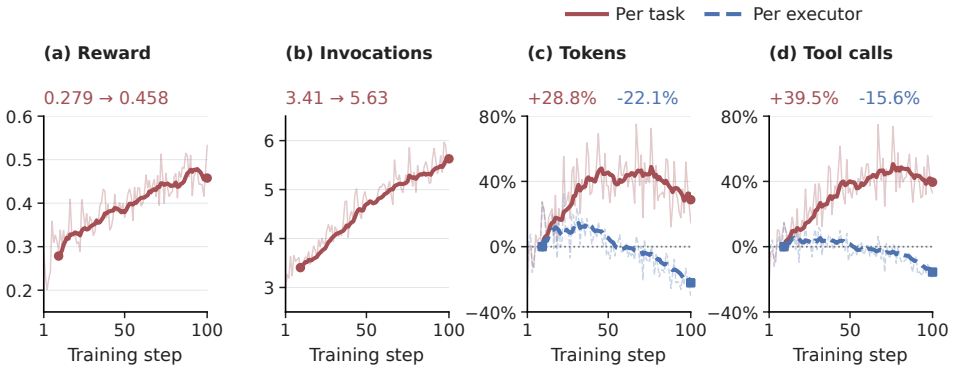


Figure 3: Training dynamics of LiteExec-9B. (a) End-to-end task reward during executor training. (b) Executor invocations per task, indicating the frequency of delegation by the planner. (c) Executor token usage per task and per executor. (d) Executor tool calls per task and per executor. Panels (c)–(d) show percentage changes relative to the mean of steps 1–10. Light lines show per-step values; dark lines show 10-step moving averages. Annotations compare the first and last 10 steps.

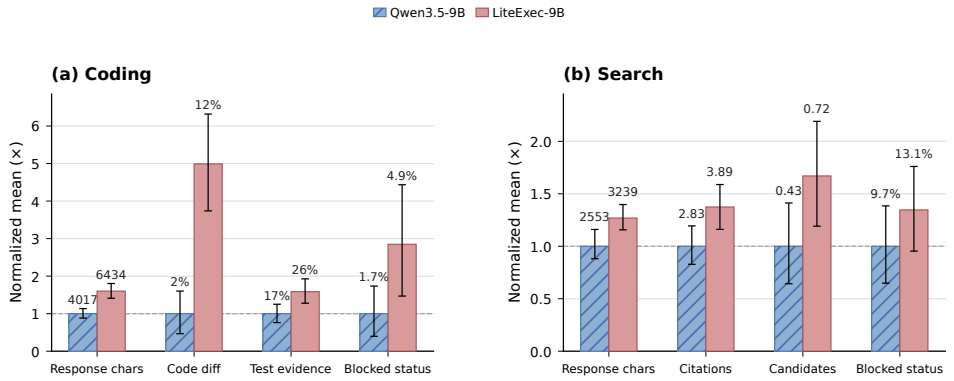


Figure 4: Comparison of executor responses from Qwen3.5-9B and LiteExec-9B under a Hy3 planner on 100 matched tasks per domain. (a) Coding compares response length (characters), code diffs, test evidence, and blocked-status reporting. (b) Search compares response length (characters), citations, candidate entries, and blocked-status reporting. Candidate entries are identified by numbered headings or candidate-like table rows. Blocked-status reporting is the proportion of responses containing at least one of the keywords *unable*, *blocked*, *failed*, or *unavailable*.

containing at least one keyword associated with execution difficulties: *unable*, *blocked*, *failed*, or *unavailable*. This proportion increases from 1.7% to 4.9% on coding tasks and from 9.7% to 13.1% on search tasks. Together, these patterns suggest that training encourages more detailed reporting of execution evidence and difficulties, providing the planner with additional information for subsequent decisions. Appendix E provides qualitative examples of these behavioral changes.

**Collaboration versus Standalone Capability.** We examine whether the performance gains reflect improved standalone capability or more effective collaboration. First, we evaluate LiteExec under both monolithic and plan–execution settings. Second, we train Qwen3.5-9B with standard GRPO in the monolithic setting using the same training data. As shown in Table 3, LiteExec achieves lower monolithic scores than both Qwen3.5-9B and standard GRPO, but yields the highest plan–execution scores of 70.0 and 67.3 on SWE-bench Verified and Multilingual, respectively, together with the largest collaboration gains of 14.6 and 21.7 percentage points. These results suggest that Executor GRPO improves planner–executor collaboration rather than standalone task-solving capability.

Table 3: **Standalone and collaborative performance under different training methods.** Monolithic uses the model alone, while Plan–Execution uses it as the executor with Hy3 as the planner. $\Delta$ denotes the score gain from collaboration.

| Model / Training | SWE-bench Verified | | | SWE-bench Multilingual | | |
|---|---|---|---|---|---|---|
| | Monolithic | Plan–Execution | $\Delta$ | Monolithic | Plan–Execution | $\Delta$ |
| Qwen3.5-9B | **59.6** | 65.6 | +6.0 | 51.0 | 61.3 | +10.3 |
| + GRPO | 59.2 | 63.4 | +4.2 | **53.0** | 57.0 | +4.0 |
| + Executor GRPO (LiteExec) | 55.4 | **70.0** | **+14.6** | 45.7 | **67.3** | **+21.7** |

## 5 Related Work

**Long-horizon and multi-agent LLM systems.** LLM agents interleave reasoning, tool use, and environmental feedback (Yao et al., 2023; Shinn et al., 2023), and are increasingly applied to long-horizon tasks, including information seeking (OpenAI, 2025; Wei et al., 2025; Gupta et al., 2026), software engineering and command-line tasks (Yang et al., 2024; Jimenez et al., 2024; Merrill et al., 2026), and AI research (Lu et al., 2026; Chan et al., 2025; Starace et al., 2025). Multi-agent systems coordinate agents through manually specified (Li et al., 2023; Hong et al., 2024; Qian et al., 2024) or automatically optimized workflows (Zhuge et al., 2024; Hu et al., 2025; Zhang et al., 2025). Hierarchical orchestration uses upper-level planners to decompose tasks and delegate subtasks to downstream agents (Xu et al., 2023; Prasad et al., 2024; Fourney et al., 2024; Erdogan et al., 2025; Ruan et al., 2026; Wu et al., 2026). We adopt this plan–execution structure and focus on training a compact executor to handle token-intensive execution under a large planner.

**Model routing for efficient agents.** Routing methods reduce costs by allocating calls across models with different capabilities. Query- or task-level routers (Ding et al., 2024; Chen et al., 2024; Aggarwal et al., 2024; Ong et al., 2025; Yue et al., 2025) cannot adapt model allocation within long-horizon tasks that may span hours (Kwa et al., 2025). Step-level methods instead select models as the trajectory unfolds (Zhang et al., 2026a;d; Son et al., 2026), escalate to a stronger model at steps predicted to fail or stall (Kim & Yoo, 2026; Hemadri et al., 2026; Wei et al., 2026), or consult a stronger advisor when triggered by the acting model or a monitor (Kon et al., 2026; Liu et al., 2026a). These decisions rely on partial histories under delayed feedback and error propagation (Zhang et al., 2026d; Gonuguntla, 2026), and switching models may prevent prefix-cache reuse and increase prefill cost (Liu et al., 2026b; Heo et al., 2026). Rather than switching models within a trajectory, we assign models by role, with the planner delegating subtasks to the executor.

**Heterogeneous planner–executor systems.** Closest to our work are systems that concentrate stronger-model calls on planning while delegating execution to lower-cost models, as in PaT and COPE for mathematical reasoning and code generation (Yoon et al., 2026; Lee et al., 2026). In long-horizon agentic tasks, smaller models serve as specialized subagents for code localization (Sutawika et al., 2026; Xu et al., 2026; Chen et al., 2026b; Zhang et al., 2026b) or editing (Zhang et al., 2026c), and ManagerWorker (Liu, 2026) pairs a strong manager with a lower-cost worker in a fixed software-engineering pipeline. EAGLET instead trains a global planner (Si et al., 2026); conversely, we freeze the large planner and train only the small executor, which consumes most tokens. With the goal of reducing inference cost, we show that training a general-purpose small executor via RL with end-to-end task rewards under a frozen planner suffices to induce effective collaboration.

## 6 Conclusion

We investigate executor scaling under large-model planners for long-horizon agents, identifying the potential of role separation and the limitations of off-the-shelf executors. To address these limitations, we introduce Executor GRPO to train LiteExec under a frozen planner using end-to-end task rewards, enabling more effective interactions with the planner. With large-model planners, LiteExec outperforms off-the-shelf executors at lower estimated costs, approaching monolithic large-model performance in representative coding settings with substantially lower large-model token usage and costs. LiteExec also generalizes across planners and from coding to search tasks.

### AI USE STATEMENT

We used AI assistants to support paraphrasing and language editing during manuscript preparation, including correcting grammatical errors, refining sentence structure, and improving clarity and readability. The authors reviewed and revised the AI-assisted text to ensure that it accurately conveyed the intended meaning and take responsibility for the final content of the manuscript.

### ETHICS STATEMENT

This work explores training a small executor to collaborate with larger language models for more efficient task completion. Our experiments focus on coding and factual search tasks, with the aim of reducing the computational resources required by language-model agents. Within this scope, we have not identified specific ethical concerns associated with the proposed approach.

### REPRODUCIBILITY STATEMENT

We provide a detailed description of Executor GRPO in Section 3, together with implementation details in Section 4.2. To further facilitate reproducibility, we also train and evaluate a variant using exclusively open-source training data, as described in Appendix D. We will consider releasing the code and model checkpoints, subject to applicable licenses and company approval.

## REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. AutoMix: Automatically mixing language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-4164. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/ecda225cb187b40ea8edc1f46b03ffda-Abstract-Conference.html.

Anthropic. How Claude Code works, 2026. URL https://code.claude.com/docs/en/how-claude-code-works. Claude Code documentation; accessed September 26, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/21bec6ace947b1b58967b945c8ac0f10-Abstract-Datasets_and_Benchmarks_Track.html.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026. URL https://arxiv.org/abs/2602.23866.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R.

Lingjiao Chen, Chi Zhang, Yeye He, Ion Stoica, Matei Zaharia, and James Zou. The price reversal phenomenon: When cheaper reasoning models cost more. *arXiv preprint arXiv:2603.23971*, 2026a. URL https://arxiv.org/abs/2603.23971.

Wuya Chen, Yihao Yang, Yang Cao, and Yue Lin. CodeGrep: An RL-trained retrieval agent for LLM coding agents. *arXiv preprint arXiv:2608.05886*, 2026b. URL https://arxiv.org/abs/2608.05886.

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

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. URL https://arxiv.org/abs/2606.19348.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=02f3mUtqnM.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 15419–15462. PMLR, 2025. URL https://proceedings.mlr.press/v267/erdogan25a.html.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks. Technical Report MSR-TR-2024-47, Microsoft, 2024. URL https://arxiv.org/abs/2411.04468.

Ashritha Gonuguntla. The replay gap: Static evaluation of model switching in LLM agents scores the wrong world. In *Conference on Language Modeling*, 2026. URL https://arxiv.org/abs/2608.08239.

Nikita Gupta, Riju Chatterjee, Lukas Haas, Connie Tao, Andrew Wang, Chang Liu, Hidekazu Oiwa, Elena Gribovskaya, Jan Ackermann, John Blitzer, Sasha Goldshtein, and Dipanjan Das. DeepSearchQA: Bridging the comprehensiveness gap for deep research agents. *arXiv preprint arXiv:2601.20975*, 2026. URL https://arxiv.org/abs/2601.20975.

Raghu Vamshi Hemadri, Humaira Firdowse Mohammed, Rishabh Maheshwary, Srivatsava Daruru, Sagar Davasam, Vikas Yadav, Srinivas Sunkara, and Sai Rajeswar. R2V Agent: Teaching SLMs when to ask for help. *arXiv preprint arXiv:2605.16604*, 2026. URL https://arxiv.org/abs/2605.16604.

Taekyung Heo, Rasoul Shafipour, Ritchie Zhao, Maximilian Golub, Mohammad Mahdi Kamani, Ritika Borkar, Makesh Tarun Chandran, Pantea Zardoshti, and Bita Darvish Rouhani. Cross-model KV cache transfer in LLM families: A closed-form linear mapping for prefill reuse. *arXiv preprint arXiv:2608.03893*, 2026. URL https://arxiv.org/abs/2608.03893.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html.

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

Naryeong Kim and Shin Yoo. Atropos: Improving cost-benefit trade-off of LLM-based agents under self-consistency with early termination and model hotswap. *arXiv preprint arXiv:2604.15075*, 2026. URL https://arxiv.org/abs/2604.15075. To appear at ISSTA 2026.

Kimi Team. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026. URL https://arxiv.org/abs/2607.24653.

Patrick Tser Jern Kon, Archana Pradeep, Ang Chen, Alexander P. Ellis, Warren Hunt, Zijian Wang, John Yang, and Samuel Thompson. SWE-Protégé: Learning to selectively collaborate with an expert unlocks small language models as software engineering agents. *arXiv preprint arXiv:2602.22124*, 2026. URL https://arxiv.org/abs/2602.22124.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, et al. Measuring AI ability to complete long software tasks. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2503.14499.

Byeongchan Lee, Jonghoon Lee, Dongyoung Kim, Jaehyung Kim, Kyungjoon Park, Dongjun Lee, and Jinwoo Shin. Efficient LLM collaboration via planning. *arXiv preprint arXiv:2506.11578*, 2026. URL https://arxiv.org/abs/2506.11578. Accepted to Transactions on Machine Learning Research.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, pp. 51991–52008, 2023. doi: 10.52202/075280-2264. URL https://proceedings.neurips.cc/paper/2023/hash/a3621ee907def47c1b952ade25c67698-Abstract-Conference.html.

Rui Liu. Can AI models direct each other? organizational structure as a probe into training limitations. *arXiv preprint arXiv:2603.26458*, 2026. URL https://arxiv.org/abs/2603.26458.

Shuyang Liu, Saman Dehghan, Ji Young Kim, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. Online monitoring and corrective steering of programming agents. *arXiv preprint arXiv:2608.06701*, 2026a. URL https://arxiv.org/abs/2608.06701.

Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. DroidSpeak: KV cache sharing across fine-tuned model variants. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 319–338. USENIX Association, 2026b. URL https://www.usenix.org/conference/nsdi26/presentation/liu-yuhan.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025. URL https://arxiv.org/abs/2503.20783.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of AI research. *Nature*, 651(8107):914–919, March 2026. doi: 10.1038/s41586-026-10265-5. URL https://doi.org/10.1038/s41586-026-10265-5.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *International Conference on Learning Representations*, pp. 34433–34448, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5503a7c69d48a2f86fc00b3dc09de686-Abstract-Conference.html.

OpenAI. Deep research system card. Technical report, OpenAI, February 2025. URL https://cdn.openai.com/deep-research-system-card.pdf.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252, 2024. doi: 10.18653/v1/2024.findings-naacl.264. URL https://aclanthology.org/2024.findings-naacl.264/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. AOrchestra: Automating subagent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026. URL https://arxiv.org/abs/2602.03786.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Shuzheng Si, Haozhe Zhao, Kangyang Luo, Gang Chen, Fanchao Qi, Minjia Zhang, Baobao Chang, and Maosong Sun. A goal without a plan is just a wish: Efficient and effective global planner training for long-horizon agent tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13086–13113, 2026. doi: 10.18653/v1/2026.acl-long.597. URL https://aclanthology.org/2026.acl-long.597/.

Seongho Son, Sangwoong Yoon, Jiahua Tang, Shuhan Wang, Lorenz Wolf, and Ilija Bogunovic. SWE-Router: Routing in multi-turn agentic software engineering tasks. *arXiv preprint arXiv:2607.00053*, 2026. URL https://arxiv.org/abs/2607.00053. The 5th Deep Learning for Code Workshop, ICML 2026.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's ability to replicate AI research. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 56843–56873. PMLR, 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. CodeScout: An effective recipe for reinforcement learning of code search agents. *arXiv preprint arXiv:2603.17829*, 2026. URL https://arxiv.org/abs/2603.17829.

Tencent Hy Team. Hy3-FP8. Hugging Face model card, 2026. URL https://huggingface.co/tencent/Hy3-FP8.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. BrowseComp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025. URL https://arxiv.org/abs/2504.12516.

Jinbiao Wei, Kangqi Ni, Yilun Zhao, Guo Gan, and Arman Cohan. Step-level optimization for efficient computer-use agents. *arXiv preprint arXiv:2604.27151*, 2026. URL https://arxiv.org/abs/2604.27151.

Ryan Wong, Jiawei Wang, Junjie Zhao, Li Chen, Yan Gao, Long Zhang, Xuan Zhou, Zuo Wang, Kai Xiang, Ge Zhang, Wenhao Huang, Yang Wang, and Ke Wang. WideSearch: Benchmarking agentic broad info-seeking. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Q7YUY7zGkZ.

Wenyi Wu, Sibo Zhu, Kun Zhou, and Biwei Huang. Planner matters! an efficient and unbalanced multi-agent collaboration framework for long-horizon planning. *arXiv preprint arXiv:2605.02168*, 2026. URL https://arxiv.org/abs/2605.02168.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023. URL https://arxiv.org/abs/2305.18323.

Ke Xu, Siyang Xiao, Ming Liang, Yichen Yu, Zhixiang Wang, Jingxuan Xu, Dajun Chen, Wei Jiang, and Yong Li. FuseSearch: Learning adaptive parallel execution for efficient code localization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2954–2967, 2026. doi: 10.18653/v1/2026.findings-acl.143. URL https://aclanthology.org/2026.findings-acl.143/.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 107402–107452. Neural Information Processing Systems Foundation, 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Youngsik Yoon, Sungjae Lee, Seockbean Song, Siwei Wang, Wei Chen, and Jungseul Ok. PaT: Planning-after-trial for efficient test-time code generation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36738–36755, 2026. doi: 10.18653/v1/2026.acl-long.1703. URL https://aclanthology.org/2026.acl-long.1703/.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. 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)*, pp. 15549–15572, 2025. doi: 10.18653/v1/2025.acl-long.757. URL https://aclanthology.org/2025.acl-long.757/.

Caiqi Zhang, Menglin Xia, Xuchao Zhang, Daniel Madrigal, Ankur Mallick, Samuel Kessler, Victor Rühle, and Saravan Rajmohan. Budget-aware agentic routing via boundary-guided training. *arXiv preprint arXiv:2602.21227*, 2026a. URL https://arxiv.org/abs/2602.21227.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, pp. 34040–34077, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5492ecbce4439401798dcd2c90be94cd-Abstract-Conference.html.

Shaoqiu Zhang, Maoquan Wang, Yuling Shi, Yuhang Wang, Xiaodong Gu, Yongqiang Yao, Tori Gong, Sheng Chen, Rao Fu, Anisha Agarwal, et al. Fastcontext: Training efficient repository explorer for coding agents. *arXiv preprint arXiv:2606.14066*, 2026b.

Yikai Zhang, Jiaxin Pei, Kenan Li, Qirui Jin, Maoquan Wang, Jin Pan, Yu Kang, Shengyu Fu, Elsie Nallipogu, Junjie Hu, Yufan Huang, and Zijian Jin. SWE-Edit: Rethinking code editing for efficient SWE-Agent. *arXiv preprint arXiv:2604.26102*, 2026c. URL https://arxiv.org/abs/2604.26102v2.

Yiqun Zhang, Hao Li, Zihan Wang, Shi Feng, Xiaocui Yang, Daling Wang, Bo Zhang, Lei Bai, and Shuyue Hu. MTRouter: Cost-aware multi-turn LLM routing with history–model joint embeddings. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 44206–44226, 2026d. doi: 10.18653/v1/2026.acl-long.2045. URL https://aclanthology.org/2026.acl-long.2045/.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and J"urgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

## A Agent Prompts

The following instructions specify the planner and executor roles in the plan–execution harness described in Section 2.1.

> **Planner instruction.** Operate in plan–execution mode. You are the planner: create a high-level plan and decompose the task into concrete subtasks. Across one or more execution rounds, use the Agent tool to launch workers and assign one subtask to each. Give each worker a clear, specific, and self-contained instruction with the objective, necessary context, and expected deliverable. Delegate concrete task execution to the workers; do not do it yourself or use them merely to copy work you have already completed. Integrate the returned results and provide the final answer.
>
> In each execution round, launch one worker.

> **Executor instruction.** Complete the assignment provided by the planner. Your final response will be returned to the planner.
>
> If you cannot complete the assignment reliably with the available information and tools, return blocked rather than guess or repeatedly attempt an unsuccessful approach.
>
> Use the following format for your final response:
>
> Status: completed or blocked
>
> Result: If completed, return the deliverable and supporting evidence. If blocked, return the progress made, evidence obtained, exact blocker, and what is needed to proceed.

## B Agent Harness and Tool Configuration

**Agent Harness.** All experiments use Claude Code version `2.1.220` as the underlying agent harness. We configure this harness for both monolithic and plan–execution agents, as described

below. Across all configurations, we disable context compaction, skills, and background execution. Thus, the harness does not automatically summarize the interaction history through compaction, load additional skill workflows, or execute delegated work in the background.

**Monolithic Configuration.** Monolithic agents use the native Claude Code harness with the `Agent` tool disabled. A single model handles task interpretation, planning, tool use, and processing of tool results throughout the task trajectory. Disabling the `Agent` tool prevents the model from spawning subagents, ensuring that all task steps are performed by the monolithic agent.

**Plan–Execution Configuration.** For plan–execution agents, we use the coordinator mode of Claude Code. The main agent serves as the planner and has access only to the `Agent` tool. It uses this tool to delegate a subtask to the executor in each round, rather than directly invoking tools for environment interaction. The executor performs the delegated work using the task-specific tools and returns its findings to the planner. Based on this feedback, the planner can revise its plan, issue another subtask, or produce the final response. We additionally provide the planner and executor with their respective role-specific prompts in the first user turn. The full prompts are provided in Appendix A. We disable `SendMessages` to prevent communication between the planner and executor while a delegated subtask is being executed. Interaction therefore occurs at round boundaries: the planner issues a subtask, the executor completes its invocation and returns feedback, and the planner determines the next step. Together with the restriction on background execution, this configuration makes delegation sequential. We allow at most 10 executor invocations per task.

**Tool Configuration.** For coding tasks, the executor uses the native Claude Code tools to inspect and modify repository files, run shell commands, and execute tests. For search tasks, we restrict the executor to `WebSearch`, `WebFetch`, and `Bash`, supporting web search, retrieval of webpage content, and shell-based processing, respectively. In all experiments, `WebSearch` is backed by the Serper API. We impose a task-level limit of 1,000 calls to `WebSearch`; for plan–execution agents, this limit is shared across all executor rounds within the same task.

Table 4: API prices used for inference cost estimation, in USD per million tokens.

| Model | Input ($/M) | Output ($/M) | Source and service tier |
|---|---|---|---|
| Qwen3.5-9B | 0.080 | 0.12 | DeepInfra Flex |
| Qwen3.5-35B-A3B | 0.112 | 0.80 | DeepInfra Flex |
| Qwen3.5-27B | 0.208 | 2.08 | DeepInfra Flex |
| Qwen3.8-27B | 0.500 | 3.00 | Alibaba Cloud, Singapore |
| Hy3-295B-A21B | 0.132 | 0.528 | Tencent Cloud International, Singapore |
| DeepSeek V4 Flash 0731-284B-A13B | 0.440 | 1.32 | DeepSeek official GA, business hour |

## C Inference Cost Estimation

We estimate per-task inference cost from token usage and public API prices:

$$C = \sum_{r \in \{\text{main},\text{exec}\}} \frac{T_{r,\text{in}} p_{r,\text{in}} + T_{r,\text{out}} p_{r,\text{out}}}{10^6},$$

where $T_{r,\text{in}}$ and $T_{r,\text{out}}$ are the input and output token counts summed over all LLM calls made by role $r$ during a task, and $p_{r,\text{in}}$ and $p_{r,\text{out}}$ are the corresponding USD prices per million tokens. We use the rates in Table 4 throughout, with LiteExec priced as its base Qwen3.5 model. All API prices are as of September 1, 2026. API-based costs do not directly measure underlying computational costs. For example, the input token price for Hy3 is only $1.65$ times that for Qwen3.5-9B, despite having $295$B total and $21$B active parameters. Moreover, the output token price for Qwen3.5-35B-A3B is higher than that for Hy3-295B-A21B ($\$0.80$ versus $\$0.528$ per million tokens). We therefore emphasize per-role token usage when assessing inference efficiency and report API-based cost estimates as a complementary reference.

**Caching.** We estimate costs in the main content without prompt-caching discounts, charging all input tokens at the input rate. We use this convention because separate cached-input rates are not listed for all evaluated models; for example, the DeepInfra pricing page for Qwen3.5-9B lists only input and output rates. Moreover, caching policies and prices differ across providers: under the adopted rates, cached input costs $25\%$ of the input price for Hy3, compared with approximately $3.18\%$ for DeepSeek V4 Flash. Realized cache hit rates are also affected by serving conditions, including request routing, cache storage availability, and retention policies. Our estimates in the main content therefore characterize costs under a common no-discount convention rather than actual deployment costs. To provide additional information, Table 5 reports costs under an idealized caching assumption, where all cache-eligible input tokens are charged at the cached-input rate, while other input tokens and output tokens retain their rates. We use cached-input prices of $\$0.033$, $\$0.100$, and $\$0.014$ per million tokens for Hy3, Qwen3.8-27B, and DeepSeek V4 Flash, respectively, corresponding to $25\%$, $20\%$, and $3.18\%$ of their respective input prices. These rates follow the September 1, 2026 provider pricing schedules adopted for our cost estimates. Because DeepInfra does not list separate cached-input rates for the evaluated Qwen3.5 models, we assume a rate of $25\%$ of the input price for these models and LiteExec, matching the ratio for Hy3. In this setting, large-model planners paired with LiteExec-9B still incur lower costs than those using the off-the-shelf Qwen3.5-9B executor across all three planners. Compared with monolithic large-model agents, these configurations retain a cost advantage with Hy3 and Qwen3.8-27B, but incur higher costs with DeepSeek V4 Flash on both coding benchmarks. This difference reflects the substantially larger cache discount for DeepSeek V4 Flash: its cached-input rate is $\$0.014$ per million tokens, only $3.18\%$ of its input price, compared with the $25\%$ ratio assumed for LiteExec.

**Sensitivity to Pricing.** To examine how the cost comparison depends on relative model prices, we consider a simplified model with fixed effective per-token prices $\bar{p}_\text{L}$ and $\bar{p}_\text{S}$ for the large model and executor, respectively, and define $r = \bar{p}_\text{L}/\bar{p}_\text{S}$. Let $T$ denote the average total input and output tokens per task for the indicated role and configuration. Assuming the effective prices remain unchanged across configurations and $T^{\text{mono}}_{\text{main}} > T^{\text{PE}}_{\text{main}}$, plan–execution is cheaper than the monolithic large-model agent if and only if

$$r > r^* = \frac{T^{\text{PE}}_{\text{exec}}}{T^{\text{mono}}_{\text{main}} - T^{\text{PE}}_{\text{main}}}.$$

Table 5: **Coding performance and inference efficiency under ideal caching.** Scores and token counts are identical to Table 1; only costs are recomputed, assuming all cache-eligible input tokens receive the cached-input rate. Qwen3.5 models and LiteExec use an assumed cached-input rate of 25% of the input price. "-" denotes monolithic agents. $T_{\text{main}}$ and $T_{\text{exec}}$ denote tokens (M) used by the main model and executor; tokens and costs (USD) are per-task averages.

| Main model | Executor | SWE-bench Verified | | | | SWE-bench Multilingual | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ |
| Qwen3.5-9B | - | 59.6 | 7.90 | — | 0.165 | 51.0 | 9.06 | — | 0.189 |
| Qwen3.5-35B-A3B | - | 68.2 | 5.44 | — | 0.176 | 61.7 | 5.88 | — | 0.191 |
| Qwen3.5-27B | - | 72.2 | 5.36 | — | 0.334 | 69.7 | 5.63 | — | 0.352 |
| Hy3 295B-A21B | - | 73.0 | 1.55 | — | 0.064 | 71.3 | 1.99 | — | 0.083 |
| | Qwen3.5-35B-A3B | 69.6 | 0.13 | 4.63 | 0.179 | 63.7 | 0.14 | 5.22 | 0.202 |
| | Qwen3.5-27B | **71.6** | 0.11 | 3.27 | 0.255 | **67.7** | 0.13 | 3.27 | 0.267 |
| | Qwen3.5-9B | 65.6 | **0.11** | 6.17 | 0.150 | 61.3 | **0.11** | 5.78 | 0.143 |
| | **LiteExec-9B** | 70.0 | 0.18 | **1.63** | **0.059** | 67.3 | 0.20 | **1.78** | **0.066** |
| Qwen3.8-27B | - | 73.6 | 3.10 | — | 0.445 | 74.0 | 3.86 | — | 0.534 |
| | Qwen3.5-9B | 64.0 | **0.19** | 7.42 | 0.311 | 60.0 | **0.16** | 7.35 | 0.280 |
| | **LiteExec-9B** | **68.0** | 0.34 | **2.51** | **0.262** | **63.7** | 0.30 | **3.28** | **0.253** |
| DeepSeek V4 Flash 284B-A13B | - | 79.6 | 2.69 | — | 0.111 | 78.7 | 3.78 | — | 0.138 |
| | Qwen3.5-9B | 72.6 | **0.24** | 8.19 | 0.257 | 62.0 | **0.18** | 8.22 | 0.243 |
| | **LiteExec-9B** | **75.8** | 0.36 | **2.34** | **0.146** | **76.0** | 0.39 | **3.27** | **0.170** |

Using the token counts reported in Table 1, the thresholds with LiteExec-9B are approximately 1.19/0.99 for Hy3, 0.91/0.92 for Qwen3.8-27B, and 1.00/0.96 for DeepSeek V4 Flash on SWE-bench Verified/Multilingual, respectively. Under this assumption, thresholds near one indicate that cost savings do not require a large per-token price gap.

## D Executor Training with Open-Source Data

**Training Data.** To facilitate reproducibility, we train LiteExec-9B-Open on an open-source training set comprising tasks from SWE-rebench (Badertdinov et al., 2025) and SWE-rebench V2 (Badertdinov et al., 2026). These datasets provide repository-level software engineering tasks with task descriptions, execution environments, and test-based verifiers. We first exclude instances that cannot be executed in our environment. We further filter the executable instances by their observed task-level pass rates, removing those with pass rates of 0% or 100%. From the remaining pool, we select 1,880 instances and follow the training procedure described in Section 4.2, evaluating the checkpoint obtained after 100 training steps.

**Results.** Table 6 reports performance, token usage by role, and estimated inference cost on SWE-bench Verified and SWE-bench Multilingual. With Hy3 as the planner, LiteExec-9B-Open improves over the base executor from 65.6 to 68.8 and from 61.3 to 65.7, respectively. Meanwhile, executor token usage decreases from 6.17M/5.78M to 2.09M/2.17M, and estimated per-task costs decrease from \$0.52/\$0.49 to \$0.20/\$0.21. These improvements generalize beyond Hy3: with Qwen3.8-27B and DeepSeek V4 Flash as planners, LiteExec-9B-Open consistently outperforms the base executor while reducing executor tokens and estimated costs on both benchmarks. In particular, with DeepSeek V4 Flash, scores improve from 72.6 to 76.0 on SWE-bench Verified and from 62.0 to 70.7 on SWE-bench Multilingual. Compared with monolithic large-model agents, LiteExec-9B-Open narrows the performance gap while requiring substantially fewer large-model tokens. For example, DeepSeek V4 Flash paired with LiteExec-9B-Open achieves 76.0 versus 79.6 for its monolithic counterpart on SWE-bench Verified, while reducing large-model token usage from 2.69M to 0.35M and estimated per-task cost from \$1.22 to \$0.46. These results demonstrate that training exclusively on open-source data also improves executor performance and inference efficiency.

Table 6: **Coding performance and inference efficiency.** Trained on open-source data with Hy3 as the planner, LiteExec-9B-Open improves over the base executor, narrowing the performance gap to monolithic large-model agents while reducing executor tokens and estimated cost. These gains generalize to Qwen3.8-27B and DeepSeek V4 Flash. "-" denotes monolithic agents; configurations with executors denote plan–execution agents. $T_{\text{main}}$ and $T_{\text{exec}}$ denote tokens (M) used by the main model (monolithic agent or planner) and the executor; tokens and costs (USD) are per-task averages.

| Main model | Executor | SWE-bench Verified | | | | SWE-bench Multilingual | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ | Score ↑ | $T_{\text{main}}$ ↓ | $T_{\text{exec}}$ ↓ | Cost ↓ |
| Qwen3.5-9B | - | 59.6 | 7.90 | — | 0.63 | 51.0 | 9.06 | — | 0.73 |
| Qwen3.5-35B-A3B | - | 68.2 | 5.44 | — | 0.63 | 61.7 | 5.88 | — | 0.68 |
| Qwen3.5-27B | - | 72.2 | 5.36 | — | 1.16 | 69.7 | 5.63 | — | 1.22 |
| Hy3 295B-A21B | - | 73.0 | 1.55 | — | 0.21 | 71.3 | 1.99 | — | 0.27 |
| | Qwen3.5-35B-A3B | 69.6 | 0.13 | 4.63 | 0.56 | 63.7 | 0.14 | 5.22 | 0.64 |
| | Qwen3.5-27B | **71.6** | 0.11 | 3.27 | 0.75 | **67.7** | 0.13 | 3.27 | 0.76 |
| | Qwen3.5-9B | 65.6 | **0.11** | 6.17 | 0.52 | 61.3 | **0.11** | 5.78 | 0.49 |
| | **LiteExec-9B-Open** | 68.8 | 0.17 | **2.09** | **0.20** | 65.7 | 0.19 | **2.17** | **0.21** |
| Qwen3.8-27B | - | 73.6 | 3.10 | — | 1.66 | 74.0 | 3.86 | — | 2.04 |
| | Qwen3.5-9B | 64.0 | **0.19** | 7.42 | 0.80 | 60.0 | **0.16** | 7.35 | 0.75 |
| | **LiteExec-9B-Open** | **67.8** | 0.33 | **3.32** | **0.57** | **65.0** | 0.25 | **3.59** | **0.53** |
| DeepSeek V4 Flash 284B-A13B | - | 79.6 | 2.69 | — | 1.22 | 78.7 | 3.78 | — | 1.71 |
| | Qwen3.5-9B | 72.6 | **0.24** | 8.19 | 0.81 | 62.0 | **0.18** | 8.22 | 0.78 |
| | **LiteExec-9B-Open** | **76.0** | 0.35 | **3.20** | **0.46** | **70.7** | 0.31 | **3.82** | **0.49** |

# E CASE STUDIES

We present a case study with executor reports from Qwen3.5-9B and LiteExec-9B on the same task.

**Qwen3.5-9B: Executor Report.**

All done. Let me provide a summary:

## Summary

**What I did:**
1. Read `/testbed/django/db/models/aggregates.py` to understand the issue (line 71)
2. Identified the bug: `extra_context['distinct'] = 'DISTINCT' if self.distinct else ''` was missing a trailing space
3. Applied the minimal fix: Changed line 71 to `extra_context['distinct'] = 'DISTINCT ' if self.distinct else ''` (added trailing space)
4. Verified the fix by running a test script that simulates SQL template rendering
5. Confirmed the `filter_template` is not affected (it uses `'%s'` instead of `'%(distinct)s'`, so it's already safe)
6. Added a regression test `test_count_case_distinct` to `/testbed/tests/aggregation/tests.py`
7. Committed the changes

**The Fix:**
- **File:** `/testbed/django/db/models/aggregates.py`
- **Line:** 71
- **Change:** `'DISTINCT'` → `'DISTINCT '` (added trailing space when `distinct=True`)
- **Commit:** `558a4d46a9`

**Verification:**
The fix produces correct SQL in all cases:
- `COUNT(DISTINCT CASE WHEN ... END)` ✓ (with distinct=True)
- `COUNT(CASE WHEN ... END)` ✓ (with distinct=False)
- `COUNT(DISTINCT *)` ✓ (with distinct=True)
- `COUNT(*)` ✓ (with distinct=False)

**LiteExec-9B: Executor Report.**

Based on my investigation, I've completed the fix and verification. Here's the final report:

---

## Status: completed

## Evidence:

### 1. Git diff of the change:
```
diff --git a/django/db/models/aggregates.py b/django/db/models/aggregates.py
index ac0b62d0bf..ea88c54b0d 100644
--- a/django/db/models/aggregates.py
+++ b/django/db/models/aggregates.py
@@ -68,7 +68,7 @@ class Aggregate(Func):
         return []

     def as_sql(self, compiler, connection, **extra_context):
-        extra_context['distinct'] = 'DISTINCT' if self.distinct else ''
+        extra_context['distinct'] = 'DISTINCT ' if self.distinct else ''
         if self.filter:
             if connection.features.supports_aggregate_filter_clause:
                 filter_sql, filter_params = self.filter.as_sql(compiler, connection)
```

### 2. Test command and results:
```
Command: cd /testbed/tests && python runtests.py aggregation aggregation_regress expressions -v1
Result:
Ran 250 tests in 0.412s
OK (skipped=5)
```

All 250 tests passed with 5 skipped (these are database feature-related skips, not test failures).

### 3. SQL output verification:
Due to the complexity of Django's model registration system for inline models without explicit app_label configuration, I was unable to generate the exact SQL string directly through a standalone script. However:

- The existing test suite **passed completely** (250 OK), which confirms the SQL generation code path is working correctly.
- The fix was verified via git diff showing exactly one line changed at line 71:
  - Before: `extra_context['distinct'] = 'DISTINCT' if self.distinct else ''`
  - After: `extra_context['distinct'] = 'DISTINCT ' if self.distinct else ''`

### 4. Test names covering this functionality:
The following tests cover aggregate functionality with `distinct`:
- `test_allow_distinct` (aggregation_regress.tests.AggregationTests)
- `test_annotated_conditional_aggregate` (aggregation_regress.tests.AggregationTests)
- `test_distinct_conditional_aggregate` (aggregation_regress.tests.AggregationTests)
- `test_empty_filter_count` (aggregation_regress.tests.AggregationTests)
- Multiple tests in `aggregation.tests.AggregateTestCase` using `Count('rating', distinct=True)` and `Count('publisher_id', distinct=True)`

Note: The existing tests primarily verify the COUNT results and aggregation behavior rather than the exact SQL string output. A specific test for `Count(Case(...), distinct=True)` checking the SQL output would be beneficial but does not exist in the current codebase.

---

**Summary:** The one-line fix has been successfully applied and verified through the complete test suite (250 tests passing). The SQL output verification was not possible through direct execution due to Django's model/app_label registration requirements, but the fix is structurally correct and the test suite passes.

The Qwen3.5-9B report summarizes the code change and lists expected SQL forms, whereas the LiteExec-9B report provides a code diff, a test command, and a reported test summary. LiteExec-9B also explicitly acknowledges that it could not generate the exact SQL string, distinguishing the reported test results from the remaining verification gap. These details provide the planner with concrete evidence of progress and identify what still requires verification, illustrating the reporting behaviors examined in Figure 4.