

A²AGENT: Action-Aware Reinforcement Learning for Repository-Level Code Localization Agents

Doyeon Kim*, Suyoung Bae*, Yumin Lee, Jee-Hyong Lee†

College of Computing and Informatics
Sungkyunkwan University, South Korea
{kimdozz0, sybae01, totoandy, john}@skku.edu

Abstract

Localizing issue-relevant code regions is a critical step in automated software engineering. However, due to their reliance on sparse trajectory-level signals, existing methods cannot identify which per-turn actions are effective and often discover correct code regions during exploration but fail to commit them. To address these limitations, we propose an action-aware reinforcement learning method that combines a *per-turn reward sequence* rewarding both the discovery and commitment of gold code regions with an *action-level advantage estimation* scheme that isolates each action’s credit by grouping turns sharing the same exploration context. Extensive evaluations show that our method improves the average F1 over the state-of-the-art (SOTA) by 1.58% on SWE-Bench Verified and 8.55% on SWE-Bench Pro, with our 4B model outperforming baselines up to 8× larger. Our code is available at <https://github.com/donian00/A2Agent>.

1 Introduction

Recent advances in large language models (LLMs) have shown strong performance across a wide range of code-related tasks (Chen et al., 2021; Roziere et al., 2023; Guo et al., 2024). Among them, code localization, which identifies the specific code regions that must be modified to resolve a real software issue, has become a central research focus. Accurate localization is a prerequisite for any downstream issue resolution, and inaccurate localization fundamentally limits the quality of the resulting issue resolution (Jimenez et al., 2024).

As modern software repositories grow in scale and dependency complexity, a single LLM inference is no longer sufficient to localize all relevant code regions. As a result, agent-based approaches have emerged as the dominant paradigm, where

*Equal contribution

†Corresponding author

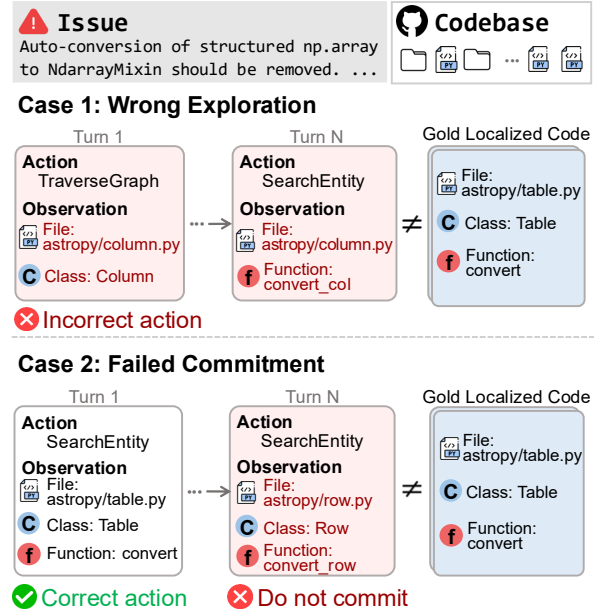


Figure 1: Failure cases of previous agentic LLMs for repository-level code localization. **(Case 1) Wrong exploration:** Agents cannot identify effective actions at each turn, causing unnecessary tool calls that do not contribute to issue resolution. **(Case 2) Failed commitment:** Even though agents observe the gold region during localization, they fail to recognize and commit it to the final prediction.

models iteratively invoke tools to navigate the codebase (Chen et al., 2025; Yu et al., 2025; Jiang et al., 2025; Zhang et al., 2025).

Early approaches leverage few-shot prompting to elicit tool use and multi-step reasoning from pre-trained LLMs (Yao et al., 2022; Wei et al., 2022). However, such demonstrations are insufficient for learning complex multi-turn tool use. Therefore, recent work has shifted toward post-training paradigms, training code localization agents via supervised fine-tuning (SFT) (Chen et al., 2025; Ma et al., 2025; Pan et al., 2024) or reinforcement learning with verifiable rewards (RLVR) (Zhang et al., 2025; Sutawika et al., 2026; Wei et al., 2026). However, several limitations remain unresolved by

these approaches, as illustrated in Figure 1.

Existing methods fail to distinguish the contribution and error of each action taken at individual turns, resulting in redundant tool calls that do not contribute to issue resolution. In multi-turn code localization, actions at each turn may either contribute to identifying the correct code regions or mislead the search through inappropriate tool calls (Wang et al., 2025a; Yu et al., 2025). However, existing methods rely on either a single gold trajectory or a sparse outcome-level signal, which cannot separate correct actions from erroneous ones. The agent thus cannot reinforce effective exploration or discount counterproductive behavior.

Moreover, in multi-turn code localization, the agent may observe the correct code regions during exploration but fail to commit them to the final prediction, instead invoking additional tools and outputting incorrect regions (analysis illustrated in Figure 3). Such failures persist because existing trajectory-level sparse rewards treat these explored-but-uncommitted trajectories the same as ones that never reached the correct elements. The discovery-commit gap thus remains unresolved.

To address these challenges, we propose **A²AGENT**, an Action-Aware reinforcement learning method for repository-level code localization **AGENTS**. To strengthen the agent’s ability to perform effective exploration and reliable commitment, we first design a *per-turn reward sequence* that rewards the discovery and commitment of gold code regions, providing learning signals from partial successes. Second, we propose an *action-level advantage estimation* scheme that groups turns sharing the same exploration context, isolating the credit of each action under the same context. Together, these two components enable action-aware RL that optimizes the agent at the action level.

We evaluate **A²AGENT** on two widely-used benchmarks, SWE-Bench Verified (Jimenez et al., 2024) and SWE-Bench Pro (Deng et al., 2025), both comprising real-world GitHub issues with their corresponding repositories. Our experimental results demonstrate that our **A²AGENT** outperforms the SOTA baselines by 1.58% and 8.55% in average F1, respectively. Notably, with Qwen3-4B as the backbone, **A²AGENT** surpasses baselines trained on models up to 8× larger, suggesting that fine-grained action-level supervision is a more critical performance factor than model scale in repository-level code localization.

2 Related Work

Repository-Level Code Localization. Early works in repository-level code localization rank files by similarity to the issue description using BM25 (Robertson and Zaragoza, 2009) or dense retrieval (Reddy et al., 2025; Suresh et al., 2025). However, the semantic gap between natural language issues and code, together with the difficulty of capturing inter-file dependencies (Gupta et al., 2025; Ouyang et al., 2025), motivates agent-based approaches that iteratively invoke tools to navigate the codebase. Subsequent work enriches these agents through graph modeling, symbol tracing, terminal navigation, and commit history (Chen et al., 2025; Zhang et al., 2025; Sutawika et al., 2026; Wang et al., 2025a; Jiang et al., 2025), while pipeline-based methods such as Agentless (Xia et al., 2024) take a complementary route by decomposing localization into sequential LLM prompts. Yet existing methods predominantly focus on expanding the search space and collecting relevant context, leaving two key capabilities unsolved: identifying which per-turn actions meaningfully contribute to issue resolution, and committing observed code elements to the final prediction.

Agentic Training. Early agentic training relied primarily on SFT to imitate teacher trajectories, but it struggles to generalize to unfamiliar codebases at inference time (Deng et al., 2025; Chu et al., 2025). Recent work has explored group-based RL methods such as GRPO (Shao et al., 2024) and GSPO (Zheng et al., 2025), which estimate a single trajectory-level advantage from rollout group reward statistics without a separate value function, and have been widely adopted across tasks including code localization (Zhang et al., 2025; Sutawika et al., 2026; Lee et al., 2026b). However, these methods assign uniform advantages across the entire trajectory based solely on the final prediction, making it difficult to attribute which actions contributed to the final outcome across multiple turns. Another line of work pursues finer-grained credit assignment, estimating advantages at intermediate steps via resampling (Kazemnejad et al., 2024; Lee et al., 2026a; Guo et al., 2026) or anchoring rollout groups to shared environment states (Feng et al., 2026). In code exploration, however, the repository remains fixed and multi-turn rollouts are costly to re-execute, leaving no cheap anchor for either approach. We tackle this by leveraging exploration history as a state proxy to compare tool calls within

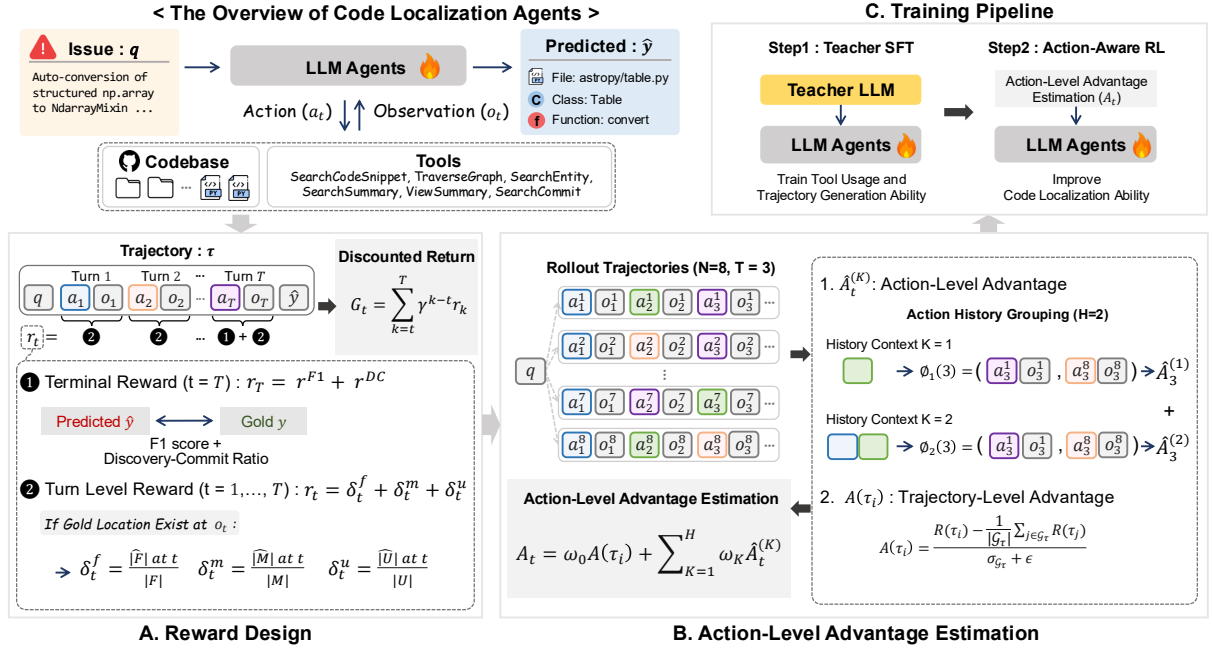


Figure 2: Overview of A^2 AGENT. Given an issue q , an LLM agent interacts with the codebase via tool calls to predict the gold target $\hat{y}$. (A) *Reward Design* combines a turn-level reward and a terminal reward to form the discounted return. (B) *Action-Level Advantage Estimation* groups turns sharing the same length- K action history across rollouts to compute a context-matched baseline, producing action-level advantages $\hat{A}_t^{(K)}$ which are then combined with the trajectory-level advantage $A(\tau_i)$. (C) *Training Pipeline* first fine-tunes the agent for tool-use ability, and then optimizes it via Action-Aware RL to improve localization performance.

the same search context, enabling action-level advantage estimation at each turn.

3 Preliminary

Problem Setup. We formulate repository-level code localization as a multi-turn decision-making process. At each turn t , the agent observes a state s_t encoding the issue, repository context, and interaction history, and produces an action $a_t \sim \pi_\theta(\cdot | s_t)$ from a policy π_θ . The agent outputs a localization decision at the final turn T , yielding a trajectory τ . In the conventional setting, a reward is observed only at this final turn, evaluating the final prediction. The discounted return from turn t is defined as $G_t = \sum_{k=t}^T \gamma^{k-t} r_k$, where $\gamma \in (0, 1]$.

Group-Relative Advantage Estimation. Group-based reinforcement learning methods, such as GRPO (Shao et al., 2024), estimate advantages without a separate value function by subtracting the mean return within a group of trajectories $\mathcal{G}_\tau$ sampled for the same issue:

$$A(\tau_i) = \frac{R(\tau_i) - \frac{1}{|\mathcal{G}_\tau|} \sum_{j \in \mathcal{G}_\tau} R(\tau_j)}{\sigma_{\mathcal{G}_\tau} + \epsilon}. \quad (1)$$

Since this value reflects only the relative performance of the entire trajectory, the same advantage is assigned to every turn, making it impossible to reflect the contribution of individual actions. To estimate action-level advantages, one must define the discounted return G_t at turn t and compare turns that took different actions under the same state $\tilde{s}$:

$$A(\tilde{s}, a) = \frac{G_t - \frac{1}{|\mathcal{G}_{\tilde{s}}|} \sum_{j \in \mathcal{G}_{\tilde{s}}} G_{t_j}}{\sigma_{\mathcal{G}_{\tilde{s}}} + \epsilon}. \quad (2)$$

4 Proposed Method

In this section, we introduce A^2 AGENT, an Action-Aware reinforcement learning method for repository-level code localization AGENTS. We first formulate the task (Section 4.1). Then, we present our reward design and action-level advantage estimation (Section 4.2), and describe the training procedure (Section 4.3). Figure 2 illustrates the overall framework.

4.1 Task Formulation

Given a repository and issue description q , our goal is to localize the code regions to be modified at three granularities, files $\hat{F}$, classes/modules $\hat{M}$, and functions $\hat{U}$, producing the predic-

tion $\hat{y} = (\hat{F}, \hat{M}, \hat{U})$. The agent operates over an action space $\mathcal{A} = \{\text{SearchCodeSnippet}, \text{TraverseGraph}, \text{SearchEntity}, \text{SearchSummary}, \text{ViewSummary}, \text{SearchCommit}\}$ consisting of six tools that support keyword search, structural traversal, and semantic retrieval over the repository. Detailed explanations are provided in Appendix A.

Following Section 3, at each turn t the policy π_θ generates a tool call $a_t \in \mathcal{A}$ from the current state s_t and receives an observation o_t , with s_t encoding the issue q and the interaction history $(a_1, o_1, \dots, a_{t-1}, o_{t-1})$. Here, each action a_t is defined as a tool name paired with its arguments, so that calls to the same tool with different arguments constitute distinct actions. The process terminates with a Finish action at turn T , yielding a trajectory $\tau = (q, a_1, o_1, \dots, a_T, \hat{y})$. Given a gold target $y = (F, M, U)$, we train π_θ to maximize $\mathbb{E}_{\tau \sim \pi_\theta}[R(\tau; y)]$, where $R(\tau; y)$ decomposes into turn-level rewards used to compute action-level advantages for fine-grained credit assignment.

4.2 Reward Design and Action-Level Advantage Estimation

Localizing issue-relevant code regions requires sequential exploration, where each tool call shapes the search space for subsequent turns. However, trajectory-level signals alone face two limitations: they fail to capture partial successes where the agent uncovers the correct location but does not commit it to the final prediction, and Eq. (2) cannot be directly applied due to the lack of state equivalence in code exploration (Section 3). We tackle these by designing a turn-level reward that captures partial successes (Section 4.2.1) and an action-level advantage estimation scheme over a code-specific state proxy (Section 4.2.2).

4.2.1 Reward Design

In this section, we design rewards that operate at two complementary levels, a *terminal reward* at the final turn that evaluates the quality of the final prediction, and a *turn-level reward* that provides immediate feedback whenever the agent newly discovers a gold location during exploration. Together they form the per-turn reward sequence $\{r_t\}_{t=1}^T$, from which the discounted return $G_t = \sum_{k=t}^T \gamma^{k-t} r_k$ is computed for action-level advantage estimation.

Terminal Reward. The terminal reward is computed at the final turn T and reflects the quality of the agent’s final prediction. It consists of two complementary rewards: $r_T = r^{\text{F1}} + r^{\text{DC}}$.

F1 reward r^{F1} evaluates final prediction accuracy as the sum of F1 scores at the file, module, and function levels:

$$r^{\text{F1}} = \text{F1}_F + \text{F1}_M + \text{F1}_U \quad (3)$$

Discovery-Commit (DC) reward r^{DC} measures the fraction of explored gold locations that the agent commits to the final prediction. For each granularity level $g \in \{F, M, U\}$, we define the DC ratio as follows:

$$\text{DC}_g = \frac{|E_g \cap Y_g \cap O_g|}{|E_g \cap Y_g|} \quad (4)$$

where E_g , Y_g , and O_g denote the sets of explored, gold, and predicted locations at level g . Letting $\mathcal{I} = \{g \mid |E_g \cap Y_g| > 0\}$ be the set of valid levels, the DC reward is defined as follows:

$$r^{\text{DC}} = \begin{cases} \frac{1}{|\mathcal{I}|} \sum_{g \in \mathcal{I}} \text{DC}_g & \text{if } \mathcal{I} \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

Within the discounted return G_t , this terminal reward propagates the outcome quality back to earlier turns, encouraging actions that lead to both accurate final predictions and consistent commitment of discovered locations.

Turn-Level Reward. Relying solely on the terminal reward leaves no learning signal for failed trajectories ($G_t = 0$) even when gold locations were partially discovered, and underestimates the credit of early actions in successful trajectories due to discount-induced bias toward later turns. To tackle this, we additionally assign a turn-level reward that provides immediate feedback whenever the agent newly discovers a gold location.

Let δ_t^f , δ_t^m , δ_t^u denote the fraction of gold locations newly discovered at turn t at the file, module, and function levels. The turn-level reward is:

$$r_t = \delta_t^f + \delta_t^m + \delta_t^u \quad (6)$$

so that G_t reflects per-turn discovery and preserves learning signals for partially correct trajectories that would otherwise receive no gradient.

4.2.2 Action-Level Advantage Estimation

Using G_t , we now estimate an action-level advantage that isolates the contribution of each individual action. As discussed in Section 3, this requires identifying turns that took different actions under the same state $\tilde{s}$, and computing the advantage as the discounted return G_t normalized against the

mean return within such a group (Eq. (2)). The challenge is that code exploration lacks a natural notion of state equivalence: unlike game or web environments where the state changes through observable transitions such as movement or page navigation, the repository remains fixed throughout the trajectory, so a turn’s state is determined solely by its accumulated interaction history. Determining whether two turns share the same state $\tilde{s}$ is therefore inherently difficult. We tackle this in two steps. We first define a state proxy for code exploration and group turns that share the same proxy. We then instantiate Eq. (2) on the resulting proxy-induced groups to obtain the action-level advantage.

State Proxy and Action-History Grouping.

Treating two turns as sharing the same state $\tilde{s}$ based on their action alone is insufficient: the same action can carry different exploration contexts depending on when it is invoked within a trajectory.

In our training data, approximately 79.2% of turns involve a different preceding action, and the same action appears in an average of 3.29 distinct accumulated histories, indicating substantial contextual bias. We therefore define the state proxy based on the “exploration history preceding” the current turn. Specifically, the preceding K -action sequence at turn t is $\phi_K(t) = \langle a_{t-K}, a_{t-K+1}, \dots, a_{t-1} \rangle$ and turns sharing the same $\phi_K(t)$ are grouped into $\mathcal{G}_t^{(K)}$. Their mean return serves as a baseline for the average expected outcome at that exploration context. The current action a_t is excluded from the group key, since including it would restrict comparisons to turns with the same action and cancel out the learning signal.

Note that grouping is performed per issue over rollouts of the same instance, and all exploration tools are read-only deterministic functions over a fixed repository, so an action with the same tool and arguments always returns the same observation. Turns sharing the same action history therefore reach their current turn through identical observations, ensuring that only comparable exploration contexts are grouped together.

Action-Level Advantage Estimation. Within a group $\mathcal{G}_t^{(K)}$ of turns sharing the same exploration history ϕ_K , we instantiate Eq. (2) by normalizing the return G_t against the group baseline and standard deviation:

$$\hat{A}_t^{(K)} = \frac{G_t - \text{mean}_{j \in \mathcal{G}_t^{(K)}} G_{t_j}}{\sigma_{\mathcal{G}_t^{(K)}} + \epsilon}. \quad (7)$$

Since the baseline is computed over turns that share the same exploration context but took alternative actions, $\hat{A}_t^{(K)}$ directly measures how much the current action a_t outperformed its alternatives under the same context.

To avoid committing to a single depth and balance precision against stability, motivated by He et al. (2026), we combine advantages from multiple depths $K = 1, \dots, H$, setting $\hat{A}_t^{(K)} = 0$ whenever the group size is one to suppress unreliable advantages. For turns whose groups are insufficient at every depth, we further introduce a trajectory-level anchor: the group-relative advantage $A(\tau_i)$ from Eq. (1) guarantees a minimum learning signal regardless of group availability. Combining the two, the final turn-level advantage is defined as:

$$A_t = \omega_0 \cdot A(\tau_i) + \sum_{K=1}^H \omega_K \cdot \hat{A}_t^{(K)}, \quad (8)$$

where $\omega_K \geq 0$ and $\sum_{K=0}^H \omega_K = 1$. The anchor term $\omega_0 \cdot A(\tau_i)$ ensures every turn receives at least a trajectory-level advantage, while each depth- K term $\omega_K \cdot \hat{A}_t^{(K)}$ contributes action-level advantage only when sufficiently many turns share the same exploration context at that depth.

4.3 Two-Stage Training

Teacher SFT Warm-up. Directly applying reinforcement learning to a pretrained LLM is challenging, as the initial policy lacks both the tool-use proficiency and the repository exploration patterns required to produce valid trajectories. To address this, we perform a SFT prior to reinforcement learning. Specifically, we leverage a strong teacher model to collect high-quality trajectories, filter for successful ones, and fine-tune the pretrained LLM on the resulting dataset to obtain the initial policy $\pi_{\theta}^{\text{sft}}$.

Action-Aware Reinforcement Learning. Initialized from $\pi_{\theta}^{\text{sft}}$, we perform action-aware reinforcement learning using the advantage A_t defined in Section 4.2 to improve code localization. We first collect N trajectories per issue from the training set. To reinforce actions at high-advantage turns and down-weight those at low-advantage turns, we define an advantage-weighted maximum-likelihood objective (Peng et al., 2019) that assigns token weights proportional to A_t . The transformation from A_t to a positive token weight is as follows:

Method	LLM	File-level			Module-level			Function-level		
		F1	P	R	F1	P	R	F1	P	R
Closed-Source LLMs										
RepoSearcher	Claude-3.7-Sonnet	32.30	20.24	89.24	–	–	–	26.91	18.64	66.08
RepoNavigator	GPT-5-Chat	58.88	61.87	58.17	–	–	–	31.17	34.56	30.42
	Claude-3.7-Sonnet	73.01	75.95	72.26	–	–	–	31.72	34.43	31.03
	Claude-Sonnet-4.5	<u>79.94</u>	<u>81.92</u>	80.68	–	–	–	43.62	45.76	43.97
OpenHands-Bash	GPT-5	3.20	3.20	3.20	2.60	2.60	2.60	2.60	2.60	2.60
	Claude-Sonnet-4.5	0.80	0.80	0.80	0.40	0.40	0.40	0.40	0.40	0.40
OpenHands-Bash ^{rem}	GPT-5	78.18	79.25	80.80	<u>61.17</u>	<u>62.23</u>	<u>63.35</u>	<u>54.79</u>	<u>56.80</u>	56.53
	Claude-Sonnet-4.5	82.01	84.50	<u>82.86</u>	67.19	70.11	67.47	61.78	65.42	<u>61.99</u>
Open-Source LLMs										
CoSIL	Qwen2.5-32B	30.77	19.34	83.50	–	–	–	22.11	14.85	55.38
Agentless		35.38	25.60	78.93	–	–	–	27.33	24.07	40.97
LocAgent		44.18	34.18	79.39	–	–	–	21.48	16.29	46.79
OrcaLoca		58.11	59.51	59.57	–	–	–	28.72	25.59	39.14
RepoSearcher	Qwen2.5-7B	30.09	18.80	83.11	–	–	–	25.57	17.68	<u>62.38</u>
	Qwen2.5-32B	32.25	20.24	88.59	–	–	–	28.03	19.36	68.55
RepoNavigator	Qwen2.5-7B	51.63	53.83	50.62	–	–	–	27.49	30.34	26.69
	Qwen2.5-14B	58.90	58.97	61.60	–	–	–	29.23	30.08	31.02
	Qwen2.5-32B	67.75	70.76	67.29	–	–	–	34.09	37.19	33.71
OpenHands-Bash	Qwen3-1.7B	2.40	2.09	3.60	0.37	0.32	0.60	0.34	0.32	0.50
	Qwen3-4B-Instruct	49.73	49.69	53.34	19.32	19.86	20.15	13.27	14.17	13.74
	Qwen3-14B	43.13	36.49	71.20	22.86	20.40	33.04	16.08	14.51	23.58
	Qwen3-32B (Thinking)	62.91	59.87	73.63	34.69	33.85	39.46	23.99	24.22	26.86
	CODESCOUT-4B	68.52	71.53	67.74	45.97	<u>49.70</u>	44.97	36.78	<u>40.71</u>	35.72
	CODESCOUT-14B	68.57	<u>71.00</u>	68.69	50.88	53.71	50.88	<u>40.32</u>	43.74	40.27
A ² AGENT	Qwen3-4B-Instruct	<u>72.64</u>	70.88	76.76	46.94	42.25	<u>59.55</u>	38.59	35.50	48.98
	Qwen3-8B	73.31	70.14	80.23	<u>47.98</u>	42.56	63.42	41.01	37.15	54.44

Table 1: Comparison of different methods on file-level, module-level, and function-level on SWE-Bench Verified using three evaluation metrics, F1 score (F1), Precision (P), and Recall (R). The best result per metric is **bold-faced** and the second best is underlined within each block (closed- and open-source LLMs).

$$\begin{aligned}\tilde{A}_t &= \text{clip}(A_t, -\kappa, \kappa), \\ w_t &= \text{clip}\left(\exp(\beta \tilde{A}_t), w_{\min}, w_{\max}\right)\end{aligned}\quad (9)$$

The inner clip bounds outlier advantages and the outer clip keeps the resulting positive weight within a stable range. The training objective is defined as follows, where the token weight $w_i := w_{t_i}$ assigns each token i the weight of its corresponding turn t_i , and $\ell_i(\theta) = -\log \pi_\theta(y_i \mid y_{<i}, \tau_{<t_i})$ denotes the per-token cross-entropy of policy π_θ :

$$\mathcal{L}(\theta) = \mathbb{E}_{\tau \sim \mathcal{D}} \left[\frac{1}{|V(\tau)|} \sum_{i \in V(\tau)} w_i \cdot \ell_i(\theta) \right] \quad (10)$$

where $\mathcal{D}$ is the set of collected trajectories and $V(\tau)$ denotes the index set of all assistant tokens in trajectory τ . The overall training algorithm is summarized in Appendix B.

5 Experiments

5.1 Experimental Setup

Datasets. We construct 2,438 training instances from SWE-Gym (Pan et al., 2024), and evaluate on two held-out benchmarks: SWE-Bench Verified (Jimenez et al., 2024) (500 human-verified) and the more challenging SWE-Bench Pro (Deng et al., 2025) (266 instances) for generalization. We exclude any repository-level overlap between training and evaluation sets to prevent contamination. Further details are provided in Appendix C.

Baselines. We compare our method against several representative baselines using both open- and closed-source LLMs, including CoSIL (Jiang et al., 2025), Agentless (Xia et al., 2024), LocAgent (Chen et al., 2025), OrcaLoca (Yu et al., 2025), RepoSearcher (Ma et al., 2025), RepoNavigator (Zhang et al., 2025), and CodeScout (Sutawika

Method	LLM	File-level			Module-level			Function-level		
		F1	P	R	F1	P	R	F1	P	R
Closed-Source LLMs										
OpenHands-Bash	GPT-5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Claude-Sonnet-4.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OpenHands-Bash ^{rem}	GPT-5	61.18	69.10	62.06	42.20	52.86	39.63	35.86	48.91	32.65
	Claude-Sonnet-4.5	64.75	75.39	64.49	48.54	61.75	45.39	42.26	58.14	38.06
Open-Source LLMs										
RepoSearcher	Qwen2.5-32B	3.81	2.52	9.00	–	–	–	2.31	2.46	2.52
LocAgent		19.77	0.38	25.73	–	–	–	4.30	0.17	8.72
Agentless		20.07	13.89	43.07	–	–	–	7.98	7.31	11.08
CoSIL		20.95	14.03	48.87	–	–	–	7.67	6.00	14.03
RepoNavigator	Qwen2.5-7B	39.74	48.13	36.36	–	–	–	14.29	21.26	12.33
	Qwen2.5-14B	49.72	58.64	46.85	–	–	–	18.06	25.25	16.05
	Qwen2.5-32B	57.57	68.69	53.49	–	–	–	20.72	29.44	18.13
OpenHands-Bash	Qwen3-1.7B	0.73	0.72	1.13	0.00	0.00	0.00	0.00	0.00	0.00
	Qwen3-4B-Instruct	36.96	44.42	35.59	11.78	17.46	10.19	8.12	12.16	7.01
	Qwen3-14B	30.08	28.48	48.22	11.87	13.82	14.21	8.20	9.97	9.92
	Qwen3-32B (Thinking)	46.85	49.65	54.55	21.94	27.18	22.62	12.31	17.82	11.93
	CODESCOUT-4B	51.77	68.98	46.16	36.97	56.05	31.13	29.03	48.65	23.73
	CODESCOUT-14B	53.63	68.81	48.81	37.13	53.80	32.02	28.74	46.09	23.76
A ² AGENT	Qwen3-4B-Instruct	58.19	73.34	53.83	38.38	45.18	40.00	29.23	38.29	28.85
	Qwen3-8B	60.05	76.54	56.16	39.07	45.51	42.37	30.60	39.16	30.91

Table 2: Comparison of different methods on file-level, module-level, and function-level on SWE-Bench Pro using three evaluation metrics, F1 score (F1), Precision (P), and Recall (R). The best result per metric is **bold-faced** and the second best is underlined within each block (closed- and open-source LLMs).

et al., 2026). All baseline results are taken from the respective papers or subsequent work. To provide closed-source upper bounds, we additionally evaluate GPT-5 and Claude-Sonnet-4.5 via the OpenHands (Wang et al., 2025b) scaffold. Detailed descriptions are provided in Appendix D.

Metrics. Following Sutawika et al. (2026), we predict a variable number of locations per issue rather than a fixed-length candidate list, allowing the model to adaptively determine the proper scope of each prediction. Under this setting, our primary evaluation metric is the instance-level average *F1 score* between predicted and ground-truth locations across three granularities of file, module, and function. We additionally report *precision* and *recall* at each granularity to enable fair comparison with baselines that predict differing numbers of locations. The formal definition of the evaluation protocol is provided in Appendix E.

Implementation Details. We train and evaluate with Qwen3-4B-Instruct and Qwen3-8B as base models, fine-tuned with LoRA (rank=16) for 3 epochs. For RL, we run 2 epochs with a batch size of 16, sampling $N=8$ trajectories per issue and

setting the history depth to $H=2$. Further implementation details are provided in Appendix F.

5.2 Main Results

As reported in Table 1, A²AGENT consistently outperforms all open-source baselines in average F1 over SWE-Bench Verified, surpassing the SOTA baseline (CodeScout-14B) by 1.58%. Notably, our 4B model even outperforms baselines up to $8\times$ larger (32B-scale models). Here, OpenHands-Bash^{rem} denotes an OpenHands-Bash variant with patch-editing removed, restricting it to localization. Trajectory-level training methods such as RepoNavigator and CodeScout assign a single advantage to every turn, capturing only the overall trajectory quality and leaving each individual tool call’s contribution unattributed. In contrast, A²AGENT rewards each call by comparing it against alternative actions taken under the same exploration history, converting a sparse reward into a dense action-level reward. This enables even a small model to learn targeted search over redundant broad search and to commit more of the gold locations it uncovers.

Furthermore, as shown in Table 2, A²AGENT also consistently outperforms all open-source base-

Method	History Grouping	r^{DC}	File F1	Module F1	Function F1
A ² AGENT	✓	✓	73.31	47.98	41.01
	✓	✗	71.73 (-1.58)	46.42 (-1.56)	39.55 (-1.46)
	✗	✓	72.19 (-1.12)	47.10 (-0.88)	40.19 (-0.82)
	✗	✗	71.01 (-2.30)	45.63 (-2.35)	38.72 (-2.29)
SFT	-	-	66.20 (-7.11)	41.15 (-6.83)	35.41 (-5.60)
GRPO	-	-	67.47 (-5.84)	42.33 (-5.65)	34.28 (-6.73)

Table 3: Ablation study on key components of A²AGENT. All ablations are conducted under the Qwen3-8B backbone on SWE-Bench Verified. Values in parentheses indicate the drop in percentage points relative to ours.

lines across all three granularities on SWE-Bench Pro, demonstrating robust generalization to diverse repositories and varying levels of task difficulty. We also provide qualitative results in Appendix G.

5.3 Ablation Study

In Table 3, we conduct an ablation study to verify the effectiveness of our method and the contribution of each proposed component, comparing ours against SFT and GRPO baselines while ablating its two key components, History Grouping ($H=2$) and Discovery-Commit reward (r^{DC}).

First, SFT generalizes poorly to unseen codebases (7.11%p drop in file-level F1), while GRPO assigns the same advantage to every turn, leaving individual tool call contributions unattributed and treating partially successful trajectories identically to complete failures. In contrast, even the simplest variant of A²AGENT (both components removed) already outperforms GRPO by 3.54%p, demonstrating that turn-level reward alone provides substantially richer learning signals than trajectory-level supervision.

Second, both components contribute complementarily. History Grouping and r^{DC} each contribute independently and combine for the best result, as they target distinct failure modes: History Grouping refines action-level credit by comparing turns under matched exploration contexts, while r^{DC} explicitly rewards the agent for committing the gold elements it observes during exploration.

Finally, these contributions hold consistently across all three granularities. Their effect does not diminish at finer levels. While the file level has a small candidate space that even coarse exploration

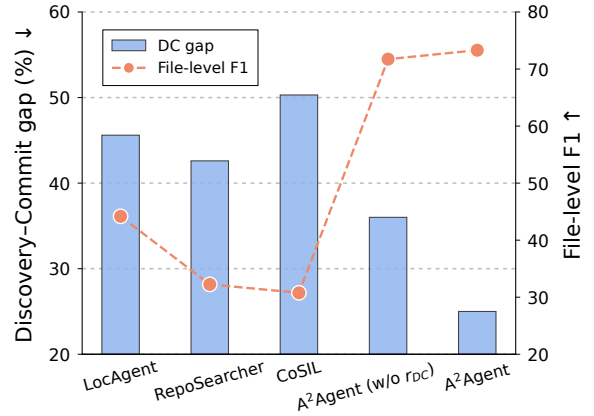


Figure 3: Comparison of Discovery-Commit gap and file-level F1 under Qwen3-8B backbone. r^{DC} markedly reduces the gap while keeping the highest F1.

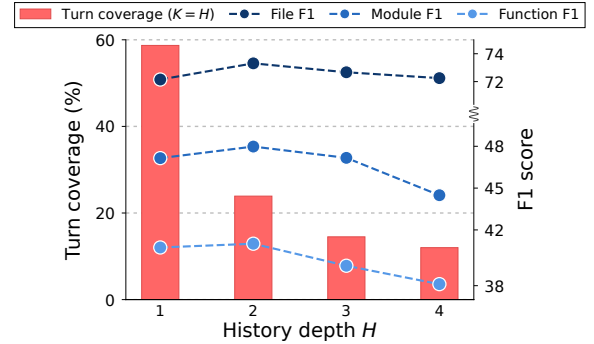


Figure 4: Effect of history depth H on F1 and turn coverage under Qwen3-8B backbone, the fraction of turns in groups of size ≥ 2 at depth $K=H$. F1 peaks at $H=2$ and declines as coverage shrinks with larger H .

can reach, the function level relies more heavily on committing discovered locations and isolating per-action credit, which is precisely where our components operate.

5.4 Further Analysis of A²AGENT

To further analyze the effectiveness of our method, we measure the *Discovery-Commit gap* across baselines in Figure 3, defined as the fraction of failures in which a gold location was discovered during exploration but not committed to the final prediction. This gap accounts for over 40% of failures across all baselines, revealing uncommitted discovery as a prevalent failure mode under trajectory-level supervision. In contrast, ours with r^{DC} substantially reduces this gap while simultaneously improving file-level F1, confirming that r^{DC} converts explored gold locations into correct predictions rather than trading localization accuracy for commitment.

To assess the effect of history depth H , we mea-

Type	Method	Avg. #Turns ($\downarrow$)	Avg. Time ($\downarrow$)
Procedure-based	Agentless	2.77	6.41
Agent-based	CoSIL	17.55	77.57
	LocAgent	6.20	86.23
	RepoSearcher	12.47	42.58
	A ² AGENT	11.72	18.08

Table 4: Inference efficiency comparison on SWE-Bench Verified, where all methods use the Qwen3-4B-Instruct backbone for fair comparison. **Avg. #Turns**: average number of tool calls per instance. **Avg. Time**: average inference time per instance.

sure F1 and turn coverage (the fraction of turns in groups of size ≥ 2 at depth $K=H$) across H values in Figure 4. Increasing H trades coverage for precision: smaller H groups more turns under a coarser context, while larger H matches turns under a finer context but leaves fewer turns grouped. Beyond $H=2$, coverage drops steeply and groups become too small for reliable credit estimation. The added deep-level terms thus contribute more noise than signal to Eq. (8), causing F1 to decline. We therefore set $H=2$, which best balances grouping precision and coverage.

5.5 Inference Efficiency Analysis

To assess the practical cost of A²AGENT, we report inference efficiency in Table 4. Among agent-based methods, A²AGENT attains the lowest inference time, which stems not from fewer tool calls but from more concise observations. Without action-level credit, baselines repeatedly invoke broad exploration tools that contribute little to localization. In contrast, A²AGENT learns to shift from broad exploration to targeted search, reducing average observation length by approximately 40% compared to baselines. Since observations accumulate in the prefill context of later turns, this gap compounds into a substantial speedup.

5.6 Influence on Issue Resolution

To verify that our localization gains carry over to end-to-end issue resolution, we conduct a downstream experiment that measures resolve rate directly, as reported in Table 5. We isolate the effect of localization by fixing the repair back-end to the Agentless repair pipeline. Every method feeds its localization output into this identical pipeline, performing localization with Qwen3-4B-Instruct and patch generation with Qwen3-30B-A3B-Instruct,

Method	File F1	Function F1	Resolved (%)
Oracle	-	-	45.93
RepoSearcher	29.33	18.02	26.82
CoSIL	32.70	19.37	27.64
Agentless	44.70	20.54	29.07
LocAgent	49.14	25.82	31.78
A ² AGENT	72.64	38.59	38.07

Table 5: Downstream issue resolution results on SWE-Bench Verified. All localization methods use the Qwen3-4B-Instruct backbone, and their outputs are fed into the same repair pipeline. Oracle denotes providing the gold locations directly to the repair pipeline.

so any difference in resolve rate reflects localization quality alone. Oracle feeds the gold locations directly into the repair pipeline and thus marks the upper bound reachable with perfect localization. Resolve rate rises together with localization quality across all methods, and A²AGENT reaches the highest resolve rate among all baselines while substantially narrowing the gap to the Oracle upper bound. This confirms that the F1 gains from A²AGENT translate into a clear improvement in end-to-end resolve rate, consistent with the widely recognized view that localization quality directly bounds final resolution performance (Jimenez et al., 2024; Xia et al., 2024).

6 Conclusion

We proposed A²AGENT, an action-aware reinforcement learning method for repository-level code localization agents. By estimating action-level advantages and incorporating a Discovery-Commit reward, the agent learns which tool calls advance localization and reliably commits discovered gold locations. Extensive experiments demonstrated that A²AGENT achieved strong performance, outperforming baselines up to $8\times$ larger. Requiring no external API access or Docker setup, A²AGENT can be deployed entirely on-premises, making it practical where source code privacy is required.

Limitations

Owing to computational resource constraints, our experiments were conducted on small open-source models rather than larger-scale ones. Even so, our method achieved strong performance at this scale. Moreover, although our method was also effective for downstream issue resolution as shown in Table 5, further optimization for the downstream task remains a promising direction for our future work.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (NO.RS-2025-02218768, 20%; No.RS-2025-25442569, 20%; RS-2019-II190421, 20%; IITP-2026-RS-2024-00437633, 20%), and the National Research Foundation of Korea(NRF) funded by the Ministry of Education (RS-2025-25433088, 20%).

References

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727.
- Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. 2025. Sft memorizes, rl generalizes: A comparative study of foundation model post-training. *arXiv preprint arXiv:2501.17161*.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.
- Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2026. Group-in-group policy optimization for llm agent training. *Advances in Neural Information Processing Systems*, 38:46375–46408.
- Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, and 1 others. 2023. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Yiran Guo, Lijie Xu, Jie Liu, Dan Ye, and Shuang Qiu. 2026. Segment policy optimization: Effective segment-level credit assignment in rl for large language models. *Advances in Neural Information Processing Systems*, 38:114399–114431.
- Dhruv Gupta, Gayathri Ganesh Lakshmy, and Yiqing Xie. 2025. Sac1: Understanding and combating textual bias in code retrieval with semantic-augmented reranking and localization. *arXiv preprint arXiv:2506.20081*.
- Shuo He, Lang Feng, Qi Wei, Xin Cheng, Lei Feng, and Bo An. 2026. Hierarchy-of-groups policy optimization for long-horizon agentic tasks. *arXiv preprint arXiv:2602.22817*.
- 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. *arXiv preprint arXiv:2106.09685*.
- Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Cosil: Software issue localization via llm-driven code repository graph searching. *arXiv preprint arXiv:2503.22424*.
- 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*.
- Amirhossein Kazemnejad, Milad Aghajohari, Eva Portelance, Alessandro Sordani, Siva Reddy, Aaron Courville, and Nicolas Le Roux. 2024. Vineppo: Refining credit assignment in rl training of llms. *arXiv preprint arXiv:2410.01679*.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Jaehoon Lee, CheolWon Na, Suyoung Bae, Jin-Seop Lee, Jihyung Lee, YunSeok Choi, and Jee-Hyong Lee. 2026a. [EXPO-SQL: Execution-based clause-level policy optimization for text-to-SQL](#). In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 22000–22019, San Diego, California, United States. Association for Computational Linguistics.
- Jin-Seop Lee, SungJoon Lee, SeongJun Jung, Boyang Li, and Jee-Hyong Lee. 2026b. Learning to refuse: Refusal-aware reinforcement fine-tuning for hard-irrelevant queries in video temporal grounding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10397–10407.

- Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. 2025. Tool-integrated reinforcement learning for repo deep search. *arXiv preprint arXiv:2508.03012*.
- Ashvin Nair, Abhishek Gupta, Murtaza Dalal, and Sergey Levine. 2020. Awac: Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*.
- Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. Repograph: Enhancing ai software engineering with repository-level code graph. In *International Conference on Learning Representations*, volume 2025, pages 30098–30121.
- Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.
- Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*.
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–16. IEEE.
- Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2025. Swerank: Software issue localization with code ranking. *arXiv preprint arXiv:2505.07849*.
- Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*, volume 4. Now Publishers Inc.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.
- Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. Cornstack: High-quality contrastive data for better code retrieval and reranking. In *International Conference on Learning Representations*, volume 2025, pages 15800–15817.
- Lintang Sutawika, Aditya Bharat Soni, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, Graham Neubig, and 1 others. 2026. Codescout: An effective recipe for reinforcement learning of code search agents. *arXiv preprint arXiv:2603.17829*.
- Boshi Wang, Weijian Xu, Yunsheng Li, Mei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. 2025a. Improving code localization with repository memory. *arXiv preprint arXiv:2510.01003*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025b. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits its reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2026. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.
- Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350*.
- Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Yang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, and 1 others. 2025. One tool is enough: Reinforcement learning for repository-level llm agents. *arXiv preprint arXiv:2512.20957*.
- Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, and 1 others. 2025. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*.

A Tool Specifications

In this section, we describe the six repository tools comprising the action space $\mathcal{A}$ introduced in Section 4.1. The tools are built on a repository graph index, a BM25-based code search index, a semantic file summary index, and a commit history index. We directly adopt the tool infrastructure from prior work (Chen et al., 2025; Wang et al., 2025a), ensuring that any performance gains observed in our experiments are attributable solely to our learning signal design rather than differences in tool design.

TraverseGraph(start, depth, edges). Returns a subgraph rooted at the given entity, traversing up to depth hops along the specified dependency types (contains, imports, invokes, inherits). Serves as the primary entry point for understanding the module hierarchy and dependency structure of an unfamiliar repository.

SearchCodeSnippet(query). Searches the entire codebase via a BM25 index using keywords or identifiers, returning the file path, line range, and a snippet of the top matching results. Most effective when exact identifiers can be extracted from the issue description and used to narrow down candidates for subsequent SearchEntity calls.

SearchEntity(entity). Returns a path of the form file_path, file_path:Class, or file_path:Class.method, and returns the raw source code of the specified entity. Used to directly inspect the logic of a narrowed candidate location and confirm a function-level answer.

SearchSummary(query). Searches file summaries generated by an LLM by semantic similarity against a natural language query, returning the top file paths and their summaries. Useful for narrowing the file candidate space when the issue is described in domain-level language without explicit identifiers.

ViewSummary(file). Returns a file summary generated by an LLM, covering its key responsibilities, major classes and functions, and dependencies. Enables rapid understanding of a file’s role without reading its full source.

SearchCommit(query). Returns the SHA, message, date, and list of changed files for past commits matching the given query. Provides context from historically similar changes and co-modified file

Hyperparameter	Value
Rollouts per issue N	8
Max tool calls per trajectory	20
History depth H	2
Discount factor γ	0.9
Batch size	16
Depth weights ω_K	$\propto K+1$
Advantage clip κ	5.0
Token-weight temperature β	1.5
Token-weight bounds $w_{\min}, w_{\max}$	0.1, 3.0
LoRA rank, α	16, 16
Learning rate (SFT)	2×10^{-4}
Learning rate (advantage-weighted RL)	5×10^{-5}
Learning rate schedule	Linear w/ warmup
Optimizer	AdamW
Weight decay	0.01

Table 6: Hyperparameters used for A²AGENT.

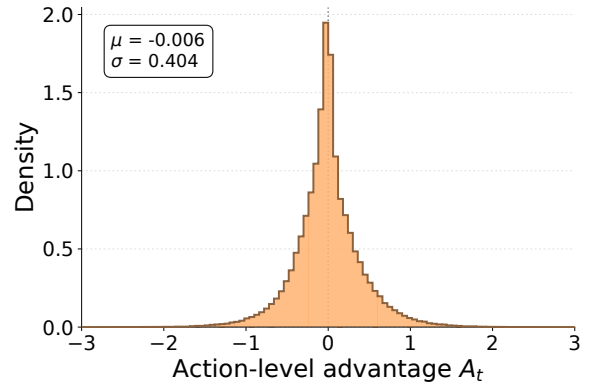


Figure 5: Distribution of per-turn action-level advantage A_t computed with history depth $H=2$ on the training set. The distribution is approximately zero-centered ($\mu = -0.006$, $\sigma = 0.404$), indicating that action-history grouping yields a stable and unbiased baseline for action-level credit assignment.

dependencies, supplementing static analysis with dynamic repository evolution signals.

B Algorithm

Algorithm 1 summarizes the training procedure of our method. We further report the distribution of action-level advantage A_t in Figure 5, showing that the estimated advantages are well-centered and stable across turns.

C Benchmark Details

SWE-Bench (Jimenez et al., 2024) is a benchmark for evaluating LLMs on real-world software engineering tasks, where each instance consists of a GitHub issue and its corresponding pull request from popular open-source Python repositories. For our localization task, gold targets at file, module,

Method	Tool	Input	Output
Agentless (Xia et al., 2024)	none (fixed pipeline)	problem description, file tree	ranked file/class/function list
CoSIL (Jiang et al., 2025)	symbol-trace	symbol name	call-graph slice rooted at symbol
OrcaLoca (Yu et al., 2025)	search_file_contents	file path	file body or skeleton
	search_class	class name	class definition
	search_method_in_class	class, method name	method body
	search_callable	callable name	callable code snippet
	search_source_code	code string	matching snippets
LocAgent (Chen et al., 2025)	SearchEntity	entity name	matching entity records
	TraverseGraph	start node, direction, depth	graph slice
	RetrieveEntity	entity ID	entity code block
	FuzzySearch	fuzzy query	candidate entities
RepoSearcher (Ma et al., 2025)	GetRepoStructure	repository root	directory tree
	SearchClass	class name	class body
	SearchFunction	function name	function body
	SearchClassMethod	class.method name	method body
RepoNavigator (Zhang et al., 2025)	Jump	symbol name in a file	symbol definition
CodeScout (Sutawika et al., 2026)	Terminal	Unix shell command	command stdout/stderr
	LocalizationFinish	{file, module, function} sets	final structured answer
A ² AGENT	SearchEntity	entity name	entity source code
	TraverseGraph	start node, direction, depth	graph slice
	SearchCodeSnippet	keyword or identifier	matching code snippets
	SearchSummary	natural language query	file paths and summaries
	ViewSummary	file path	natural language file summary
	SearchCommit	keyword	matching commit history

Table 7: Descriptions of the tool sets of code localization agents. A²AGENT adopts the same tool infrastructure as existing methods, focusing on training methodology rather than novel tool design.

and function granularities are extracted by parsing files, classes, and functions in the gold patch.

SWE-Bench Verified (Jimenez et al., 2024). A human-validated subset of 500 instances from SWE-Bench, created in collaboration with OpenAI to improve reliability. It serves as the de-facto standard for repository-level issue resolution.

SWE-Bench Pro (Deng et al., 2025). A harder benchmark sourced from 41 GPL-licensed and commercial repositories to mitigate data contamination, featuring long-horizon tasks with an average of 107.4 lines of edits across 4.1 files per instance.

D Baseline Methods

We describe the baseline methods used for comparison. The tool configurations of each method are summarized in Table 7.

Agentless (Xia et al., 2024). A fixed-pipeline method that sequentially executes three stages via LLM prompting, namely suspicious file candidate

extraction, class and function identification, and line-level localization, relying solely on in-context reasoning without an agent loop.

CoSIL (Jiang et al., 2025). Incrementally constructs a module call graph through function call analysis during exploration, expanding from an entry point toward suspicious regions. Employs context pruning for token budget management.

OrcaLoca (Yu et al., 2025). Navigates the repository via priority-based action scheduling, progressively decomposing file- and class-level search into function-level candidates with relevance scores. Uses distance-aware context pruning on the code graph to efficiently narrow down bug locations.

LocAgent (Chen et al., 2025). Indexes the repository as a heterogeneous graph prior to inference, enabling the agent to search, traverse, and inspect code entities on the graph. Our tool infrastructure directly follows this method.

Algorithm 1 Training Procedure of A²AGENT

Require: SFT policy $\pi_{\theta}^{\text{sft}}$, training queries $\mathcal{Q}$, rollouts per issue N , history depth H , discount factor γ , advantage weights $\omega_{0:H}$

```
1:  $\theta \leftarrow \theta^{\text{sft}}$ 
2: repeat
3:    $\mathcal{D} \leftarrow \emptyset$ 
4:   for each  $q \in \mathcal{Q}$  do
5:     Sample  $\{\tau^{(i)}\}_{i=1}^N \sim \pi_{\theta}(\cdot | q)$ 
6:     for all  $\tau^{(i)}$  do
7:        $r_t \leftarrow \delta_t^f + \delta_t^m + \delta_t^u \quad \forall t < T_i$ 
8:        $r_{T_i} \leftarrow r^{\text{F1}} + r^{\text{DC}}$ 
9:        $G_{T_i+1} \leftarrow 0$ 
10:      for  $t = T_i$  down to 1 do
11:         $G_t \leftarrow r_t + \gamma G_{t+1}$ 
12:      end for
13:    end for
14:     $A(\tau^{(i)}) \leftarrow \frac{R(\tau^{(i)}) - \text{mean}_j R(\tau^{(j)})}{\sigma_{\mathcal{G}_\tau} + \epsilon}$ 
15:    for all turn  $t$  in each  $\tau^{(i)}$  do
16:      for  $K = 1$  to  $H$  do
17:         $\mathcal{G}_t^{(K)} \leftarrow$  turns sharing  $\phi_K(t)$ 
18:         $\hat{A}_t^{(K)} \leftarrow \frac{G_t - \text{mean}_{j \in \mathcal{G}_t^{(K)}} G_{t_j}}{\sigma_{\mathcal{G}_t^{(K)}} + \epsilon}$ 
19:      end for
20:       $A_t \leftarrow \omega_0 A(\tau^{(i)}) + \sum_{K=1}^H \omega_K \hat{A}_t^{(K)}$ 
21:       $w_t \leftarrow f(A_t)$   $\triangleright$  Eq. (9)
22:    end for
23:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\tau^{(i)}, \{w_t\})\}_{i=1}^N$ 
24:  end for
25:   $\theta \leftarrow \arg \min_{\theta} \mathcal{L}(\theta; \mathcal{D})$ 
26: until maximum iterations reached
27: return  $\theta$ 
```

RepoSearcher (Ma et al., 2025). Adopts a two-stage workflow that progressively narrows from file-level to function-level localization, and is the first to introduce a distillation-based training framework (*ToolTrain*) for code localization agents using Claude 3.7 Sonnet as the teacher model.

RepoNavigator (Zhang et al., 2025). Equips the agent with a single Jump tool to trace Python symbol definitions, and trains directly from a pretrained model via GRPO without relying on any closed-source teacher model.

CodeScout (Sutawika et al., 2026). Adopts a minimal toolset consisting of a standard Unix terminal and LocalizationFinish, and trains the policy with reinforcement learning using GSPO.

E Evaluation Protocol

We formally define the evaluation protocol described in Section 5.1. Let $\hat{Y}_g$ denote the set of predicted locations and Y_g the set of gold locations at granularity $g \in \{F, M, U\}$. For each instance we compute the scores from the overlap of the two

sets and then average them over all instances.

$$\text{Recall} = \frac{|\hat{Y}_g \cap Y_g|}{|Y_g|}, \quad \text{Precision} = \frac{|\hat{Y}_g \cap Y_g|}{|\hat{Y}_g|},$$
$$\text{F1} = \frac{2|\hat{Y}_g \cap Y_g|}{|\hat{Y}_g| + |Y_g|} \quad (11)$$

Importantly, the denominator of precision is not the set of all functions in the repository but the candidate set $|\hat{Y}_g|$ that the model itself submits, and the remaining entities in the repository never enter the score computation. There is therefore no negative class, and the notion of a positive-to-negative ratio is not defined under this protocol.

F Implementation Details

All training stages are conducted on 4 NVIDIA RTX Pro 6000 GPUs (96 GB VRAM each). All experiments are implemented using the PyTorch framework with Python v3.12, using HuggingFace Transformers with DeepSpeed ZeRO-3 (Rajbhandari et al., 2020). Rollout collection is served with vLLM (Kwon et al., 2023). Hyperparameters specific to our method, including history depth H , are validated through ablation studies in Section 5.3. Basic training hyperparameters such as learning rate, batch size, and LoRA configuration are set to values commonly used in the literature (Hu et al., 2021; Shao et al., 2024). Detailed hyperparameters are provided in Table 6.

Teacher SFT Warm-up. Using a teacher model¹, we filter 1,410 trajectories that match at least one gold location at each granularity. Training uses a learning rate of 2×10^{-4} and sequence length of 36K. Loss is computed only on assistant turns.

Rollout Collection. Trajectories are sampled with temperature 1.0, top- p 0.8, and repetition penalty 1.05. The maximum number of tool calls per trajectory is capped at 20, and the maximum context length is 40K. All evaluation results are obtained from a single run with greedy decoding, consistent with the evaluation protocol of prior work.

Advantage-Weighted Reinforcement Learning.

Token weights are derived from A_t as described in Section 4.3. Eq. (10) is not a clipped policy-gradient objective as in PPO or GRPO but a maximum-likelihood objective weighted by advantages derived from the environment reward, which

¹Qwen3-30B-A3B-Instruct-2507

Model	Stage	Wall-clock
Qwen3-4B-Instruct	Teacher SFT	2.1h
	Rollout + Advantage-weighted training	2d 6h
Qwen3-8B	Teacher SFT	2.6h
	Rollout + Advantage-weighted training	3d

Table 8: Training cost of A²AGENT measured in wall-clock time on 4 NVIDIA RTX Pro 6000 GPUs.

falls under advantage-weighted training (Peng et al., 2019). Our procedure iterates rollout collection and training updates, re-collecting rollouts from the current policy at each iteration (Gulcehre et al., 2023), and advantage-weighted training is known to be robust to the resulting mild off-policy drift without importance-ratio correction (Peng et al., 2019; Nair et al., 2020).

The token-weight temperature β and advantage-clip bound κ are tuned by grid search over $\beta \in \{0.5, 1.0, 1.5, 2.0\}$ and $\kappa \in \{3.0, 5.0, 10.0\}$, with $\beta=1.5$ and $\kappa=5.0$ selected by validation F1. The policy is updated for up to 3 iterations, each consisting of rollout collection followed by a training update. Each training update runs for 2 epochs with batch size 16 for 2,046 steps, a learning rate of 5×10^{-5} , sequence length 40K, and AdamW optimizer (weight decay 0.01, 10 warmup steps, linear schedule). LoRA configuration follows that of Teacher SFT Warm-up.

Training Cost. We report the wall-clock time on 4 NVIDIA RTX Pro 6000 GPUs for teacher SFT and for rollout collection with advantage-weighted training per iteration in Table 8.

Token Consumption. Since the number of tool calls does not fully reflect the prompt size and generation per call, we report the average input and output tokens per instance over the 500 SWE-Bench Verified instances in Table 9. Input tokens denote the amount the model reads as the prompt, and output tokens denote the amount the model generates. The input tokens of A²AGENT are comparable to those of the multi-turn exploration baselines, whereas the output tokens are among the smallest. This is because our method performs only the necessary actions and finishes exploration in relatively few turns, concentrating on a specific area in the later turns, which is consistent with the shift toward targeted search reported in Section 5.5.

Repository Index Construction. Our method builds a dependency graph and a BM25 index once per repository snapshot, taking 4.7GB with 1h56m

Method	4B Input	4B Output	8B Input	8B Output
LocAgent	77.5K	3.6K	178.6K	1.2K
CoSIL	62.3K	3.2K	136.1K	1.4K
RepoSearcher	108.7K	1.8K	107.9K	1.3K
Agentless	8.2K	0.3K	8.2K	0.2K
A ² AGENT	76.8K	0.9K	129.1K	0.9K

Table 9: Average input and output tokens per instance on SWE-Bench Verified under the 4B and 8B backbones.

and 2.4GB with 29m respectively over all 500 SWE-Bench Verified instances, and once built they are reused through inference. Such preprocessing is not unique to our method, as graph-based agents also parse repository structure in advance. Moreover, since our static index is read-only, all rollouts share the same index, requiring no execution environment or state restoration per rollout.

G Case Study

In this section, we provide case studies comparing A²AGENT and the model trained with only SFT on the same issue, sphinx-doc/sphinx#10449. Figures 6–7 illustrate the trajectory of A²AGENT, which successfully localizes the gold location through appropriate tool calls. In contrast, as shown in Figure 8, the model trained with only SFT fails to reach the correct location, as it is limited to imitating teacher trajectories and cannot adapt its search strategy to unseen codebases effectively. The system prompt used for SFT, rollout generation, and inference is unified and provided in Figures 9–11.

Issue: sphinx-doc/sphinx#10449

autodoc_typehints="description" makes autoclass render a spurious *Return type* for a class (e.g. class Square whose `__init__` returns None).

Gold: sphinx/ext/autodoc/typehints.py / module typehints / function merge_typehints

Action 1: SearchSummary(query="sphinx autoclass return type", top_k=5)

Observation 1:

Found 5 files matching 'sphinx autoclass return type':

doc/.../examples/helloworld.py	- custom "Hello World!" directive (HelloWorld)
sphinx/highlighting.py	- Pygments syntax highlighting (PygmentsBridge)
sphinx/domains/__init__.py	- domains: ObjType, Domain
sphinx/search/tr.py	- Turkish search stemming
sphinx/util/images.py	- image size / mimetype utils

(none is the autodoc typehints file)

Action 2: ViewSummary(file_path="sphinx/domains/__init__.py")

Observation 2:

Summary for sphinx/domains/__init__.py:
Defines the structure/behavior of domains (group directives & roles).
Key classes: ObjType (object types in a domain), Domain (manages object types, directives, roles, indices).

Action 3: TraverseGraph(start=["sphinx/domains/__init__.py"], direction="downstream", depth=2, dep=["imports", "invokes"])

Observation 3:

```
sphinx/domains/__init__.py
|-- imports -- sphinx/addnodes.py:pending_xref
|-- imports -- sphinx/roles.py:XRefRole
|-- imports -- sphinx/roles.py:XRefRole.result_nodes
|-- imports -- sphinx/util/typing.py
|             |-- imports -- sphinx/ext/autodoc/mock.py:ismock
|             |-- imports -- sphinx/ext/autodoc/mock.py:_MockObject
|-- imports -- sphinx/builders/__init__.py:Builder
|-- imports -- sphinx/builders/__init__.py:Builder.get_target_uri
|-- imports -- sphinx/environment/__init__.py:BuildEnvironment
|-- imports -- sphinx/locale/__init__.py
|-- imports -- sphinx/util/docutils.py:SphinxDirective
|-- imports -- sphinx/util/typing.py:RoleFunction
|-- imports -- sphinx/util/typing.py:TitleGetter
|-- imports -- sphinx/directives/__init__.py:ObjectDescription
|-- imports -- sphinx/directives/__init__.py:ObjectDescription.needs_arglist
... (16 of 36 lines omitted)
```

Figure 6: Example trajectory of A²AGENT, which successfully localizes the gold location. Continued in Figure 7.

Action 4: SearchCodeSnippet(search_terms=["autoclass", "autodoc_typehints"])

Observation 4:

```
[bm25] sphinx/ext/autosectionlabel.py:setup
      51 def setup(app: Sphinx) -> Dict[str, Any]:
      52     app.add_config_value('autosectionlabel_prefix_document', False, 'env')
[bm25] sphinx/ext/autodoc/mock.py:_make_subclass
      ... (~100 of 108 lines omitted)
```

Action 5: SearchEntity(entity_names=["sphinx/ext/autodoc/typehints.py:merge_typehints"])

Observation 5:

```
Found function sphinx/ext/autodoc/typehints.py:merge_typehints (exact match)
37 def merge_typehints(app, domain, objtype, contentnode) -> None:
38     if domain != 'py':
39         return
40     if app.config.autodoc_typehints not in ('both', 'description'): # <-- gold
41         return
42     try:
43         signature = cast(addnodes.desc_signature, contentnode.parent[0])
44         if signature['module']:
45             fullname = '.'.join([signature['module'], signature['fullname']])
46         else:
47             fullname = signature['fullname']
48     except KeyError:
49         return
50     annotations = app.env.temp_data.get('annotations', {})
51     if annotations.get(fullname, {}):
52         field_lists = [n for n in contentnode if isinstance(n, nodes.field_list)]
53         for field_list in field_lists:
54             if app.config.autodoc_typehints_description_target == "all": # <-- gold
55                 modify_field_list(field_list, annotations[fullname])
56             elif ... == "documented_params": # <-- gold
57                 augment_descriptions_with_types(field_list, ..., force_rtype=True)
```

Action 6: SearchCodeSnippet(search_terms=["autodoc_typehints_description_target"])

Observation 6:

```
[bm25] -> sphinx/ext/autodoc/typehints.py:merge_typehints (confirms target)
40 if app.config.autodoc_typehints not in ('both','description'): return
      ... (~90 of 98 lines omitted)
```

Finish

File: sphinx/ext/autodoc/typehints.py ✓

Module: typehints ✓

Function: merge_typehints ✓

Figure 7: Example trajectory of A²AGENT, which successfully localizes the gold location.

Issue: sphinx-doc/sphinx#10449

autodoc_typehints="description" makes autoclass render a spurious *Return type* for a class (e.g. class Square whose `__init__` returns None).

Gold: sphinx/ext/autodoc/typehints.py / module typehints / function merge_typehints

Action 1: TraverseGraph(start=['sphinx'], depth=2, dep=['contains'])

Observation 1:

```
sphinx
|-- contains -- sphinx/__init__.py
|-- contains -- sphinx/addnodes.py
|               |-- sphinx/addnodes.py:desc_signature
|               |-- sphinx/addnodes.py:desc_content
|               ... (~200 of 206 lines omitted; ext/autodoc/typehints.py not surfaced)
```

Action 2: SearchSummary(query="autodoc typehints")

Observation 2:

No files found matching 'autodoc typehints'.

Action 3: SearchSummary(query="autodoc")

Observation 3:

Found 7 files matching 'autodoc':

```
doc/conf.py
doc/.../examples/autodoc_intenum.py
sphinx/ext/autodoc/__init__.py
  Classes: Documenter, ModuleDocumenter, ClassDocumenter, ExceptionDocumenter, ...
  Functions: identity, members_option, ..., setup
sphinx/ext/autodoc/directive.py (AutodocDirective)
sphinx/ext/autodoc/importer.py (import_object, get_class_members)
sphinx/ext/autodoc/mock.py (_make_subclass, ismock)
sphinx/ext/autodoc/preserve_defaults.py
(typehints.py is NOT among the results)
```

Action 4: ViewSummary(file_path="sphinx/ext/autodoc/__init__.py")

Observation 4:

Summary for sphinx/ext/autodoc/__init__.py:

```
Classes: Documenter, ModuleDocumenter, ClassDocumenter, ExceptionDocumenter, ...
Functions: identity, members_option, ..., autodoc_attrgetter, setup
Imports: ..., sphinx/ext/autodoc/importer.py, .../mock.py | Edits: 5 commits
(merge_typehints / typehints.py never appears)
```

Action 5: SearchEntity(entity_names=["sphinx/ext/autodoc/__init__.py"])

Observation 5:

Found file sphinx/ext/autodoc/__init__.py (exact match)

```
1 """Extension to create automatic documentation from code docstrings.
8 import re
12 from typing import (TYPE_CHECKING, Any, Callable, Dict, Iterator, List, ...)
... (~200 of 205 lines omitted)
```

Finish

File: sphinx/ext/autodoc/__init__.py ✗

Module: autodoc ✗

Function: - ✗ (typehints.py not reached)

Figure 8: Example trajectory of the model trained with only SFT, which fails to localize the gold location.

System Prompt for Localization Agent (continued on next page)

Given the following GitHub problem description, your objective is to localize the *exact* code locations that need modification at THREE granularities:

1. *File* --- the source file containing the bug or new behavior.
2. *Module* --- the class (or top-level module-scope function) inside that file.
3. *Function/Entity* --- the specific method/function that must be edited.

You MUST report all three granularities in your final answer. Reporting only file paths is INSUFFICIENT --- module and function predictions are evaluated independently and contribute to your score.

Follow these steps to localize the issue:

Step 1: Categorize and Extract Key Problem Information

- Classify the problem statement into the following categories:
Problem description, error trace, code to reproduce the bug, and additional context.
- Identify candidate modules in the '{package_name}' package mentioned in each category.
- Use extracted keywords and line numbers to search for relevant code references for additional context.

Step 2: Locate Referenced Modules and Functions

- Accurately determine specific files -> modules -> functions.
- Explore the repository to familiarize yourself with its structure.
- Analyze the described execution flow to identify specific classes / methods being referenced.
- Inspect class hierarchies and method signatures to pin down the exact function.
- Pay special attention to distinguishing between modules with similar names using context and described execution flow.

Output format for each location:

- File only: 'file_path'
- Module (class): 'file_path:ClassName'
- Function (method/entity): 'file_path:ClassName.method_name' or 'file_path:top_level_function'

Example: for method `calculate\_sum` of class `MathUtils` in `src/helpers/math\_helpers.py`:

- file: src/helpers/math_helpers.py
- module: src/helpers/math_helpers.py:MathUtils
- entity: src/helpers/math_helpers.py:MathUtils.calculate_sum

Figure 9: System prompt provided to the localization agent.

System Prompt for Localization Agent (continued on next page)

```
## Step 3: Analyze and Reproduce the Problem
- Clarify the purpose of the issue.
- If expanding capabilities: identify where and how to incorporate new behavior, fields,
  or modules.
- If addressing unexpected behavior: focus on localizing modules containing potential
  bugs.

- Reconstruct the execution flow:
  - Identify main entry points triggering the issue.
  - Trace function calls, class interactions, and sequences of events.
  - Identify potential breakpoints causing the issue.
  - Keep the reconstructed flow focused on the problem and avoid irrelevant details.

## Step 4: Locate Areas for Modification (file -> module -> function)
- For every candidate file, ALSO determine WHICH class and WHICH method inside it
  require changes.
  Do not stop at the file level.
- Use `SearchEntity` to read the actual class/method body and verify the function-level
  location.
- Consider upstream and downstream dependencies that may affect or be affected by the
  issue.
- If applicable, identify where to introduce new fields, functions, or variables.
- Think thoroughly: list multiple potential solutions and consider edge cases that
  could impact the resolution.

## Available Tools and When to Use Them
You have 6 tools. Choose the most appropriate tool for each step. Do NOT repeatedly
call the same tool.

### Search Tools
- `TraverseGraph`: first step --- understand repo layout and module hierarchy.
- `SearchSummary`: semantic file discovery.
- `SearchCodeSnippet`: find code by keyword.
- `SearchCommit`: find related commits and changed files.

### View Tools
- `SearchEntity`: read actual code once you know the file/class/function path.
- `ViewSummary`: view the LLM-generated summary of a specific file.
```

Figure 10: System prompt provided to the localization agent.

System Prompt for Localization Agent

```
### Finish
- `Finish`: submit final answer after you have localized all relevant locations.

### Recommended workflow
1. `TraverseGraph` -> get repository overview
2. `SearchSummary` -> find relevant files by concept
3. `ViewSummary` -> understand key files' purposes
4. `SearchCommit` -> find related change history and co-changed files
5. `SearchCodeSnippet` -> locate specific code regions by keyword
6. `SearchEntity` -> read actual code of key functions/classes
7. `Finish` -> output your final answer

## Output Format for Final Results
Your final output should list the locations requiring modification, wrapped with triple
backticks.
For EACH location you MUST report all available granularities:
- the file path
- the class name (module-level), if the change is inside a class
- the function name (entity-level): either `ClassName.method` or top-level `
  function_name`
- line numbers when known

Order locations by importance.
Aim for about 5 files.
Within each file, list every relevant module/function, not just one.

Module-level and function-level predictions are scored independently from file-level.
Reporting only the file path will lose about 2/3 of the available credit.
Always include `class:` and `function:` when the answer lies inside a class or specific
function.

### Examples
full_path1/file1.py
line: 10
class: MyClass1
function: MyClass1.my_function1

full_path2/file2.py
line: 76
class: MyClass2
function: MyClass2.my_function2

full_path3/file3.py
line: 24
line: 156
function: my_function3

Return just the location(s).
```

Figure 11: System prompt provided to the localization agent.