CODESCOUT: AN EFFECTIVE RECIPE FOR REINFORCEMENT LEARNING OF CODE SEARCH AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A prerequisite for coding agents to perform tasks on large repositories is code localization - the identification of relevant files, classes, and functions to work on. While repository-level code localization has been performed using embedding-based retrieval approaches such as vector search, recent work has focused on developing agents to localize relevant code either as a standalone precursor to or interleaved with performing actual work. Most prior methods on agentic code search equip the agent with complex, specialized tools, such as repository graphs derived using static analysis. In this paper, we demonstrate that, with an effective reinforcement learning recipe, a coding agent equipped with *nothing more* than a standard Unix terminal can be trained to achieve strong results. Our experiments on three benchmarks (SWE-Bench Verified, Pro, and Lite) reveal that our models consistently achieve superior or competitive performance over **2-18**$\times$ larger base and post-trained LLMs and sometimes approach performance provided by closed models like Claude Sonnet, even when using specialized scaffolds. Furthermore, our models improve over their corresponding base models on programming languages not seen during RL, demonstrating cross-language generalization with a language-agnostic bash-only scaffold. Our work particularly focuses on techniques for re-purposing existing coding agent environments for code search, reward design, and RL optimization. We release the resulting model family, CODESCOUT, along with all our code and data for the community to build upon.

1 INTRODUCTION

For repository-level coding tasks such as those in the popular SWE-Bench benchmark (Jimenez et al., 2024), a critical first step is **code localization**: given an issue description and a codebase, the system must identify the relevant files and finer-grained code entities (e.g., classes and functions) to modify (§2; Husain et al. (2019); Xia et al. (2024)). This is challenging in large repositories with complex inter-dependencies, and relying on general-purpose language models to solve this localization problem as part of the agentic coding loop can result in high costs, incorrect fixes, and code bloat (Hong et al., 2025).

To address this issue, it is also common to incorporate some variety of specialized localization module to acquire the relevant context from large codebases in a more efficient and effective manner. Traditionally, this code search was performed through semantic code search using vector databases (Xia et al., 2024; Wang et al., 2025b). In contrast, recent methods have investigated *agentic code search* - using agents to iteratively navigate the repository and uncover necessary evidence for solving the issue under consideration. Typically, these methods have involved significant modifications to the agent itself to incorporate static analysis of the codebase, such as LocAgent's repository graph navigation (Chen et al., 2025) and RepoNavigator's "jump" tool that retrieves definitions of Python symbols (Zhang et al., 2025). While well-grounded, these necessitate the use of static analysis tools tailored to a *particular* programming language, increasing operational complexity of deploying such agents on a broader variety of coding scenarios. Simultaneously, there have been various anecdotal reports from industry regarding reinforcement learning methods that increase the ability of agents to perform localization rapidly (Pan et al., 2025a; Cursor Team, 2025), with varying levels of detail but without a clear recipe. Given this background, we address the fundamental research question: *given an appropriate reinforcement learning algorithm, what is an effective recipe to train a code localization agent that simply uses the terminal tool typical of more generic coding*

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.

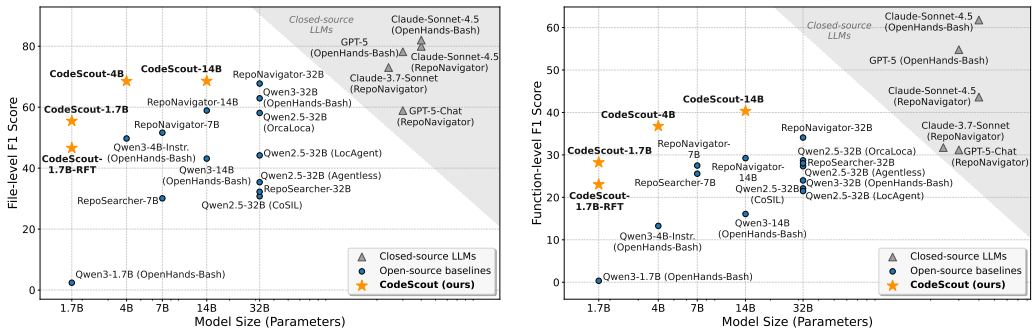


Figure 1: An overview of code localization performance of various approaches on SWE-Bench Verified. CODESCOUT achieves superior or competitive results over larger SoTA open-source LLMs and closes the gap with frontier closed-source LLMs.

agents, yet achieves competitive or superior accuracy over agents using special-purpose tools? This work provides the first demonstration of such a recipe, that involves scalable methods for data and environment creation (§3.1), a standard agent scaffold (§3.2), careful attention to reward design (§3.3) and training algorithm (§3.4). Our experiments (§4, §5) show that the resulting model, CODESCOUT, achieves state-of-the-art results on three localization datasets derived from SWE-Bench Verified, Pro, and Lite. Notably, all models and code are released publicly for future research to build upon.

2 RELATED WORK

Prior work proposes specialized agents and trains open-source LLMs for code localization (Table 1), but is often limited to a single programming language (typically Python) due to reliance on static analysis tools (e.g., AST parsers). Extending them to other languages requires additional engineering effort. LocAgent (Chen et al., 2025) and OrcaLoca (Yu et al., 2025) construct code graphs to capture hierarchical dependencies, requiring expensive pre-indexing. CoSIL (Jiang et al., 2025) dynamically builds module and function call graphs, and RepoSearcher (Ma et al., 2025) has specialized tools, for example to extract imports or search for methods in a class. While CoSIL and RepoSearcher avoid pre-indexing, they still require static analysis. RepoNavigator (Zhang et al., 2025) has a "jump" tool that resolves Python symbol definitions through a language server built using AST. Our agent *solely* uses a terminal, making it language-agnostic by design.

Table 1: Comparison of code localization methods. CODESCOUT is the only method that directly post-trains LLMs with RL using a programming language-agnostic scaffold equipped *solely* with a bash terminal.

Method	Language-Agnostic Scaffold	Pure RL post-training	# Tools
LocAgent (Chen et al., 2025)	✗	✗	3
CoSiL (Jiang et al., 2025)	✗	✗	3
OrcaLoca (Yu et al., 2025)	✗	✗	5
RepoSearcher (Ma et al., 2025)	✗	✗	5
RepoNavigator (Zhang et al., 2025)	✗	✓	1
CODESCOUT *(Ours)*	✓	✓	1

Most prior methods either do not train LLMs using their proposed agents, like CoSIL and OrcaLoca, or rely on supervised distillation from closed-source LLMs via rejection sampling fine-tuning (Yuan et al., 2023), like LocAgent and RepoSearcher. CODESCOUT trains LLMs directly with RL, removing dependence on expensive proprietary LLMs for data curation.

Finally, CODESCOUT has a *significantly simpler* scaffold in terms of engineering overhead and tool count. Since bash terminal is already a core component of standard coding agents (Wang et al., 2025a; Yang et al., 2024; Anthropic, 2025), we do not need to implement specialized task-specific tools or scaffolds, unlike prior work. Moreover, when comparing the size of the agent's action space, excluding any finish tools, CODESCOUT has only 1 tool, compared to 3-5 tools in prior work. Appendix D includes a detailed review of prior work.

3 CODESCOUT: AN EFFECTIVE RL RECIPE FOR CODE LOCALIZATION

This section presents the methodology used to train CODESCOUT, covering training data curation and environment construction (§3.1), the agent scaffold (§3.2), reward design (§3.3), and the RL training setup (§3.4). Figure 2 provides an overview of our approach.

3.1 DATA AND ENVIRONMENT CURATION

Given a GitHub issue $\mathcal{I}$ in a Python repository $\mathcal{R}$, we process the ground-truth issue resolution patch $\mathcal{P}$ to extract the localization targets at three granularities. Specifically, we define the ground-truth target as $y^\star = (F^\star, M^\star, U^\star)$, where $F^\star = f_1^\star, \ldots, f_{N_f}^\star$ is the set of modified files, $M^\star = m_1^\star, \ldots, m_{N_m}^\star$ is the set of modified modules, and $U^\star = u_1^\star, \ldots, u_{N_u}^\star$ is the set of functions or methods edited by the patch $\mathcal{P}$. An example of these targets is illustrated in Figure 2. The LLM agent is tasked with predicting $y^\star$ given the issue description $\mathcal{I}$ and the pre-PR repository state $\mathcal{R}$. We extract ground truth by using patch-processing scripts from Chen et al. (2025).

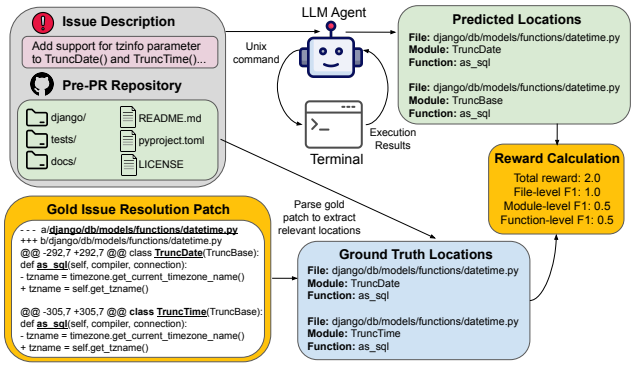


Figure 2: An overview of CODESCOUT: given an issue, the agent navigates the pre-PR codebase using a terminal and predicts the relevant files, modules, and functions. The reward function computes F1 scores for these three granularities using ground truth locations extracted from the gold issue resolution patch.

We curate training instances by processing GitHub issues collected by prior work (Yang et al., 2025b; Pan et al., 2025b), originally intended for training agents to fix issues. We discard issues with empty descriptions and whose PRs create or delete files, as the agent cannot predict the name of newly created files and we cannot determine relevant modules or functions for deleted files. We ignore non-Python files (e.g., `README.md`) because function and module-level information cannot be extracted from them. We construct the RL environment by cloning the pre-PR commit of the repository to a specific path known to the agent via its prompt. As localization does not require code execution, we do not install dependencies or use sandboxes. Notably, as our agent only uses a terminal, its environment setup overhead is much lower than prior agents that use code graphs, vector indices, or dependency parsers.

3.2 OPENHANDS-BASH: OUR AGENT SCAFFOLD

We use OpenHands Agent SDK (Wang et al., 2025a) to implement our agent as it performs strongly across software engineering benchmarks, implements core tools (e.g., terminal), supports parallel tool-calling, and has a modular design that easily integrates with our training backend. The agent is equipped with a `Terminal` tool that supports standard Unix commands (e.g., `find`, `ls`). We install `ripgrep` (Gallant, 2016) in the agent's environment, which is a command-line utility for fast grep-style search using regex patterns recursively through all files in a directory. The agent uses the `LocalizationFinish` tool to submit predictions. We refer to this scaffold as **OpenHands-Bash**.

Our initial experiments used a string-based output format from LocAgent (Chen et al., 2025), but we found training with this format is sensitive to noisy reward signals due to brittle validation and parsing. We address this by requiring the agent to terminate via the `LocalizationFinish` tool, enforcing a structured output schema and simplifying parsing, improving reward fidelity. Appendix A includes detailed prompts and tool definitions for our agent.

3.3 REWARD DESIGN

For each trajectory τ sampled from the agent, we extract the prediction $y = (F, M, U)$ from the `LocalizationFinish` tool, which specifies the predicted files, modules, and functions. Given

the ground-truth $y^\star = (F^\star, M^\star, U^\star)$, we compute F1 scores at each granularity. For each set $S \in \{F, M, U\}$ with the corresponding ground truth set $S^\star \in \{F^\star, M^\star, U^\star\}$, we compute F1 score (i.e. harmonic mean of precision and recall): $r^{\mathsf{F1-file}}$, $r^{\mathsf{F1-module}}$, and $r^{\mathsf{F1-func}}$. Our reward function is defined as:

$$r(\tau, y, y^\star) = r^{\mathsf{F1-file}}(y, y^\star) + r^{\mathsf{F1-module}}(y, y^\star) + r^{\mathsf{F1-func}}(y, y^\star) \tag{1}$$

Prior training runs for our 14B LLM collapsed to zero rewards in later stages as the agent frequently exhausted the step budget without answer submission, an error behavior we do not observe for the 1.7B and 4B LLMs. We fix this with an auxiliary binary reward $r^{\mathsf{turn}}(\tau, k)$ that assigns 1 if and only if the agent terminates in exactly k turns, where k is the step limit, and add this binary term $r(\tau, y, y^\star)$ in Equation 1. Appendix I includes more discussion on reward design.

3.4 RL Training Algorithm

We use SkyRL (Griggs et al., 2025) and train asynchronously (Fu et al., 2026) to improve GPU utilization by parallelizing rollouts and weight optimization and set maximum staleness to 4. We train models using Group Sequence Policy Optimization (GSPO) (Zheng et al., 2025). Given model parameters θ and input prompt x; i indexes sequences in a group of size G; y_i is the i^{th} output sequence with length $|y_i|$, and $y_{i,t}$ is its t^{th} token. π_θ and $\pi_{\theta_{\text{old}}}$ are current and older policies, $\hat{A}_i$ is the advantage, and ε is the clipping parameter:

$$\mathcal{J}_{\text{GSPO}}(\theta) = \mathbb{E}_i \left[\frac{1}{G} \sum_{i=1}^{G} \min \left(s_i(\theta)\hat{A}_i,\ \text{clip}\left(s_i(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_i \right) \right] \tag{2}$$

where, s_i is the importance ratio derived from sequence likelihood (Zheng et al., 2023):

$$s_i(\theta) = \exp\left(\frac{1}{|y_i|} \sum_{t=1}^{|y_i|} \log \frac{\pi_\theta(y_{i,t}|x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|x, y_{i,<t})} \right) \tag{3}$$

Following Liu et al. (2025), we remove standard deviation from the advantage calculation.

$$\hat{A}_i = r_i - \text{mean}(\mathbf{r}), \mathbf{r} = \{r_1, ..., r_G\} \tag{4}$$

Finally, we remove KL regularization from the loss, disable entropy loss, and mask loss for rollouts that exhaust maximum steps without submitting predictions.

4 Experimental Setup

4.1 Training Setup

We process the SWE-Smith (Yang et al., 2025b) dataset using the method from §3.1, resulting in a training set of 39K instances from 128 repositories. None of these repositories are present in our evaluation benchmarks (§4.2), to avoid dataset contamination. We train LLMs from Qwen3 series (Yang et al., 2025a): `Qwen3-1.7B`, `Qwen3-4B-Instruct-2507`, and `Qwen3-14B`. We disable thinking for 1.7B and 14B LLMs and mask loss for tokens not generated by the LLM like system and user prompts, and tool execution results.

CodeScout-4B and 14B are directly trained from their corresponding base models using the algorithm from §3.4. As Qwen3-1.7B performs very poorly (§5), we do not directly train it using RL but warm-start the model with rejection sampling fine-tuning (RFT) (Yuan et al., 2023) on 4K successful trajectories (i.e., F1 = 1.0 for file, module, and function; see §4.2) sampled from CodeScout-14B. The resulting model (CodeScout-1.7B-RFT) is further trained using RL (§3.4) on samples not used in RFT resulting in CodeScout-1.7B. Appendix B.1 provides more experimental details.

4.2 Evaluation Setup

We report results on SWE-Bench Verified (Chowdhury et al., 2024) (500 instances), SWE-Bench Lite (Jimenez et al., 2024) (300 instances), and the Python subset of SWE-Bench Pro (Deng et al., 2025) (266 instances), which is more challenging than the other two. We re-purpose the datasets to

Table 2: Summary of results on SWE-Bench Verified, Pro, and Lite. For all benchmarks, CODESCOUT achieves a new open-source SoTA, with stronger/comparable results over base and post-trained LLMs upto **8-18× larger**, narrowing the gap with and often surpassing closed-source LLMs. The highest metric is **bold-faced** and 2^{nd} highest metric is underlined.

Benchmark	Method	File-level F1	Function-level F1
SWE-Bench Verified	RepoNavigator-7B	51.63	27.49
	CODESCOUT-1.7B	55.46	28.22
	RepoNavigator-32B	67.75	34.09
	CODESCOUT-4B	68.52	36.78
	CODESCOUT-14B	68.57	40.32
	RepoNavigator + GPT-5-Chat	58.88	31.17
	RepoNavigator + Claude-Sonnet-4.5	**79.94**	**43.62**
SWE-Bench Pro	RepoNavigator-7B	39.74	14.29
	CODESCOUT-1.7B	40.96	18.24
	RepoNavigator-32B	**57.57**	20.72
	CODESCOUT-4B	51.77	**29.03**
	CODESCOUT-14B	53.63	28.74
SWE-Bench Lite	OpenHands-Bash + Qwen3-32B (Thinking)	58.98	23.76
	CODESCOUT-1.7B	56.57	27.07
	CODESCOUT-4B	67.03	39.87
	CODESCOUT-14B	**71.84**	**44.43**

extract ground-truth locations (§3.1). Our primary evaluation metric is the instance-wise average of F1 score between prediction and ground-truth for three granularities: file, module, and function. We also report average precision and recall for each granularity. We mainly focus on F1 score as it captures both precision and recall.

Baselines: We compare CODESCOUT against various baselines using both open-source and closed-source LLMs: RepoNavigator (Zhang et al., 2025), RepoSearcher (Ma et al., 2025), LocAgent (Chen et al., 2025), OrcaLoca (Yu et al., 2025), CoSIL (Jiang et al., 2025), and Agentless (Xia et al., 2024). All baseline metrics are taken from prior work. While Ma et al. (2025); Jiang et al. (2025); Chen et al. (2025); Xia et al. (2024) output a ranked list of top-K locations (typically K is fixed and set to 5), Zhang et al. (2025); Yu et al. (2025) and CODESCOUT dynamically predict a variable number of locations, which is a better design as issues may require editing varying number of locations. Thus, some baselines have precision-recall disparities (see Appendix K). We evaluate other LLMs with OpenHands-Bash (§3.2): the base variants of CODESCOUT (`Qwen3-4B-Instruct-2507`, and the non-thinking versions of `Qwen3-1.7B` and `Qwen3-14B`), and `Qwen3-32B` with thinking enabled to compare with a larger reasoning LLM. We also evaluate GPT-5 and Claude-Sonnet-4.5. Despite specifying the turn limit in the system prompt, these closed-source LLMs frequently exhaust turn limits without answer submission. Similar to RepoNavigator, we fix this by adding submission reminder before the last turn. Appendix B.2 includes further experimental details.

5 RESULTS

This section describes our experimental results. Table 2 provides a summary, and detailed results for SWE-Bench Verified are in Table 3, SWE-Bench Pro are in Table 4, and SWE-Bench Lite are in Table 5. Appendix B.4 discusses the statistical significance of the reported performance improvements.

5.1 CODESCOUT OUTPERFORMS **8-18X** LARGER BASE LLMS WITH OPENHANDS-BASH

For all benchmarks and search granularities, CODESCOUT LLMs *significantly* outperform their corresponding base models using OpenHands-Bash. CODESCOUT-1.7B achieves absolute gains in F1 score over its base LLM of **40-54%** for files, **25-41%** for modules, and **18-28%** for functions. CODESCOUT-4B yields absolute gains of **14-19%**, **25-38%**, and **21-31%** in file, module, and function-level F1 scores respectively. CODESCOUT-14B has absolute F1 gains of **23-34%** for files, **25-39%** for modules, and **20-33%** for functions.

Table 3: Results on **SWE-Bench Verified**. CODESCOUT outperforms upto **8**× larger open-source LLMs, competing with/surpassing proprietary LLMs. Qwen2.5 results use instruct LLMs. ^{rem}uses submission reminder (§4.2), †trained with RL; ‡RFT from CODESCOUT-14B; $^{\triangle}$RFT from Claude-3.7-Sonnet. Best metric is **bold-faced** and 2^{nd} best is underlined.

Scaffold	LLM	File-level			Module-level			Function-level		
		F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.
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	79.94	81.92	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	61.17	62.23	63.35	54.79	56.80	56.53
	Claude-Sonnet-4.5	**82.01**	**84.50**	82.86	**67.19**	**70.11**	**67.47**	**61.78**	**65.42**	61.99
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$^{\triangle\dagger}$	30.09	18.80	83.11	-	-	-	25.57	17.68	62.38
	Qwen2.5-32B$^{\triangle\dagger}$	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-1.7B-RFT‡	46.60	48.60	45.82	29.79	31.60	29.13	23.04	25.30	22.32
	CODESCOUT-1.7B‡†	55.46	58.40	54.27	36.45	39.37	35.46	28.22	31.77	27.18
	CODESCOUT-4B†	68.52	**71.53**	67.74	45.97	49.70	44.97	36.78	40.71	35.72
	CODESCOUT-14B†	**68.57**	71.00	68.69	**50.88**	**53.71**	**50.88**	**40.32**	**43.74**	40.27

CODESCOUT demonstrates exceptional parameter efficiency and consistently outperforms significantly larger base LLMs. CODESCOUT-1.7B outperforms the **8**× larger Qwen3-14B with absolute F1 gains of **11-18%** for files, **13-21%** for modules, and **10-15%** for functions. When compared against the **18**× larger Qwen3-32B, CODESCOUT-1.7B remains competitive, surpassing its F1 score by **2-4%** and **3-6%** for modules and functions respectively, while trailing by **2-7%** for files. CODESCOUT-4B consistently outperforms **8**× larger Qwen3-32B for all benchmarks, with absolute F1 gains of **5-8%**, **11-16%**, and **13-17%** for files, modules, and functions respectively. Finally, CODESCOUT-14B surpasses F1 scores of Qwen3-32B **6-13%** for files, **15-22%** for modules, and **16-21%** for functions.

5.2 CODESCOUT OUTPERFORMS LARGER BASE AND POST-TRAINED LLMS USING COMPLEX AGENTS

Comparisons with larger base LLMs using complex scaffolds: CODESCOUT LLMs consistently outperform Qwen2.5-32B with the OrcaLoca scaffold (Yu et al., 2025) on SWE-Bench Verified. CODESCOUT-4B and CODESCOUT-14B *significantly* exceed its F1 scores by **10%** for files and **8-11%** for functions. Impressively, our **18**× **smaller** CODESCOUT-1.7B remains competitive, trailing Qwen2.5-32B with OrcaLoca by $<$ **0.5%** in function-level F1 and by **2.65%** in file-level F1. On SWE-bench Pro, CODESCOUT models exhibit more pronounced gains over Qwen2.5-32B with the CoSIL agent (Jiang et al., 2025). CODESCOUT-1.7B, 4B, and 14B achieve absolute F1 gains of **20-33%** for files and **11-21%** for functions.

Table 4: Results on **SWE-Bench Pro**. CODESCOUT outperforms upto **8**× **larger** base and post-trained LLMs across various agents, narrowing the gap with closed-source LLMs. Qwen2.5 results use instruct LLMs. ^{rem} adds submission reminder (§4.2), †trained with RL; ‡RFT from CODESCOUT-14B. Best metric is **bold-faced** and 2^{nd} best metric is underlined.

Scaffold	LLM	File-level			Module-level			Function-level		
		F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.
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-1.7B-RFT‡	34.54	47.74	30.22	22.43	36.53	18.59	16.07	29.04	13.02
	CODESCOUT-1.7B‡†	40.96	56.52	35.91	25.27	40.66	20.83	18.24	32.08	14.72
	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**

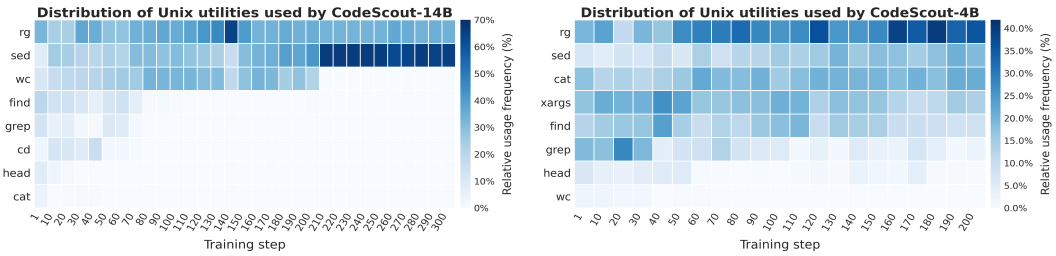
Comparisons with larger post-trained LLMs using complex scaffolds: We primarily compare CODESCOUT with RepoNavigator (Zhang et al., 2025) as it is our strongest post-trained baseline. It is also closer to our work than RepoSearcher (Ma et al., 2025) as it is trained without relying on closed-source LLMs and can predict a variable number of locations (§4.2). For both SWE-bench Verified and Pro, CODESCOUT models frequently achieve superior or competitive performance over **3-8**× **larger** RepoNavigator models. For both benchmarks, CODESCOUT-1.7B achieves slightly better performance than RepoNavigator-7B with a **1-4%** higher F1 score for both files and functions. Furthermore, CODESCOUT-4B consistently outperforms RepoNavigator-14B on both benchmarks, with absolute F1 improvements of **2-10%** for files and **8-11%** for functions. Comparison with RepoNavigator-32B reveals distinct trends across benchmarks. On SWE-bench Verified, CODESCOUT-4B and CODESCOUT-14B achieve better function-level F1 by **3-6%** with a very similar file-level F1. On SWE-bench Pro, CODESCOUT-4B and CODESCOUT-14B outperform RepoNavigator-32B by **8%** in function-level F1, whereas the 32B baseline has **4-6%** better file-level F1 scores.

5.3 CODESCOUT NARROWS THE PERFORMANCE GAP WITH CLOSED-SOURCE LLMS

On SWE-bench Verified, CODESCOUT-4B and 14B outperform GPT-5-Chat with the specialized RepoNavigator agent, with F1 gains of **9%** for files and **5-9%** for functions. CODESCOUT-4B and 14B surpass function-level F1 score of Claude-3.7-Sonnet with RepoNavigator by **5-8%**. Interestingly, Claude-Sonnet-4.5 achieves stronger performance with OpenHands-Bash over RepoNavigator, with **18%** higher function-level F1 and slightly better file-level F1 (2%). Similarly, GPT-5 with our agent outperforms Claude 3.7 Sonnet with RepoNavigator, with absolute F1 gains of **23%** for functions and **5%** for files. Thus, specialized scaffold design may not always improve performance. Soni et al. (2025); SWE-Agent Team (2024) report similar findings for other domains. Notably, GPT-5 and Claude-Sonnet-4.5 consistently outperform CODESCOUT when using OpenHands-Bash with the submission reminder (§4.2). However, both LLMs are surprisingly sensitive to prompt engineering. Without the reminder, their performance drops to ≈**0**. Appendix F includes more discussion on the submission reminder.

Table 5: Results on **SWE-Bench Lite**. CODESCOUT outperforms/competes with base LLMs upto **18**× larger, narrowing the gap with proprietary LLMs. Qwen2.5 results use instruct LLMs. ^{rem}uses submission reminder (§4.2), †trained with RL; ‡RFT from CODESCOUT-14B; $^{\triangle}$RFT from Claude-3.7-Sonnet. Best metric is **bold-faced** and 2^{nd} best is underlined.

Scaffold	LLM	File-level			Module-level			Function-level		
		F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.
Closed-Source LLMs										
LocAgent	Claude-3.5-Sonnet	31.39	18.83	**94.16**	29.91	18.18	**86.98**	27.53	17.08	**76.61**
OpenHands-Bash	GPT-5	1.09	1.09	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	Claude-Sonnet-4.5	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.36
OpenHands-Bash^{rem}	GPT-5	77.77	75.73	82.48	67.86	66.09	72.63	**61.12**	59.43	67.21
	Claude-Sonnet-4.5	**81.87**	**80.17**	85.40	**69.62**	**68.87**	72.87	61.11	**61.72**	63.72
Open-Source LLMs										
OpenHands-Bash	Qwen3-1.7B	2.16	1.96	2.92	0.36	0.36	0.36	0.00	0.00	0.00
	Qwen3-4B-Instruct	47.41	43.70	55.47	14.50	13.90	16.24	8.72	8.67	9.61
	Qwen3-14B	38.63	31.30	71.90	20.10	16.74	31.57	11.73	9.88	18.13
	Qwen3-32B (Thinking)	58.98	54.26	71.53	37.20	34.64	43.98	23.76	23.11	26.89
	CODESCOUT-1.7B-RFT‡	45.99	45.99	45.99	34.79	35.22	34.67	24.51	25.18	24.39
	CODESCOUT-1.7B‡†	56.57	56.57	56.57	41.24	41.61	41.24	27.07	28.28	26.82
	CODESCOUT-4B†	67.03	66.61	67.88	53.10	53.47	53.65	39.87	41.59	39.96
	CODESCOUT-14B†	**71.84**	**71.17**	**73.36**	**59.23**	**59.18**	**60.28**	**44.43**	**45.59**	**45.13**



(a) Tool use for CODESCOUT-14B

(b) Tool use for CODESCOUT-4B

Figure 3: Distribution of the top-8 most frequent commands used by CODESCOUT. LLMs start with diverse utilities and converge to a small set as training proceeds. CODESCOUT-14B uses only ripgrep (rg) and sed, while CODESCOUT-4B mainly uses rg, cat, sed, and xargs.

6 ANALYSIS

We discuss the benefits of effective localization on issue resolution (§6.1), the cross-language generalization of CODESCOUT (§6.2) and evolving tool-use behaviors during RL (§6.3). Appendix H includes detailed error analysis of CODESCOUT models.

6.1 DOES EFFECTIVE CODE LOCALIZATION IMPROVE ISSUE RESOLUTION?

We show that augmenting OpenHands issue resolution agent (Wang et al., 2025a) with relevant code retrieved by CODESCOUT improves performance and efficiency on SWE-Bench Verified. We consider three settings: (1) a vanilla baseline without localization, (2) augmenting the agent with locations retrieved by CODESCOUT-14B, and (3) augmenting the agent with oracle locations from the gold patch (§3.1). For (2) and (3), the user prompt is modified to include the locations of relevant files, modules, and functions (Appendix L). We evaluate Qwen3-4B-Instruct-2507 and Qwen3-Coder-30B-A3B-Instruct and report % of issues resolved, number of steps per trajectory and the total input and output tokens accumulated across all turns in Table 6.

For Qwen3-4B-Instruct, augmenting the agent with CODESCOUT-14B's outputs improves resolution rate by **3.8%**, while reducing trajectory length by **2.18** steps, and input and output tokens by **17.46%** and **6.71%** respectively. Similar efficiency gains are observed for Qwen3-Coder-30B-A3B-Instruct, with **2.89** fewer steps and **5.33%** fewer input tokens and **2.00%** fewer output tokens, while achieving comparable performance (+0.80%). For both LLMs, replacing CODESCOUT predictions with oracle locations yields further performance gains, and the 30B model also benefits with improved efficiency. Thus, continued improvements in localization further improve performance and efficiency of issue resolution agents. Appendix E shows fine-grained performance breakdown with respect to CodeScout-14B's localization quality.

Table 6: Issue resolution performance on SWE-Bench Verified: augmenting the agent with relevant locations improves performance and efficiency. Best metric is **bold-faced** and 2^{nd} best is underlined.

Localization Approach	Resolution Rate ↑	Avg. # Steps ↓	Avg. Input Tokens ↓	Avg. Output Tokens ↓
Qwen3-4B-Instruct				
None (Vanilla)	13.40%	16.09	344.37K	2.98K
CODESCOUT-14B	17.20%	**13.91**	**284.26K**	**2.78K**
Oracle	**19.60%**	16.41	327.75K	3.17K
Qwen3-Coder-30B-A3B-Instruct				
None (Vanilla)	45.20%	51.00	1596.71K	13.49K
CODESCOUT-14B	46.00%	48.11	1511.53K	13.22K
Oracle	**52.00%**	**46.74**	**1358.60K**	**12.48K**

6.2 DOES CODESCOUT GENERALIZE ACROSS PROGRAMMING LANGUAGES?

We show that the performance improvements of CODESCOUT over their base models generalize to programming languages other than Python, despite our RL post-training using Python-only repositories. We evaluate CODESCOUT-4B and 14B, and their corresponding base models, on the 465 non-Python instances in SWE-Bench Pro, that include Go, JavaScript, and TypeScript, using our language-agnostic OpenHands-Bash scaffold with a similar evaluation setup from §4.2. Since the patch processing scripts from LocAgent (Chen et al., 2025) only support Python, we do not have ground truth functions and modules edited by the patch, and only report file-level localization performance ignoring functions and modules predicted from the model.

CODESCOUT-4B improves file-level F1 from 17.38 to **32.56**, while CODESCOUT-14B improves from 19.41 to **22.99**. These results provide evidence that the learned localization behavior transfers across programming languages, while our Unix-terminal-based scaffold requires no language-specific engineering. All specialized baseline scaffolds from §4.2 require additional engineering to perform localization on languages other than Python.

6.3 HOW DOES THE TOOL-USE BEHAVIOUR OF CODESCOUT EVOLVE DURING RL?

To analyze the evolving tool-use behaviors during RL, we plot the distribution of Unix utilities invoked by the LLM in Figure 3. We parse rollouts sampled from model checkpoints at every 10 training steps and aggregate command usage across these checkpoints to identify the top-8 frequently used commands. For each checkpoint, we compute the relative usage frequency of each utility for all rollouts logged in that training step.

We observe convergence in tool usage during RL. While the 14B LLM initially uses various utilities, after 200 training steps it uses *only* two commands: `ripgrep` (`rg`) and `sed`. Similarly, while the 4B LLM initially uses many utilities, it converges to mainly using `ripgrep`, `sed`, `cat`, `find` and `xargs`. Thus, we can further simplify the agent to a small subset of Unix utilities without needing access to the entire terminal which is crucial in security-sensitive deployments. Appendix J presents example trajectories for 4B and 14B LLMs.

7 CONCLUSION

Our work presents a fully open-source RL recipe for training code search agents and releases the CODESCOUT family, achieving state-of-the-art localization results without complex, language-specific scaffolds used by prior work. On SWE-Bench Verified, Pro, and Lite, our models outperform larger open-source LLMs across complex setups and narrow the gap with some closed-source models. We show that augmenting code agents with localized context improves efficiency and performance, and analyze fine-grained behavioral shifts during RL via command-line tool usage, highlighting the potential for even simpler scaffolds. Appendix G discusses limitations and future research directions.

AI USE STATEMENT

In this work, we used generative AI tools to edit a research paper to improve readability like aid with formatting latex and correcting grammar/improving writing style. We have not used generative AI tools to generate synthetic data sets, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results, and the rest of the required disclosure tasks like formulate mathematical claims, help develop theoretical models or conceptual frameworks, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, are not applicable to this work. We have reviewed all AI-assisted work. For example, we manually scrutinized the entire paper before final submission. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

CODESCOUT is a fully open-source RL recipe and we will release our model weights, code, and training data upon acceptance of our work. To ensure reproducibility of our RL training runs, we fix the random seed in our data-loaders, and we will also share all the trajectories curated from the various models during evaluation on different benchmarks for further fine-grained analysis. Since supporting the OpenHands agent harness in the SkyRL training stack required significant implementation effort, we will also release that to allow extending it to other agentic RL use-cases.

REFERENCES

Anthropic. Claude Code, 2025. URL https://code.claude.com/docs/en/overview. Agentic AI coding assistant.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

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, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Cursor Team. Composer: Building a fast frontier model with rl. https://cursor.com/blog/composer, October 2025. Accessed: 2026-01-28.

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?, 2025. URL https://arxiv.org/abs/2509.16941.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. Areal: A large-scale asynchronous reinforcement learning system for language reasoning, 2026. URL https://arxiv.org/abs/2505.24298.

Andrew Gallant. ripgrep. https://github.com/BurntSushi/ripgrep, 2016. Accessed: 2026-03-11.

Tyler Griggs, Sumanth Hegde, Eric Tang, Shu Liu, Shiyi Cao, Dacheng Li, Charlie Ruan, Philipp Moritz, Kourosh Hakhamaneshi, Richard Liaw, Akshay Malik, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Evolving skyrl into a highly-modular rl framework, 2025. Notion Blog.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025. URL https://research.trychroma.com/context-rot.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents, 2025. URL https://arxiv.org/abs/2504.07164.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching, 2025. URL https://arxiv.org/abs/2503.22424.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Myeongsoo Kim, Shweta Garg, Baishakhi Ray, Varun Kumar, and Anoop Deoras. Codeassistbench (cab): Dataset benchmarking for multi-turn chat-based code assistance, 2026. URL https://arxiv.org/abs/2507.10646.

Philipp Koehn. Statistical significance tests for machine translation evaluation. In Dekang Lin and Dekai Wu (eds.), *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pp. 388–395, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-3250/.

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, 2025. URL https://arxiv.org/abs/2503.20783.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Erran Li Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl. https://www.together.ai/blog/deepswe, July 2025. Together AI Blog.

Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. Tool-integrated reinforcement learning for repo deep search, 2025. URL https://arxiv.org/abs/2508.03012.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph, 2025. URL https://arxiv.org/abs/2410.14684.

Ben Pan, Carlo Baronio, Pietro Marsella Albert Tam, Mokshit Jain, Daniel Chiu, Swyx, and Silas Alberti. Introducing SWE-grep and SWE-grep-mini: RL for multi-turn, fast context retrieval. https://cognition.ai/blog/swe-grep, 2025a. Blog post.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025b. URL https://arxiv.org/abs/2412.21139.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2026. URL https://arxiv.org/abs/2309.00071.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. Swe-qa: Can language models answer repository-level code questions?, 2025. URL https://arxiv.org/abs/2509.14635.

SID Research. Sid-1 technical report: Test-time compute for retrieval. *SID AI*, 2025. https://www.sid.ai/research/SID-1-technical-report.

Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3:333–389, 2009. URL https://api.semanticscholar.org/CorpusID:207178704.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297. ACM, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig. Coding agents with multimodal browsing are generalist problem solvers, 2025. URL https://arxiv.org/abs/2506.03011.

SWE-Agent Team. mini-swe-agent, 2024. URL https://github.com/SWE-agent/mini-swe-agent.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents, 2025a. URL https://arxiv.org/abs/2511.03690.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-bench: Can retrieval augment code generation? In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3199–3214, Albuquerque, New Mexico, April 2025b. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.176. URL https://aclanthology.org/2025.findings-naacl.176/.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1123–1139, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.62. URL https://aclanthology.org/2025.findings-acl.62/.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

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, 2024. URL https://arxiv.org/abs/2405.15793.

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, 2025b. URL https://arxiv.org/abs/2504.21798.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: An llm agent framework for software issue localization, 2025. URL https://arxiv.org/abs/2502.00350.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models, 2023. URL https://arxiv.org/abs/2308.01825.

Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Yang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, et al. One tool is enough: Reinforcement learning for repository-level llm agents. *arXiv preprint arXiv:2512.20957*, 2025.

Chujie Zheng, Pei Ke, Zheng Zhang, and Minlie Huang. Click: Controllable text generation with sequence likelihood contrastive learning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1022–1040, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.65. URL https://aclanthology.org/2023.findings-acl.65/.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025. URL https://arxiv.org/abs/2507.18071.

A PROMPTS AND TOOL DEFINITIONS FOR OPENHANDS-BASH

Figures 4, 5 describe the system prompt, and Figure 6 includes the user prompt, used for all LLMs (including CODESCOUT) trained or evaluated with OpenHands-Bash (§3.2). The system prompt specifies the `max_turns` to be 6 for CODESCOUT-4B and its base model, and to 4 for all other LLMs. Table 7 provides the schemas for the tools used by OpenHands-Bash.

Table 7: Pythonic Tool Schema for the tools used by OpenHands-Bash

Tool Name	Parameters	Python Type / Constraints	Required
terminal	command	str	✓
	security_risk	Literal["UNKNOWN", "LOW", "MEDIUM", "HIGH"]	✓
	is_input	bool	✗
	timeout	float	✗
	reset	bool	✗
localization_finish	locations	List[Dict[str, Optional[str]]]	✓
	↪ file	str	✓
	↪ class_name	str | None	✗
	↪ function_name	str | None	✗

B DETAILED EXPERIMENTAL SETTINGS FOR TRAINING AND EVALUATION

B.1 EXPERIMENTAL SETUP FOR TRAINING RUNS

This section details the experimental setup for all our training runs from §4.1 used for training CODESCOUT models. Note that when processing the ground truth patches in our training data we enhance the patch processing scripts from LocAgent (Chen et al., 2025) to (i) detect additions of member functions and class attributes at the module and file level, (ii) capture modifications to import statements and global variables at the file level, and (iii) ignore edits to docstrings within functions and classes.

We train CODESCOUT-4B for 200 steps on 1.6K instances using a batch size of 8 and sample 8 rollouts per instance. The model uses a maximum context length of 40K tokens, the reward function from Equation 1, and a maximum of 6 turns per episode. We train CODESCOUT-14B for 300 steps on 9.6K instances with a batch size of 32, sampling 4 rollouts per instance. We use a maximum context length of 50K tokens, extended with YaRN (Peng et al., 2026), and the reward function from Equation 1 augmented with the auxiliary term $r^{\text{turn}}(\tau, k)$ while k is set to 4 as the agent is allowed a maximum of 4 turns per rollout.

To warm-start the 1.7B model with rejection sampling fine-tuning (RFT) (Yuan et al., 2023), we sample rollouts from CODESCOUT-14B on a random subset of 7.7K training instances and retain only those achieving perfect localization across all three granularities (i.e., F1 = 1.0 for file, module, and function), yielding 4K training examples. We then use the veRL framework (Sheng et al., 2025) to perform supervised fine-tuning of `Qwen3-1.7B` on these successful trajectories for one epoch, with a learning rate of $5e^{-5}$, a cosine learning rate scheduler, warmup ratio of 0.1, global batch size of 8, and the AdamW optimizer (Loshchilov & Hutter, 2019). The resulting checkpoint (CODESCOUT-1.7B-RFT) is subsequently trained using RL (§3.4) for 100 steps on 800 instances (not seen during RFT) with a batch size of 8, sampling 8 rollouts per instance. We use 32K context window, the reward from Equation 1, and allow upto 4 turns per episode.

Across all CODESCOUT RL training runs, we use a constant learning rate of $1e^{-6}$, set `clip_ratio_low` to $3e^{-4}$ and `clip_ratio_high` to $4e^{-4}$, use the AdamW optimizer, and sample rollouts with temperature of 1.0. Note that we use a modified chat template for 1.7B and 14B models that preserves the `<think>` and `</think>` tokens from earlier turns during tokenization (`https://huggingface.co/OpenPipe/Qwen3-14B-Instruct`), overriding the default behavior that removes them. This modification preserves the sequence extension property, where previous trajectory steps are guaranteed to be prefixes for future steps allowing us to merge all steps into a single training sequence, greatly improving efficiency. All CODESCOUT models are trained with a single node of 8×H100 GPUs. Appendix C presents the training curves.

B.2 Experimental setup for evaluation

This section describes the experimental setup for all the evaluation runs performed in §4.2 and §6.1. During evaluation of localization performance of Qwen3 models, as well as during rollout generation from CodeScout-14B for training CodeScout-1.7B-RFT (§4.1), we use the decoding hyperparameters recommended by the Qwen3 developers. For instruct and reasoning models with thinking disabled, we use temperature $= 0.7$, top-$k = 20$, and top-$p = 0.8$. For reasoning models with thinking enabled, we use temperature $= 0.6$, top-$k = 20$, and top-$p = 0.95$. Finally, we set the maximum context length to 132K tokens for all Qwen3 models. For experiments in §6.1, our models are evaluated using a 128K token context window, allowing upto 100 turns, and using the recommended decoding hyperparameters from Qwen3 developers.

B.3 Experimental Costs for Training and Evaluation

Training costs: We use a single node of 8xH100 GPUs for RL training and the runtime is 3 hours for CodeScout-1.7B, 9 hours for CodeScout-4B, and 45 hours for CodeScout-14B. Most baselines that perform RL post-training do not report costs.

Evaluation Costs: Note that comparing inference costs of our simple bash-only scaffold with graph-based localization approaches like LocAgent (Chen et al., 2025) is non-trivial because they often involve an expensive pre-indexing step that pre-processes the repository to construct the code graph for each instance. Furthermore, most baselines do not report the LLM inference costs for their experiments making cost comparisons difficult. Using OpenHands-Bash agent harness, the total LLM inference cost (in USD) on SWE-Bench Verified is $4 for CodeScout-14B, $538 for GPT-5 and $600 for Claude-Sonnet-4.5. (for CodeScout-14B, we use OpenRouter API costs for Qwen3-14B). In the final revision, we will include detailed LLM inference costs across all benchmarks and models using our scaffold.

B.4 Statistical Significance Testing of Performance Improvements

We perform the statistical significance testing of the reported evaluation metrics to understand the reliability of the observed performance gains. Unfortunately, prior baselines do not provide the per-instance evaluation logs making this analysis infeasible. Thus, we compare CodeScout-4B and CodeScout-14B against Qwen3-32B which is the strongest OSS baseline using the OpenHands-Bash scaffold. For both pairs of models, we perform the paired bootstrap test (Koehn, 2004) for all three evaluation benchmarks and for all three localization granularities. We observe that the performance gains of CodeScout 4B and 14B over Qwen3-32B for all 18 comparisons (2 model pairs, 3 benchmarks, 3 granularities) are statistically significant with all p-values < 0.05. Furthermore, for module-level and function-level F1 scores, the p-values are < 0.0001.

C Reward and Loss Curves for CodeScout

This section presents the RL reward curves and SFT loss curves from the training runs of CodeScout. We present the aggregate reward (computed as the sum of the F1 scores across the three localization granularities for 4B and 1.7B, and for 14B as the sum of these three F1 scores and the auxiliary binary reward) and the individual file-level, module-level, and function-level F1 scores during training. Note that all reward curves are smoothed using a 16-step running average. Figures 7, 8, and 9 present these reward curves for CodeScout-14B, CodeScout-4B, and CodeScout-1.7B respectively. Figure 10 presents the SFT loss curve for CodeScout-1.7B-RFT.

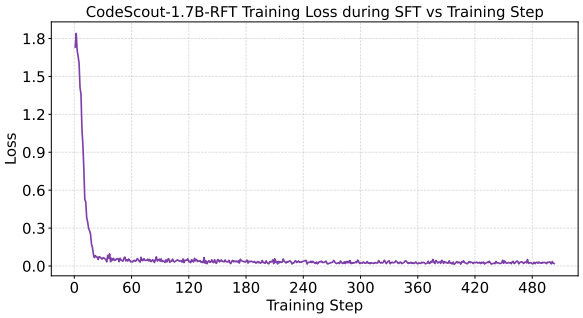


Figure 10: Training Loss curve during rejection sampling finetuning used to train CodeScout-1.7B-RFT.

Table 8: Comparison of tools used by prior code localization agents and CODESCOUT. Existing approaches rely on specialized task-specific tools, while CODESCOUT achieves competitive or superior performance using only a bash terminal typical of coding agents.

Method	Tools	Tool Behaviour
LocAgent (Chen et al., 2025)	`SearchEntity`	Performs keyword-based retrieval to identify relevant code entities.
	`TraverseGraph`	Traverse/parse codebase dependencies by navigating hops in the code graph.
	`RetrieveEntity`	Accesses the complete implementation for a specific entity ID.
CoSIL (Jiang et al., 2025)	`search_class_node`	Retrieves the full code snippet of a class given its file path.
	`search_class_function_node`	Extracts implementation details for a specific member function in a class.
	`search_file_function_node`	Fetches the source code for standalone/static functions within a file.
OrcaLoca (Yu et al., 2025)	`search_file_contents`	Returns file contents or a structural skeleton for long files (>200 lines).
	`search_class`	Locates and returns class definitions or their structural skeletons.
	`search_method_in_class`	Extracts specific method implementations from within a designated class.
	`search_callable`	Identifies and returns code snippets for callable objects (functions/methods).
	`search_source_code`	Performs a general search to match source code strings to snippets.
RepoSearcher (Ma et al., 2025)	`GetRepoStructure`	Provides an overview of the repository's file and directory hierarchy.
	`GetImportOfFile`	Identifies and lists all imports for a given file.
	`SearchClass`	Fetches the raw code content of a specific class definition.
	`SearchFunction`	Fetches the raw code content of a standalone function definition.
	`SearchClassMethod`	Specifically targets and retrieves implementations of class-specific methods.
RepoNavigator (Zhang et al., 2025)	`jump`	Navigates directly to a symbol's definition using a language server.
CODESCOUT *(Ours)*	`terminal`	Executes standard Unix commands in a persistent, stateful `tmux` session

As expected, all reward curves across all models show an approximately increasing nature (or remain constant during the later stages of training). Notably, all the reward curves for CODESCOUT-4B and the CODESCOUT-14B models show an increasing trend with no visible signs of saturation, indicating that training further on more GitHub issues will likely result in even stronger localization performance. However, we observe that the reward curves for CODESCOUT-1.7B have mostly plateaued and the model no longer shows signs of further improvement with more training. Interestingly, even after fine-tuning `Qwen3-1.7B` on **4K successful trajectories** sampled from CODESCOUT-14B with the training loss approaching *almost zero* (Figure 10), we find that training this checkpoint further using reinforcement learning improves the model performance even further - we observe an increasing reward curve with a steep slope during the first ≈ 20 training steps before saturating.

D Detailed Overview of Related Work

D.1 Approaches for code localization

Several prior methods have developed methods targeting code localization for downstream issue resolution in repositories. Jimenez et al. (2024) identify relevant source code files using BM25 retrieval (Robertson & Zaragoza, 2009) treating the issue description as the query and the Python source code files as documents. SWE-Fixer (Xie et al., 2025) use a two-step coarse-to-fine localization approach where they first retrieve relevant source code files using BM25 retrieval and then prompt a fine-tuned 7B model with the skeleton of the retrieved files to predict the names of relevant files for issue resolution. On the other hand, Agentless (Xia et al., 2024) uses a complex, multi-step localization pipeline wherein it first retrieves suspicious code files from the repository using both embedding-based retrieval and by prompting an LLM with the high-level repository structure, followed by prompting an LLM to predict relevant functions and classes given the skeletons of the suspicious files.

Many recent approaches have increasingly shifted towards developing models and systems for code localization using agentic scaffolds. Table 8 provides an overview of tools used by several prior agentic approaches and CODESCOUT for code localization. LocAgent (Chen et al., 2025) and RepoGraph (Ouyang et al., 2025) utilize a graph-based indexing approach wherein all dependencies in the code repository (for e.g. import, invoke, inherit, etc.) are captured in a code graph and also develop specialized tools allowing the agent to search and traverse the graph. Furthermore, Chen et al. (2025) train a 7B model for their scaffold using rejection sampling fine-tuning on successful trajectories sampled from Claude-3.5-Sonnet. On the other hand, CoSIL (Jiang et al., 2025) first localizes relevant files using module-call graphs and then identifies relevant functions using function-

call graphs. Note that these call graphs are constructed on-the-fly during inference as opposed to pre-indexing the code graph done by Chen et al. (2025). OrcaLoca (Yu et al., 2025) performs code localization through efficient exploration of the code graph using priority-based action scheduling, action decomposition with relevance scoring, and distance-aware context pruning. Note that many of these scaffolds are not suitable for reinforcement learning as they often require expensive repository pre-processing (for e.g. creating a code graph) increasing the computational overhead of performing rollouts during reinforcement learning.

RepoSearcher (Ma et al., 2025) develop a light-weight scaffold with tools specialized for localization that allow searching for classes and functions in files, obtaining imports of a given file, etc. They use a two-stage approach to train open-source LLMs for their scaffold: (1) rejection sampling fine-tuning from a closed-source LLM (Claude3.7-Sonnet) to warm-up their model, (2) reinforcement learning on the fine-tuned checkpoint to further enhance performance. RepoNavigator (Zhang et al., 2025) use an even simpler scaffold comprising with just one tool - `jump` - which allows the agent to retrieve definitions of Python symbols in files. Although Zhang et al. (2025) do not rely on rejection sampling fine-tuning from closed-source LLMs and directly train models using reinforcement learning, they still rely on selecting "easy" training instances by discarding all instances that were not successfully solved by the base Qwen2.5-7B model (equipped with their scaffold) atleast once among 16 sampled trajectories.

D.2 Training methods for software engineering agents

Several prior approaches have trained LLM-based software engineering agents for various downstream tasks like code localization and issue resolution. Many methods rely on performing rejection sampling fine-tuning (Yuan et al., 2023): sample trajectories (for training instances) from a stronger (often closed-source) model and train a smaller model on trajectories which successfully solve the task. Prior approaches that use this approach include LocAgent (Chen et al., 2025) and RepoSearcher (Ma et al., 2025) for code localization, and SWE-Gym (Pan et al., 2025b), SWE-Smith (Yang et al., 2025b), and R2E-Gym (Jain et al., 2025) for issue resolution. Recently, various attempts have been made to train models directly with reinforcement learning without relying on proprietary, closed-source LLMs for rejection sampling fine-tuning. While RepoNavigator (Zhang et al., 2025), SWE-Grep (Pan et al., 2025a), and SID-1 (Research, 2025) train LLM agents with RL for code localization, DeepSWE (Luo et al., 2025) and SkyRL-v0 (Cao et al., 2025) leverage RL to train LLM agents for issue resolution.

E Fine-Grained Breakdown of Effect of Localization Quality on Issue Resolution

We analyze how issue resolution performance and inference efficiency vary with the quality of CodeScout-14B's retrieved locations. We group SWE-Bench Verified instances into four buckets based on the average localization F1 score across file-, module-, and function-level predictions, and report bucket-level averages for all downstream metrics. For both issue resolution LLMs, the highest-quality localization bucket yields substantially higher resolution rates and lower inference costs. In particular, the $[0.75, 1.00]$ bucket achieves resolution rates of 30.1% and 64.2% for the 4B and 30B agents, respectively, while also requiring fewer steps and substantially fewer input and output tokens than lower-quality localization buckets. While the trend is not perfectly monotonic across intermediate buckets, the results show a strong overall association between localization quality and downstream issue resolution performance and efficiency. Even with perfect localization, however, the issue resolution agent must still infer and implement the required code changes. Thus, identifying the correct locations substantially narrows the search space but does not make issue resolution trivial.

Table 9: Issue resolution performance and efficiency as a function of CodeScout-14B localization quality.

Issue Resolution LLM	Avg. F1 Bucket	Resolution Rate ↑	Avg. # Steps ↓	Avg. Input Tokens ↓	Avg. Output Tokens ↓
Qwen3-4B-Instr.	$[0.00, 0.25)$	6.4%	15.70	333.86K	3.23K
	$[0.25, 0.50)$	8.1%	16.71	444.75K	3.28K
	$[0.50, 0.75)$	14.7%	14.75	310.91K	2.92K
	$[0.75, 1.00]$	30.1%	11.17	179.87K	2.19K
Qwen3-Coder-30B-A3B-Instr.	$[0.00, 0.25)$	32.7%	52.88	1765.13K	14.11K
	$[0.25, 0.50)$	38.7%	54.31	1840.57K	15.72K
	$[0.50, 0.75)$	33.8%	46.91	1490.40K	12.77K
	$[0.75, 1.00]$	64.2%	42.32	1190.80K	11.76K

F Discussion on Submission Reminder

Despite clearly describing the maximum allowed turns in the system prompt for the agent (Appendix L), proprietary LLMs do not terminate within the specified turn limits. Therefore, we followed RepoNavigator (Zhang et al., 2025) and include an additional submission reminder for these frontier closed-source LLMs. We argue that this failure-to-terminate behaviour is a weakness of the LLMs and not of the harness since we do not encounter such problems for open-source LLMs under an identical scaffold. Concretely, we observe that the failure-to-terminate rates on SWE-Bench Verified are only 0.2% for Qwen3-32B and CodeScout-14B, and 2.84% for Qwen3-14B.

G Limitations and Directions for Future Work

Although the proposed CodeScout recipe is significantly more general and scalable than most prior approaches for code localization, several important limitations remain. First, despite our scaffold being inherently programming language-agnostic by design and our models generalizing to other programming languages, our training experiments are largely restricted to Python repositories. This constraint arises as the patch processing scripts from LocAgent (Chen et al., 2025) for extracting ground-truth locations requires programming language-specific processing. Thus, we leave this analysis for future work.

Another limitation is that our ground truth localization target is primarily derived from the modified regions of the ground truth issue resolution code patch. However, note that there may be multiple valid patches that can resolve an issue that our recipe does not reward the agent for predicting locations from these alternative issue resolution patches. Furthermore, our data curation methodology omits other code segments that are relevant for resolving an issue but do not require modification. Note that all these limitations also exist in all the baselines we compare CodeScout against. Moreover, it is non-trivial to address them as we need to curate data at a training scale. First, all the issues in our training and evaluation datasets provide a single ground truth issue resolution patch without any alternative solutions. Second, determining relevance of a code location (that has not been modified) for resolving a given GitHub issue is somewhat subjective and difficult to determine without in-depth understanding of the repository. Thus, we leave it to future work to develop better data curation methodologies to address these limitations.

Thirdly, we acknowledge that several baselines use models from the Qwen2.5 series, while CodeScout uses the Qwen3 series. However, note that our work makes significant efforts to compare CodeScout against numerous competitive baselines to ensure fair comparisons. Firstly, we include comparisons of CodeScout with significantly larger Qwen3 base models when using the same scaffold (§5.1). Secondly, when comparing with prior complex scaffolds that use models from Qwen2.5 series, we primarily compare CodeScout against **2-18**$\times$ larger LLMs (§5.2). Finally, we also include comparisons with numerous frontier closed-source LLMs, both when using specialized agents and our OpenHands-Bash scaffold (§5.3). We argue that our evaluation setup is already very comprehensive and we leave it to future work to run additional RL experiments with these specialized agent frameworks equipped with the latest model series, given the high cost of running RL experiments.

Furthermore, while our reward design (§3.3) is quite intuitive and incorporates localization performance at three distinct granularities (files, modules, and functions), and a turn-limit reward for the 14B LLM, we do not study more sophisticated approaches to aggregate these individual components instead of simply summing them with equal weights. Alternative reward designs may allocate different weights to these individual components, which may result in better localization performance. However, we do not perform this analysis because determining these individual weights is non-trivial and often requires numerous ablation experiments for RL training that are very expensive to perform.

In addition, as with all prior baselines that post-train models for code localization, we acknowledge that CodeScout models are trained and evaluated under an identical scaffold (OpenHands-Bash), and the models may not generalize equally well to other agent harnesses. While training with multiple scaffolds can potentially improve the generalization of CodeScout models, it significantly increases the cost and complexity of our experiments. Thus, cross-scaffold training and evaluation is beyond the scope of our work.

Regarding the choice of training algorithm, we acknowledge that we do not perform exhaustive ablations on different algorithmic choices and training hyper-parameters, and further performance gains may be potentially achieved by tuning these. This analysis is beyond the scope of our work due to resource limitations and high experimental costs of RL training experiments.

Finally, localizing relevant/faulty code from the repository for resolving GitHub issues does not cover the broader scope of repository-level code search tasks, and our work does not address general-purpose repository-level question-answering/search tasks like those covered by benchmarks like CodeAssistBench (Kim et al., 2026) and SWE-QA (Peng et al., 2025). Future research directions include expanding the scope of our work to a broader range of tasks requiring repository-level code localization.

H Error Analysis

This section presents insights and key takeaways from our quantitative and qualitative error analysis of agent trajectories for evaluation benchmarks.

H.1 How does the localization performance vary with number of ground truth locations edited in the issue resolution patch?

We analyze the effect of number of locations (files and functions) edited in the ground truth issue resolution patch on localization performance of different LLMs when using the OpenHands-Bash scaffold. This analysis is performed with the 14B, 4B and 1.7B variants of CodeScout, and we also include the top-3 strongest performing LLMs - Claude-Sonnet-4.5, GPT-5, and Qwen3-32B (with thinking enabled). We consider the Python-only subset of SWE-Bench Pro (Deng et al., 2025) since it is significantly more challenging than SWE-Bench Verified (Chowdhury et al., 2024) and SWE-Bench Lite (Jimenez et al., 2024). Concretely, for each model, we group instances based on the number of locations (files or functions) in the ground truth localization targets, and compute the average F1 score of the agent on instances in each of these groups, and plot the trends for each of these granularities (file-level and function-level) in Figure 11.

For all LLMs, we observed that the localization performance generally decreases as the number of locations edited by the ground truth issue resolution patch increases, and this trend is especially evident for file-level localization. Interestingly, for file-level localization, we observe that the performance gap between the frontier, closed-source LLMs and CodeScout-14B and 4B models widens with increasing number of ground truth files, implying that our models still lag behind proprietary LLMs when localizing context for issues that require editing more files. A similar trend is observed when comparing Qwen3-32B and CodeScout-1.7B for file-level localization. Finally, we observe that rate of drop in model performance is much higher for the CodeScout models as compared to Qwen3-32B. We hypothesize that this because our training dataset (SWE-Smith) does not include many complex issues that edit several files as a result of which the model may not generalize too well on such issues.

Moreover, we observe that the drop in performance with increasing number of functions edited in the ground truth has a much smaller slope as compared to that for file-level localization for all LLMs. A potential reason for this observation is that many issues require editing multiple functions in the smaller set of file(s) and LLMs can identify these relatively more easily once the right set of files are localized. Similar to file-level localization, we observe that closed-source LLMs generally achieve much better localization performance over CodeScout-14B and 4B models for issues that edit a larger number of functions compared to the performance gap on issues that edit fewer functions. However, when comparing CodeScout-1.7B with Qwen3-32B, we find that the 1.7B consistently achieves stronger or competitive performance over **18x** larger base LLM regardless of the number of ground truth functions. This trend contrasts against that observed for file-level localization for these models implying that LLMs from the Qwen3 family greatly benefit from our RL recipe for function-level localization and base LLMs perform much poorly at this task.

H.2 QUALITATIVE ERROR ANALYSIS

To understand the qualitative error behaviours of CODESCOUT models when performing code localization, we analyze agent trajectories on the subset of examples from SWE-Bench Pro. In particular, we choose 20 successful trajectories (where the total F1 score for the three granularities is $>= 2.4$) and 20 failure trajectories (where the total F1 score for the three granularities is $<=$ 0.3), and identify model behaviours responsible for success/failure during localization. We perform this analysis for CODESCOUT-1.7B, 4B, and 14B models. Across all three model scales, our analysis reveals consistent behavioral contrasts between successful and failed localization attempts as discussed below. Note that the general problem-solving strategy for all these models is to first perform a broader keyword-based search to identify the right set of files to view, and then read specific line ranges from these files to determine the relevant set of classes and functions.

Search specificity: We find that the ability of the model to search for precise keywords in the beginning of the trajectory is the strongest differentiator - successful trajectories are characterized by precise search terms like specific function or class names (for e.g. `signal_name`, `isidentifier`, or `get_distribution`) whereas failed trajectories, by contrast, rely on broad domain keywords (e.g., `default`, `pip`, `load`) that often floods the context window with irrelevant matches. The ability of the model to determine these keywords also depends on the issue specificity as it is harder to determine these keywords from under-specified/ambiguous issues, and also on the nature of the issue; bug reports often include more low-level details using error logs/code snippets for reproducing/describing the error whereas feature requests are often described at a higher-level that require adding new code in the repository. A related error behaviour that further distinguishes the two groups lies in the model's ability to determine the correct directories to search within. Successful trajectories always reach the ground truth directory as expected, whereas failed trajectories, in contrast, reach the ground truth directory in much fewer cases (often due to imprecise keyword-search). Surprisingly, we also observe some failed trajectories where the ground truth file *was viewed* during exploration, yet the model still selected a different file, indicating a failure in final-stage reasoning rather than localization.

Architectural layer selection: We observe a consistent pattern of failures in architectural layer selection that reveals the ability of the model to understand the codebase structure. For example, selecting test files instead of source code files, or utility files and classes instead of the core implementation modules, especially when these files have similar names. Successful trajectories correctly identify the implementation layer where the fix must belong, whereas failed trajectories often make mistakes here targeting the wrong layers like utility scripts, configuration files, tests, etc. instead of the actual source code that needs to be fixed.

Multi-file localization: As previously discussed, successful localization trajectories often involve identifying fewer files on average as compared to the failed trajectories. One reason for this behaviour is that our models are not explicitly trained to perform parallel tool-calling to explore multiple files simultaneously. As a result, for issues that require changes to several files, the models sometimes fail to capture cross-file dependencies and focus on a subset of files leading to incomplete localization. Another reason for this behaviour also stems from the choice of our training dataset (SWE-Smith) which is synthetically curated and edits fewer files in general.

I ADDITIONAL DISCUSSION ON DESIGN OF REWARD FUNCTION

This section discusses our rationale behind the design of the reward function for our training experiments. The reward function from Equation 1 reflects the ability of the LLM agent to localize relevant files, modules, and functions, and is used for all CODESCOUT models. We need the additional auxiliary binary reward $r^{\text{turn}}(\tau, k)$ for the 14B model because our initial experiments without this reward observed a collapse in overall reward to $\approx$0 during later training stages wherein the LLM frequently exceeds the configured turn limit. We do not observe any such problems when training the 4B or 1.7B models, so we do not use this auxiliary binary reward for their training runs. We also explored auxiliary rewards to incentivize parallel tool-calling, motivated by Pan et al. (2025a), but found these interventions to hurt overall performance. Thus, we instead explicitly prompt the agent to use parallel tool-calling (Appendix A).

Table 10: Comparing localization performance and efficiency when training the 14B model with different choices of the auxiliary binary reward function for the turn-limit.

Benchmark	Reward function	File F1	Module F1	Function F1	Avg. # Steps
SWE-Bench Verified	$r^{\mathsf{turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	68.57	50.88	40.32	3.96
	$r^{\mathsf{flex-turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	70.37	46.43	37.30	3.78
SWE-Bench Pro	$r^{\mathsf{turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	53.63	37.13	28.74	3.98
	$r^{\mathsf{flex-turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	54.27	35.76	27.60	3.67
SWE-Bench Lite	$r^{\mathsf{turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	71.84	59.23	44.43	4.01
	$r^{\mathsf{flex-turn}}(\tau,k)+r^{\mathsf{F1-file}}(y,y^\star)+r^{\mathsf{F1-module}}(y,y^\star)+r^{\mathsf{F1-func}}(y,y^\star)$	70.92	53.22	39.29	3.62

A natural concern regarding the design of $r^{\mathsf{turn}}(\tau, k)$ is that why does it strictly assign a reward of 1 for using exactly k turns instead of flexibly assigning a reward of 1 whenever the agent terminates using $<= k$ turns. We investigate this by training the 14B model using this flexible auxiliary binary reward, $r^{\mathsf{flexible-turn}}(\tau, k)$ that assigns a reward of 1 whenever the agent submits its predictions using $<= k$ steps, with k set to 4. All other experimental settings are identical to those used in the default CODESCOUT-14B model. We evaluate both the LLMs on all three benchmarks (SWE-Bench Verified, Pro, and Lite) using our setup from §4.2. We also report the average number of steps used by the agent during localization for each of these benchmarks.

As shown in Table 10, we observe that using a flexible reward function for turn limits does not improve the localization performance of the model. While the file-level F1 scores remain comparable for both the reward variants, we observe that using a flexible auxiliary reward for turn limits results in weaker function-level and module-level F1 scores for all three benchmarks, with upto **6%** drop in module-level F1 score and utpo **5.14%** drop in function-level F1 score. The number of steps used by the agent to solve the task intuitively aligns with the respective reward designs - using a flexible reward for turn limit results in the agent using slightly fewer steps on average, whereas the model trained with the stricter turn limits uses ≈4 steps on average. Given the stronger localization performance of the stricter reward function for turn limits, our main paper describes results from the model trained with this stricter auxiliary reward.

J EXAMPLE TRAJECTORIES FOR CODESCOUT-14B AND CODESCOUT-4B

This section provides some example trajectories sampled from CODESCOUT-14B and CODESCOUT-4B models using our evaluation setup (§4.2). We use an identical GitHub issue (`django__django-13363`) from the SWE-Bench Verified benchmark (Chowdhury et al., 2024) for both these models to allow a direct comparison of their tool-use behaviors and problem-solving approaches. Note that both models achieve perfectly correct localization for this instance, i.e. they achieve an F1 score of 1.0 for all three granularities: file, module, and function. Note that we omit the system prompt and user prompt (which mentions the issue description) for these example rollouts. Figures 12 and 13 present the trajectory sampled from CODESCOUT-14B, and figures 14 and 15 present the trajectory sampled from CODESCOUT-4B.

As noted in §6.3, CODESCOUT-14B relies only on two Unix command-line utilities for code localization: `ripgrep (rg)` and `sed`. By analyzing the trajectory, we observe that the model uses `ripgrep` for efficient keyword-based search of the code repository and `sed` for reading specific line-ranges from relevant files. Therefore, by exploring the repository for *only* 3 steps and using *only* 2 command-line utilities, CODESCOUT-14B can determine the exact set of files, classes, and functions that require modification to fix a given GitHub issue, highlighting the effectiveness of our approach.

Interestingly, CODESCOUT-4B uses a few more Unix command-line utilities for code localization: `rg`, `find`, `cat`, `xargs` and `sed`. The model mostly uses `rg`, `find` and `xargs` during keyword-based searches, and `sed` and `cat` for reading particular line-ranges from files. Furthermore, the bash commands issued by CODESCOUT-4B are more complex and often use more than one Unix utility in the same command chained together through piping (|). Here as well, CODESCOUT-4B performs code localization by exploring the repository for *only* 3 steps and using *only* 5 command-line utilities.

For both the models, the command-line utilities used in these examples align with those used by the final training checkpoint in Figure 3.

A noticeable characteristic of CODESCOUT-4B is that it frequently performs parallel tool-calling unlike CODESCOUT-14B. We hypothesize that this is because of two reasons: (1) our training algorithm, and (2) choice of base model for these models. Firstly, our training algorithm does not reward/encourage parallel tool-calling and this is mainly mentioned through the system prompt: thus rollouts with parallel tool-calling have no advantage over those without any parallel tool-calling during training. Secondly, the base model for CODESCOUT-4B, `Qwen3-4B-Instruct-2507`, has strong instruction-following capabilities (stronger than `Qwen3-4B`, and possibly even better than `Qwen3-14B` which is the base model for CODESCOUT-14B). As a result, the simpler prompting mechanism is sufficient for our 4B model but not for the 14B model.

K ADDITIONAL COMPARISON WITH SCAFFOLDS THAT PREDICT A RANKED LIST OF TOP-K LOCATIONS

As opposed to some prior approaches that predict a fixed number of top-K relevant locations, CODESCOUT and RepoNavigator (Zhang et al., 2025) flexibly allow the agent to dynamically predict a variable number of locations. This has the added advantage of not having to choose an appropriate value of K based on the domain or choice of evaluation benchmark. In particular, for our choice of evaluation benchmarks, the design choice of predicting a ranked list of top-K locations (with K generally set to 5) results in a higher recall but lower precision. Importantly, this design is particularly not suitable for RL fine-tuning of LLM agents, especially when K is larger than the number of ground truth locations as there is no way to determine the relevance of the remaining locations predicted by the agent and the reward function will generally ignore these predictions completely during training. Furthermore, Pan et al. (2025a) report that polluting the context of the coding agent is more detrimental than leaving some context out, as the agent is typically only a few searches away from recovering any remaining context, implying that precision may be more crucial than recall when augmenting code agents for downstream issue resolution.

Note that the choice of K will also determine the trade-off between precision and recall, and the results in §5 for these baselines are based on the corresponding default choice of K used by their scaffolds. We also include comparisons with these methods for different choices of K to understand the precision-recall tradeoff and to compare these methods with CODESCOUT-4B and CODESCOUT-14B models. Table 11 presents the detailed results for three choices of K - 1, 3, and all (the default setting where K is set to 5). Impressively, CODESCOUT models outperform all prior methods that use a larger LLM across different choices of K even when dynamically predicting a variable number of locations. Furthermore, note that there is significant variance in the precision, recall, and F1 scores of the baseline methods for different choices of K implying that deciding the right value of K for the optimal performance is non-trivial and also depends on the characteristics of the GitHub issue and the benchmark/domain. Impressively, CODESCOUT strikes the right balance between precision and recall as a result of its scaffold design and our RL recipe that incorporates both precision and recall by computing the F1 scores.

L PROMPTS FOR ISSUE RESOLUTION WITH LOCALIZATION-AUGMENTED CONTEXT

We provide detailed user and system prompts in the OpenHands software agent SDK across various experimental settings used to demonstrate the advantages of augmenting issue resolution agents with relevant localization context (§6.1). For the vanilla setup that is not augmented with any additional localization context, we provide the user prompt in Figure 16. Figures 17 and 18 describe the user prompt for the experimental setting where the issue resolution agent is augmented with locations retrieved by CODESCOUT-14B. Figures 19 and 20 describe the user prompt for the experimental setting where the issue resolution agent is augmented with oracle locations. All three settings use the default system prompt for OpenHands Software Agent SDK (Wang et al., 2025a).

While scaffold engineering is not the primary research focus of our work, we note that the system and user prompts of OpenHands-Bash were manually written and some specific instructions (for e.g.

Table 11: Comparison of CODESCOUT with methods that predict a ranked list of top-K locations for different choices of K (Return@K) on **SWE-Bench Verified**. The setting `all` refers to the default setting with K=5 used by these scaffolds.

Scaffold + LLM	Return@K	File			Function		
		F1	Prec.	Rec.	F1	Prec.	Rec.
CoSIL + Qwen2.5-32B-Instruct	1	57.25	64.65	51.36	29.11	42.51	22.13
	3	41.60	29.53	70.30	25.35	20.80	32.46
	all	30.77	19.34	83.50	22.11	14.85	55.38
Agentless + Qwen2.5-32B-Instruct	1	56.17	63.93	50.08	22.18	32.78	16.75
	3	41.46	30.13	66.45	23.51	20.83	26.98
	all	35.38	25.60	78.93	27.33	24.07	40.97
LocAgent + Qwen2.5-32B-Instruct	1	64.66	78.31	55.06	32.78	52.06	23.92
	3	49.34	38.54	68.54	27.87	24.60	32.14
	all	44.18	34.18	79.39	21.48	16.29	46.79
CODESCOUT-4B	-	<u>68.52</u>	71.53	67.74	<u>36.78</u>	40.71	35.72
CODESCOUT-14B	-	**68.57**	71.00	68.69	**40.32**	43.74	40.27

output formatting) were added based on observed error behaviors from Qwen3 base LLMs in our initial evaluations. We argue that these prompts are already quite effective given that our OpenHands-Bash agent outperforms prior scaffolds using Claude-Sonnet-4.5, and has strong results with the Qwen3-32B. Importantly, note that we do not tune the system prompt based on error behaviors of CodeScout models to allow fair comparisons in our evaluation.

System Prompt for the OpenHands-Bash agent used by CODESCOUT (continued on next page)

```
You are a specialized code localization agent. Your sole objective
    is to identify and return the files in the codebase that are
    relevant to the user's query. You are given access to the
    codebase in a linux file system.

## PRIMARY DIRECTIVE
- Find relevant files, do NOT answer the user's query directly
- Prioritize precision: every file you return should be relevant
- You have up to {{ max_turns }} turns to explore and return your
    answer

## TOOL USAGE REQUIREMENTS
### bash tool (REQUIRED for search)
- You MUST use the bash tool to search and explore the codebase
- Execute bash commands like: rg, grep, find, ls, cat, head, tail,
    sed
- Use parallel tool calls: invoke bash tool up to 5 times
    concurrently in a single turn
- NEVER exceed 5 parallel tool calls per turn
- Common patterns:
  * `rg "pattern" -t py` - search for code patterns
  * `rg --files | grep "keyword"` - find files by name
  * `cat path/to/file.py` - read file contents
  * `find . -name "*.py" -type f` - locate files by extension
  * `wc -l path/to/file.py` - count lines in a file
  * `sed -n '1,100p' path/to/file.py` - read lines 1-100 of a file
  * `head -n 100 path/to/file.py` - read first 100 lines
  * `tail -n 100 path/to/file.py` - read last 100 lines
### Reading Files (CRITICAL for context management)
- NEVER read entire large files with `cat` - this will blow up your
     context window
- ALWAYS check file size first: `wc -l path/to/file.py`
- For files > 100 lines, read in chunks:
  * Use `sed -n '1,100p' file.py` to read lines 1-100
  * Use `sed -n '101,200p' file.py` to read lines 101-200
  * Continue with subsequent ranges as needed (201-300, 301-400, etc
     .)
- Strategic reading approach:
  * Read the first 50-100 lines to see imports and initial structure
  * Use `rg` to find specific patterns and their line numbers
  * Read targeted line ranges around matches using `sed -n 'START,
     ENDp'`
  * Only read additional chunks if the initial sections are relevant
### Submitting Your Answer (REQUIRED)

When you have identified all relevant locations, you MUST use the `
    localization_finish` tool to submit your results.

**When to include what:**
1. If the required modifications belong to a specific function that
     belongs to a class, provide the file path, class name, and
    function name.
2. If the required modification belongs to a function that is not
    part of any class, provide the file path and function name.
3. If the required modification does not belong to any specific
    class or a function (e.g. global variables, imports, new class,
    new global function etc.), it is sufficient to provide only the
    file path.
```

Figure 4: System prompt for the OpenHands-Bash agent scaffold used by CODESCOUT (continued on next page).

System Prompt for the OpenHands-Bash agent used by CODESCOUT

```
4. If the required modification belongs to a class (e.g. adding a
    new method to a class), provide the file path and class name.

## SEARCH STRATEGY
1. **Initial Exploration**: Cast a wide net
  - Search for keywords, function names, class names
  - Check file names and directory structure
  - Use up to 3 parallel bash calls to explore multiple angles
  - Check file sizes with `wc -l` before reading
  - Read promising files in chunks (lines 1-100) to verify
      relevance

2. **Deep Dive**: Follow the most promising leads
  - Use up to 3 parallel bash calls to investigate further
  - Read files in chunks to confirm they address the query
  - Use `rg` with line numbers to locate specific code, then read
      those ranges
  - Start eliminating false positives

3. **Final Verification**: Confirm your location list and terminate
     execution by calling the `localization_finish` tool

## CRITICAL RULES
- NEVER exceed 5 parallel bash tool calls in a single turn
- ALWAYS use the `localization_finish` tool after identifying all
    relevant locations
- ALWAYS use bash tool to search (do not guess file locations)
- NEVER read entire large files - always read in chunks (100-line
    ranges)
- Check file size with `wc -l` before reading
- Read file contents in chunks to verify relevance before including
     them
- Return file paths as they appear in the repository. Do not begin
    the path with "./"
- Aim for high precision (all files relevant) and high recall (no
    relevant files missed)
- Class and function names are OPTIONAL - only include when changes
     are at that level

## EXAMPLE OUTPUT BEHAVIOUR
Here are some examples of how to format your output when calling
    the `localization_finish` tool:
- src/parsers/parser.py requires changes to imports, a function
    parse_data which belongs to the class DataParser, and another
    function __str__ inside the same class. This should be
    represented as three separate entries: one with just the file
    path and one each for the two functions parse_data and __str__
    with file path, class name, and function name.
- src/user.py requires changes to a global function get_user
    outside of any class. This should be represented as a single
    entry with file path and function name.
- utils/visualizer.py requires adding new function visualize inside
     the class Visualizer. This should be represented as a single
    entry with file path and class name.
- utils/configs/default_config.py requires adding a new global
    function and a new class. This should be represented as a single
     entry with just the file path. Do NOT include class or function
     names for this file since multiple implementations might be
    possible with different function names.
```

Figure 5: System prompt for the OpenHands-Bash scaffold used by CODESCOUT.

User prompt for the OpenHands-Bash agent used by CODESCOUT

```
I have access to a python code repository in the directory {{
    working_dir }} . Consider the following issue description:

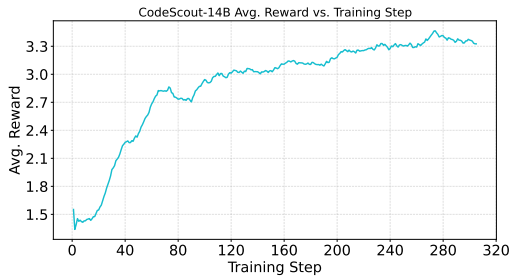
<issue_description>
{{ instance.problem_statement }}
</issue_description>

Act as a code search agent and localize the specific files, classes
     or functions of code that need modification to resolve the
    issue in <issue_description>.

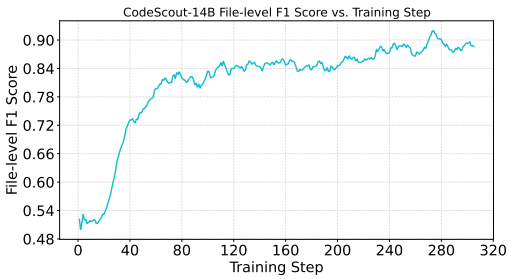
NOTE: You do not need to solve the issue, all you need to do is
    localize relevant code from the repository. Your output will be
    used to guide another agent to solve the issue.

IMPORTANT: Your output MUST follow the below rules:
1. The final output must be a tool call to the "localization_finish"
     tool containing relevant code locations.
2. The locations of the file path must be RELATIVE to the {{
    working_dir }} directory WITHOUT any leading "./" in the output.
3. Only include those locations in your output that need
    modification to resolve the issue in <issue_description>. Do NOT
     include any locations that do not need modification.
```

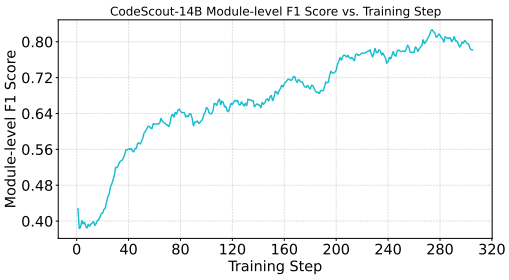
Figure 6: User prompt for the OpenHands-Bash scaffold used by CODESCOUT.



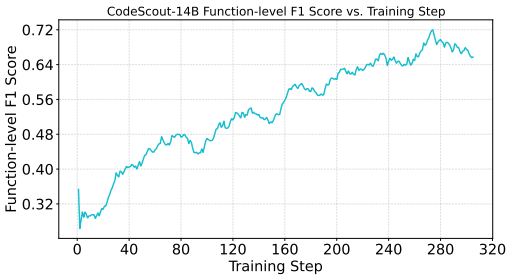
(a) Aggregate reward vs. Training Step



(b) File-level F1 score vs. Training Step

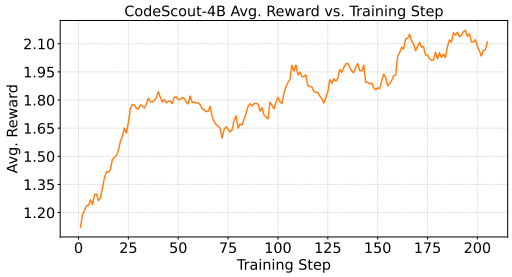


(c) Module-level F1 score vs. Training Step

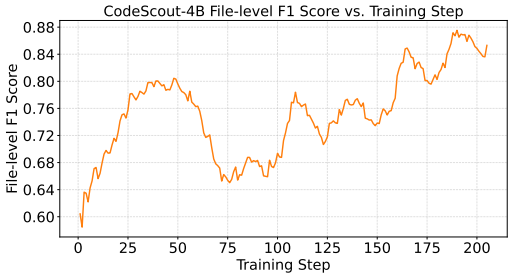


(d) Function-level F1 score vs. Training Step

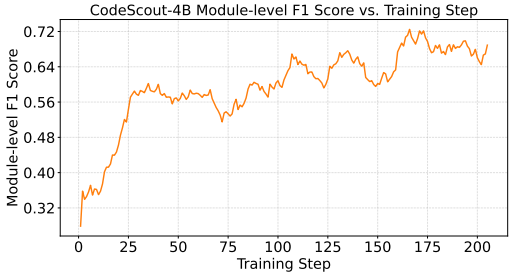
Figure 7: Reward curves for RL training of CODESCOUT-14B.



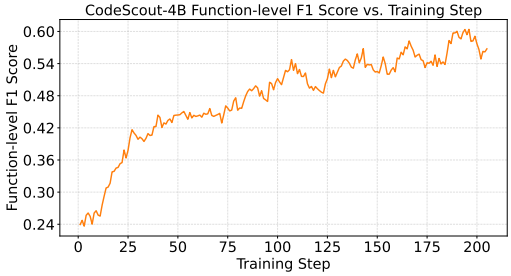
(a) Aggregate reward vs. Training Step



(b) File-level F1 score vs. Training Step

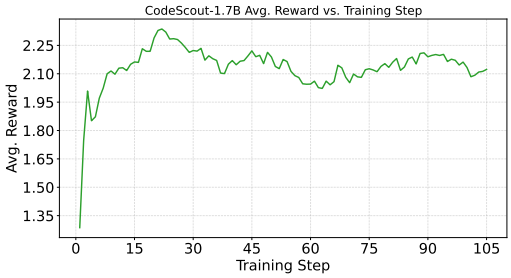


(c) Module-level F1 score vs. Training Step



(d) Function-level F1 score vs. Training Step

Figure 8: Reward curves for RL training of CODESCOUT-4B.



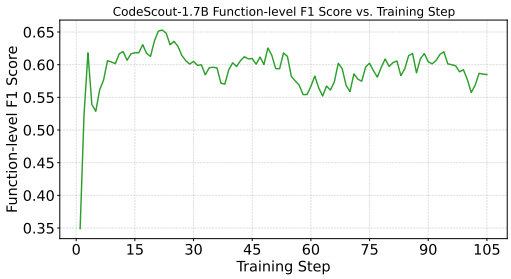
(a) Aggregate reward vs. Training Step



(b) File-level F1 score vs. Training Step



(c) Module-level F1 score vs. Training Step



(d) Function-level F1 score vs. Training Step

Figure 9: Reward curves for RL training of CODESCOUT-1.7B.

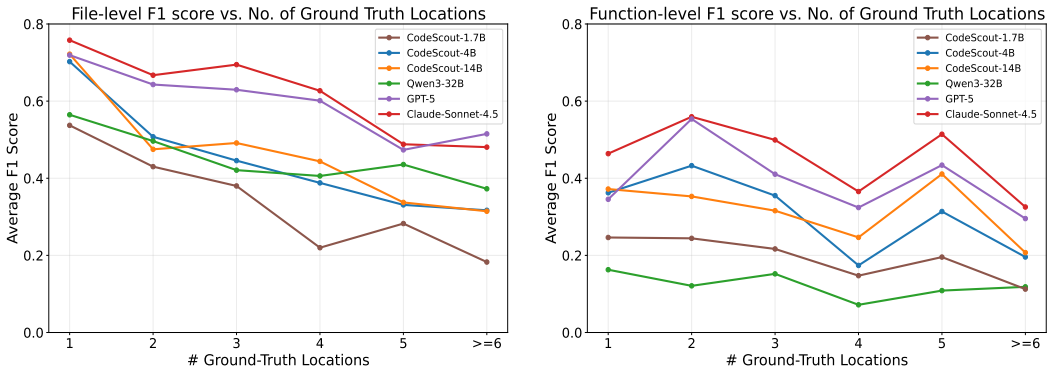


Figure 11: Average F1 score versus the number of locations edited by the ground truth issue resolution patch at the file-level (left) and function-level (right) on the SWE-Bench Pro dataset. The performance of all models drops as issues grow more complex with more files/functions being edited by the issue resolution patch.

Action 1 (Terminal Tool)

```
rg "TruncDate" -t py
```

Observation 1

```
django/db/models/functions/__init__.py
5: ExtractWeekDay, ExtractYear, Now, Trunc, TruncDate, TruncDay, TruncHour,
30: 'TruncDate', 'TruncDay', 'TruncHour', 'TruncMinute', 'TruncMonth',

django/db/models/functions/datetime.py
287:class TruncDate(TruncBase):
325:DateTimeField.register_lookup(TruncDate)

tests/db_functions/datetime/test_extract_trunc.py
13: ExtractWeekDay, ExtractYear, Trunc, TruncDate, TruncDay, TruncHour,
839: DTModel.objects.annotate(extracted=TruncDate('start_datetime')).order_by('
    start_datetime'),
```

Action 2 (Terminal Tool)

```
sed -n '1,500p' django/db/models/functions/datetime.py
```

Observation 2 (Trimmed)

```
from django.db.models.fields import (
   DateField, DateTimeField, DurationField, Field, IntegerField, TimeField,
)
...

class TruncBase(TimezoneMixin, Transform):
   def as_sql(self, compiler, connection):
      inner_sql, inner_params = compiler.compile(self.lhs)
      if isinstance(self.output_field, DateTimeField):
         tzname = self.get_tzname()
         sql = connection.ops.datetime_trunc_sql(self.kind, inner_sql, tzname)
...

class TruncDate(TruncBase):
   kind = 'date'
   lookup_name = 'date'
   output_field = DateField()

   def as_sql(self, compiler, connection):
      # Cast to date rather than truncate to date.
      lhs, lhs_params = compiler.compile(self.lhs)
      tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None
      sql = connection.ops.datetime_cast_date_sql(lhs, tzname)
      return sql, lhs_params

class TruncTime(TruncBase):
   kind = 'time'
   lookup_name = 'time'
   output_field = TimeField()
...
```

Figure 12: Example trajectory sampled from CODESCOUT-14B for the task `django__django-13363` (continued on the next page).

Action 3 (Terminal Tool)

```
rg "get_current_timezone_name" -t py
```

Observation 3

```
django/db/models/functions/datetime.py
25: tzname = timezone.get_current_timezone_name()
295: tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None
308: tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None

django/utils/timezone.py
64:def get_current_timezone_name():
```

Action 4 (Localization Finish Tool - Agent Loop Terminated)

```
[
   {
      "file": "django/db/models/functions/datetime.py",
      "class_name": "TruncDate",
      "function_name": "as_sql"
   },
   {
      "file": "django/db/models/functions/datetime.py",
      "class_name": "TruncTime",
      "function_name": "as_sql"
   }
]
```

Reward Computation

```
Ground truth locations:
   Files: django/db/models/functions/datetime.py
   Modules: django/db/models/functions/datetime.py:TruncDate, django/db/models/
       functions/datetime.py:TruncTime
   Functions: django/db/models/functions/datetime.py:TruncDate.as_sql, django/db/
       models/functions/datetime.py:TruncTime.as_sql
Rewards:
   File-level F1 score: 1.0
   Module-level F1 score: 1.0
   Function-level F1 score: 1.0
```

Figure 13: Example trajectory sampled from CODESCOUT-14B for the task `django__django-13363`.

Action 1 (Terminal Tool - 4 parallel calls)

```
# Command 1
find . -type f -name "*.py" | xargs rg -l TruncDate
# Command 2
find . -type f -name "*.py" | xargs rg -l TruncTime
# Command 3
rg -l TimeZone
# Command 4
rg -l TruncBase
```

Observation 1 (trimmed)

```
# [Results for Command 1]
./django/db/models/functions/datetime.py
./django/db/models/functions/__init__.py
./tests/db_functions/datetime/test_extract_trunc.py

# [Results for Command 2]
./django/db/models/functions/datetime.py
./django/db/models/functions/__init__.py
./tests/db_functions/datetime/test_extract_trunc.py

# [Results for Command 3]
django/db/backends/postgresql/base.py
django/templatetags/tz.py
tests/db_functions/datetime/test_extract_trunc.py
... (truncated)

# [Results for Command 4]
django/db/models/functions/datetime.py
```

Action 2 (Terminal Tool - 3 parallel calls)

```
# Command 1
cat django/db/models/functions/datetime.py | sed -n '1,500p'
# Command 2
cat django/db/models/functions/datetime.py | sed -n '501,1000p'
# Command 3
rg -n 'get_current_timezone_name' django/db/models/functions/datetime.py
```

Observation 2 (trimmed)

```
# [Results for Command 1]
...
from django.utils import timezone
class TimezoneMixin:
    def get_tzname(self):
...
class TruncDate(TruncBase):
    def as_sql(self, compiler, connection):
        tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None
        sql = connection.ops.datetime_cast_date_sql(lhs, tzname)
...

# [Results for Command 2 (empty)]
# [Results for Command 3]
25: tzname = timezone.get_current_timezone_name()
295: tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None
308: tzname = timezone.get_current_timezone_name() if settings.USE_TZ else None
```

Figure 14: Example trajectory sampled from CODESCOUT-4B for the task `django__django-13363` (continued on the next page).

Action 3 (Terminal Tool - 2 parallel calls)

```
# Command 1
rg -n 'datetime_cast_date_sql' django/db/models/functions/datetime.py
# Command 2
rg -n 'datetime_cast_time_sql' django/db/models/functions/datetime.py
```

Observation 3

```
# [Results for Command 1]
296: sql = connection.ops.datetime_cast_date_sql(lhs, tzname)

# [Results for Command 2]
309: sql = connection.ops.datetime_cast_time_sql(lhs, tzname)
```

Action 4 (Localization Finish Tool - Agent Loop Terminated)

```
[
   {
      "file": "django/db/models/functions/datetime.py",
      "class_name": "TruncDate",
      "function_name": "as_sql"
   },
   {
      "file": "django/db/models/functions/datetime.py",
      "class_name": "TruncTime",
      "function_name": "as_sql"
   }
]
```

Reward Computation

```
Ground truth locations:
   Files: django/db/models/functions/datetime.py
   Modules: django/db/models/functions/datetime.py:TruncDate, django/db/models/
        functions/datetime.py:TruncTime
   Functions: django/db/models/functions/datetime.py:TruncDate.as_sql, django/db/
        models/functions/datetime.py:TruncTime.as_sql
Rewards:
   File-level F1 score: 1.0
   Module-level F1 score: 1.0
   Function-level F1 score: 1.0
```

Figure 15: Example trajectory sampled from CODESCOUT-4B for the task `django__django-13363`.

User Prompt for OpenHands Issue Resolution Agent (Vanilla Setting)

```
I have access to a python code repository in the directory {{ instance.repo_path }} . You can
      explore and modify files using the available tools. Consider the following issue
     description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>
Can you help me implement the necessary changes to the repository so that the requirements
     specified in the <issue_description> are met?
I've already taken care of all changes to any of the test files described in the <
     issue_description>. This means you DON'T have to modify the testing logic or any of the
     tests in any way! Also the development Python environment is already set up for you (i.e
     ., all dependencies already installed), so you don't need to install other packages.
     Your task is to make the minimal changes to non-test files in the {{ instance.repo_path
     }} directory to ensure the <issue_description> is satisfied.

Follow these phases to resolve the issue:
Phase 1. READING: read the problem and reword it in clearer terms
  1.1 If there are code or config snippets. Express in words any best practices or
       conventions in them.
  1.2 Hightlight message errors, method names, variables, file names, stack traces, and
       technical details.
  1.3 Explain the problem in clear terms.
  1.4 Enumerate the steps to reproduce the problem.
  1.5 Hightlight any best practices to take into account when testing and fixing the issue

Phase 2. RUNNING: install and run the tests on the repository
  2.1 Activate the environment by running ./opt/miniconda3/etc/profile.d/conda.sh ; conda
       activate testbed
  2.2 Follow the readme
  2.3 Install the environment and anything needed
  2.4 Iterate and figure out how to run the tests

Phase 3. EXPLORATION: find the files that are related to the problem and possible solutions
  3.1 Use `grep` to search for relevant methods, classes, keywords and error messages.
  3.2 Identify all files related to the problem statement.
  3.3 Propose the methods and files to fix the issue and explain why.
  3.4 From the possible file locations, select the most likely location to fix the issue.

Phase 4. TEST CREATION: before implementing any fix, create a script to reproduce and verify
     the issue.
  4.1 Look at existing test files in the repository to understand the test format/structure.
  4.2 Create a minimal reproduction script that reproduces the located issue.
  4.3 Run the reproduction script to confirm you are reproducing the issue.
  4.4 Adjust the reproduction script as necessary.

Phase 5. FIX ANALYSIS: state clearly the problem and how to fix it
  5.1 State clearly what the problem is.
  5.2 State clearly where the problem is located.
  5.3 State clearly how the test reproduces the issue.
  5.4 State clearly the best practices to take into account in the fix.
  5.5 State clearly how to fix the problem.

Phase 6. FIX IMPLEMENTATION: Edit the source code to implement your chosen solution.
  6.1 Make minimal, focused changes to fix the issue.

Phase 7. VERIFICATION: Test your implementation thoroughly.
  7.1 Run your reproduction script to verify the fix works.
  7.2 Add edge cases to your test script to ensure comprehensive coverage.
  7.3 Run existing tests related to the modified code to ensure you haven't broken anything.

8. FINAL REVIEW: Carefully re-read the problem description and compare your changes with the
     base commit {{ instance.base_commit }}.
  8.1 Ensure you've fully addressed all requirements.
  8.2 Run any tests in the repository related to:
    8.2.1 The issue you are fixing
    8.2.2 The files you modified
    8.2.3 The functions you changed
  8.3 If any tests fail, revise your implementation until all tests pass

Be thorough in your exploration, testing, and reasoning. It's fine if your thinking process
     is lengthy - quality and completeness are more important than brevity.
```

Figure 16: User prompt for OpenHands issue resolution agent in vanilla setup without augmenting it with any localization results (§6.1).

User Prompt for OpenHands Issue Resolution Agent Augmented with Locations Retrieved by CODESCOUT-14B (continued)

```
I have access to a python code repository in the directory {{ instance.repo_path }} . You can
      explore and modify files using the available tools. Consider the following issue
     description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>

A separate search subagent has already explored the repository to identify files, modules,
     and functions that are likely relevant to the issue.

<relevant_context>
{{ instance.search_results }}
</relevant_context>
Important notes about this context:
- The relevant context is intended to accelerate navigation, not replace your own judgment.
- The listed files/functions are very likely but not guaranteed to be sufficient.
- You may inspect files outside this list ONLY if needed, but you should prioritize the
     provided context.
- If you determine that some listed items are irrelevant or that additional files are
     required, state this explicitly in your analysis.

Can you help me implement the necessary changes to the repository so that the requirements
     specified in the <issue_description> are met?
I've already taken care of all changes to any of the test files described in the <
     issue_description>. This means you DON'T have to modify the testing logic or any of the
     tests in any way!
Also the development Python environment is already set up for you (i.e., all dependencies
     already installed), so you don't need to install other packages.
Your task is to make the minimal changes to non-test files in the {{ instance.repo_path }}
     directory to ensure the <issue_description> is satisfied.

Follow these phases to resolve the issue:

Phase 1. READING: read the problem and reword it in clearer terms
  1.1 If there are code or config snippets. Express in words any best practices or
       conventions in them.
  1.2 Hightlight message errors, method names, variables, file names, stack traces, and
       technical details.
  1.3 Explain the problem in clear terms.
  1.4 Enumerate the steps to reproduce the problem.
  1.5 Hightlight any best practices to take into account when testing and fixing the issue

Phase 2. RUNNING: install and run the tests on the repository
  2.1 Activate the environment by running
     ./opt/miniconda3/etc/profile.d/conda.sh ; conda activate testbed
  2.2 Follow the readme
  2.3 Install the environment and anything needed
  2.4 Iterate and figure out how to run the tests

Phase 3. CONTEXT REVIEW & CODE EXPLORATION: use the relevant context to orient yourself in
     the codebase.
  3.1 Review the <relevant_context> and understand why each listed file or function may be
       relevant.
  3.2 Prioritize and inspect the files and functions mentioned in the <relevant_context>.
  3.3 Read the relevant source code to understand its behavior.
  3.4 Identify discrepancies between current behavior and the expectations in the <
       issue_description>.
  3.5 Document any irrelevant items or missing files and adjust exploration as needed.

Phase 4. TEST CREATION: before implementing any fix, create a script to reproduce and verify
     the issue.
  4.1 Look at existing test files in the repository to understand the test format/structure.
  4.2 Create a minimal reproduction script that reproduces the located issue.
  4.3 Run the reproduction script to confirm you are reproducing the issue.
  4.4 Adjust the reproduction script as necessary.

Phase 5. FIX ANALYSIS: state clearly the problem and how to fix it
  5.1 State clearly what the problem is.
  5.2 State clearly where the problem is located.
  5.3 State clearly how the test reproduces the issue.
```

Figure 17: User prompt for OpenHands issue resolution agent augmented with locations retrieved by CODESCOUT-14B (§6.1).

User Prompt for OpenHands Issue Resolution Agent Augmented with Locations Retrieved by CODESCOUT-14B (continued)

```
  5.4 State clearly the best practices to take into account in the fix.
  5.5 State clearly how to fix the problem.

Phase 6. FIX IMPLEMENTATION: Edit the source code to implement your chosen
    solution.
  6.1 Make minimal, focused changes to fix the issue.

Phase 7. VERIFICATION: Test your implementation thoroughly.
  7.1 Run your reproduction script to verify the fix works.
  7.2 Add edge cases to your test script to ensure comprehensive coverage.
  7.3 Run existing tests related to the modified code to ensure you haven't
      broken anything.

Phase 8. FINAL REVIEW: Carefully re-read the problem description and compare
    your changes with the base commit {{ instance.base_commit }}.
  8.1 Ensure you've fully addressed all requirements.
  8.2 Run any tests in the repository related to:
    8.2.1 The issue you are fixing
    8.2.2 The files you modified
    8.2.3 The functions you changed
  8.3 If any tests fail, revise your implementation until all tests pass

Be thorough in your exploration, testing, and reasoning. It's fine if your
    thinking process is lengthy - quality and completeness are more important
    than brevity.
```

Figure 18: User prompt for OpenHands issue resolution agent augmented with locations retrieved by CODESCOUT-14B (§6.1).

User Prompt for OpenHands Issue Resolution Agent Augmented with Oracle Locations (continued)

```
I have access to a python code repository in the directory {{ instance.
    repo_path }} . You can explore and modify files using the available tools.
     Consider the following issue description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>

An oracle has already determined the exact set of files, modules and functions
    that need modification to resolve the above issue description.

<oracle_context>
{{ instance.search_results }}
</oracle_context>

Important notes about this context:
- The oracle context is guaranteed to be exhaustive and precise for resolving
    the issue.
- In cases where the context doesn't mention functions or modules, it means
    that the changes either belong to the global scope of the file or involve
    adding new functions/modules in the given file.
- You may inspect files outside this list ONLY if needed, but you primarily
    should focus on the provided context as that is the one which requires
    modification to resolve the issue.
- The provided context will not include files that need to be created from
    scratch for resolving the issue.

Can you help me implement the necessary changes to the repository so that the
    requirements specified in the <issue_description> are met?
I've already taken care of all changes to any of the test files described in
    the <issue_description>. This means you DON'T have to modify the testing
    logic or any of the tests in any way!
Also the development Python environment is already set up for you (i.e., all
    dependencies already installed), so you don't need to install other
    packages.
Your task is to make the minimal changes to non-test files in the {{ instance.
    repo_path }} directory to ensure the <issue_description> is satisfied.
```

Figure 19: User prompt for OpenHands issue resolution agent augmented with oracle locations (§6.1).

User Prompt for OpenHands Issue Resolution Agent Augmented with Oracle Locations

```
Follow these phases to resolve the issue:

Phase 1. READING: read the problem and reword it in clearer terms
  1.1 If there are code or config snippets. Express in words any best practices
       or conventions in them.
  1.2 Hightlight message errors, method names, variables, file names, stack
      traces, and technical details.
  1.3 Explain the problem in clear terms.
  1.4 Enumerate the steps to reproduce the problem.
  1.5 Hightlight any best practices to take into account when testing and
      fixing the issue

Phase 2. RUNNING: install and run the tests on the repository
  2.1 Activate the environment by running
     ./opt/miniconda3/etc/profile.d/conda.sh ; conda activate testbed
  2.2 Follow the readme
  2.3 Install the environment and anything needed
  2.4 Iterate and figure out how to run the tests

Phase 3. CONTEXT REVIEW & CODE EXPLORATION: use the oracle context to orient
    yourself in the codebase.
  3.1 Review the <oracle_context> and understand why each listed file or
      function needs modification.
  3.2 Prioritize and inspect the files and functions mentioned in the <
      oracle_context>.
  3.3 Read the relevant source code to understand its behavior.
  3.4 Identify discrepancies between current behavior and the expectations in
      the <issue_description>.

Phase 4. TEST CREATION: before implementing any fix, create a script to
    reproduce and verify the issue.
  4.1 Look at existing test files in the repository to understand the test
      format/structure.
  4.2 Create a minimal reproduction script that reproduces the located issue.
  4.3 Run the reproduction script to confirm you are reproducing the issue.
  4.4 Adjust the reproduction script as necessary.

Phase 5. FIX ANALYSIS: state clearly the problem and how to fix it
  5.1 State clearly what the problem is.
  5.2 State clearly where the problem is located.
  5.3 State clearly how the test reproduces the issue.
  5.4 State clearly the best practices to take into account in the fix.
  5.5 State clearly how to fix the problem.

Phase 6. FIX IMPLEMENTATION: Edit the source code to implement your chosen
    solution.
  6.1 Make minimal, focused changes to fix the issue.

Phase 7. VERIFICATION: Test your implementation thoroughly.
  7.1 Run your reproduction script to verify the fix works.
  7.2 Add edge cases to your test script to ensure comprehensive coverage.
  7.3 Run existing tests related to the modified code to ensure you haven't
      broken anything.

Phase 8. FINAL REVIEW: Carefully re-read the problem description and compare
    your changes with the base commit {{ instance.base_commit }}.
  8.1 Ensure you've fully addressed all requirements.
  8.2 Run any tests in the repository related to:
    8.2.1 The issue you are fixing
    8.2.2 The files you modified
    8.2.3 The functions you changed
  8.3 If any tests fail, revise your implementation until all tests pass

Be thorough in your exploration, testing, and reasoning. It's fine if your
    thinking process is lengthy - quality and completeness are more important
    than brevity.
```

Figure 20: User prompt for OpenHands issue resolution agent augmented with oracle locations (§6.1).