LocAgent: Graph-Guided LLM Agents for Code Localization

Anonymous ACL submission

Abstract

Code localization–identifying precisely where in a codebase changes need to be made–is a fundamental yet challenging task in software maintenance. Existing approaches struggle to efficiently navigate complex codebases when identifying relevant code snippets. The challenge lies in bridging natural language problem descriptions with the target code elements, often requiring reasoning across hierarchical structures and multiple dependencies. We introduce LocAgent, a framework that addresses code localization through a graph-guided agent. By parsing codebases into directed heterogeneous graphs, LocAgent creates a lightweight representation that captures code structures and their dependencies, enabling LLM agents to effectively search and locate relevant entities through powerful multi-hop reasoning. Experimental results on real-world benchmarks demonstrate that our approach significantly enhances accuracy in code localization. Notably, our method with the fine-tuned `Qwen-2.5-Coder-Instruct-32B` model achieves comparable results to SOTA proprietary models at greatly reduced cost (approximately 86% reduction), reaching up to 92.7% accuracy on file-level localization while improving downstream GitHub issue resolution success rates by 12% for multiple attempts (Pass@10). Our code is available at `https://github.com/gersteinlab/LocAgent`.

1 Introduction

Code localization can be viewed as an information retrieval (IR) task that aims to identify relevant code snippets given natural language descriptions (Suresh et al., 2024). Accurate code localization is crucial for software maintenance and evolution, as it identifies the precise locations for code modifications such as bug fixes, refactoring, and feature additions (Figure 1), thus streamlining the development workflow (Wang et al., 2024b). This capability has become fundamental for powerful AI assistants (OpenAI, 2023; Anthropic, 2023), code-aware search engines (PerplexityAI, 2023), and automated programming agents (Cognition.ai, 2024; Wang et al., 2025). However, understanding code for localization consumes up to 66% of developers' debugging time (Böhme et al., 2017), and remains difficult for automated tools (Kang et al., 2024; Qin et al., 2024). Unlike traditional retrieval tasks that primarily focus on lexical or semantic matching between queries and documents (Guo et al., 2016, 2020), code localization presents unique challenges. Specifically, it requires bridging the gap between natural language and programming languages, understanding structural and semantic properties of the codebase, and performing multi-step reasoning to analyze the issue (Xia et al., 2024; Zhang et al., 2024b).

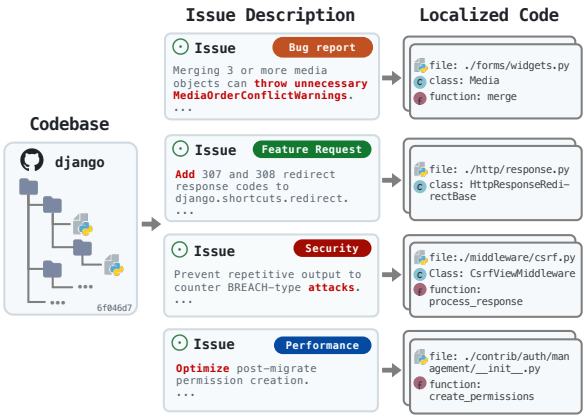

Figure 1: Code localization across four common programming scenarios. Given a codebase and an issue description, the goal of code localization is to identify the relevant code snippets that require modification to resolve the issue.

Existing approaches to code localization face significant limitations. Dense retrieval methods (Wang et al., 2023b; Günther et al., 2023) require maintaining and continuously updating vector representations of the entire codebase, creating engineering challenges for large and fast-evolving repositories.

Method	Relation Types				Node Types				Search/Traversal Strategy
	Contain	Import	Inherit	Invoke	Directory	File	Class	Function	
CodexGraph(Liu et al., 2024)	✔✗	✗	✔	✔	✗	✗	✔	✔	Cypher queries
RepoUnderstander(Ma et al., 2024)	✔	✗	✔	✔	✔	✔	✔	✔	MCTS
RepoGraph(Ouyang et al., 2025)	✔✗	✗	✔	✔	✗	✗	✔	✔	Ego-graph retrieval
OrcaLoca(Yu et al., 2025)	✔	✗	✗	✔✗	✔	✔	✔	✔	Simple search tools
LocAgent(Ours)	✔	✔	✔	✔	✔	✔	✔	✔	Unified retrieval tools

Table 1: Comparison of Graph-Based Code Representation Methods.

While LLMs demonstrate strong code understanding abilities (Kang et al., 2023; Wu et al., 2023), models with large context windows still cannot process entire codebases at once, necessitating strategic navigation. Moreover, issue descriptions often mention only symptoms rather than underlying causes. For instance, a report of 'XSS vulnerability in user profile' might require changes to a shared validation utility in the codebase, which do not explicitly mentioned in the issue. This disconnect between issue descriptions and affected code blocks presents a substantial challenge for traditional retrieval approaches, which struggle to trace implicit dependencies across the codebase. Recent agent-based methods attempt to address these limitations through iterative exploration (Yang et al., 2024; Wang et al., 2025). However, they still struggle to efficiently navigate and comprehend complex code structures and dependencies, particularly when multi-hop reasoning is required to trace from issue descriptions to affected code regions.

This raises a key question: *How can we design efficient, structure-aware indexing as intermediate representations that are both easy and performant for LLM agents to consume?* It is intuitive to design an agentic retrieval system that carefully combines traditional IR methods and LLM agent's reasoning ability to achieve accurate, efficient and cost-effective code localization in codebases.

To address this challenge, we propose LocAgent, a framework that builds directed heterogeneous graph indexing to unify code structures, dependencies, and contents. The indexing process typically takes only a few seconds per codebase, making it highly practical for real-time use. Our lightweight graph representation, coupled with sparse indexing techniques, enables efficient entity search while maintaining rich structural information. Moreover, the framework integrates a set of unified tools that empower the agent with efficient exploration capabilities and autonomous navigation based on contextual needs. Furthermore, by fine-tuning `Qwen-2.5-Coder- Instruct` (Hui et al., 2024) 7B and 32B models (abbreviated as `Qwen-2.5-7B` and `Qwen-2.5-32B` respectively), our framework achieves performance comparable to state-of-the-art models like `Claude-3-5-sonnet-20241022` (`Claude-3.5`) (Anthropic, 2023) while significantly reducing API costs by over 80% (from \$0.66 to \$0.09 per example), making it practical for real-world deployment.

Additionally, to facilitate a comprehensive evaluation of code localization methods, we introduce Loc-Bench, a new benchmark specifically designed for this task. Existing benchmarks like SWE-Bench(Jimenez et al., 2023) present significant limitations: (1) they risk contamination through data overlap with LLM training sets (Mündler et al., 2024), and (2) they primarily focus on bug fixing, lacking diversity in maintenance scenarios such as feature requests, performance optimizations, and security fixes. In contrast, Loc-Bench covers diverse scenarios and mitigates potential contamination concerns by incorporating more examples from popular Python repositories collected after known LLMs' training cutoff dates.

2 Related Work

2.1 Traditional Retrieval-based Methods

Traditional IR methods rely on lexical or semantic matching to return ranked lists of code snippets. Sparse retrievers, such as BM25 (Robertson et al., 1994), have demonstrated robustness to domain adaptation. Dense retrievers utilize embeddings for improved semantic searching, including models with open checkpoints such as general text embedding models E5-base-v2 (Wang et al., 2022) and proprietary APIs (VoyageAI, 2024). Code embedding models such as Jina-Code-v2 (Günther et al., 2023), Codesage-large-v2 (Zhang et al., 2024a), and CodeRankEmbed (Suresh et al., 2024), trained specifically for code related tasks, showing significant performance in Code2Code and NL2Code semantic search tasks.

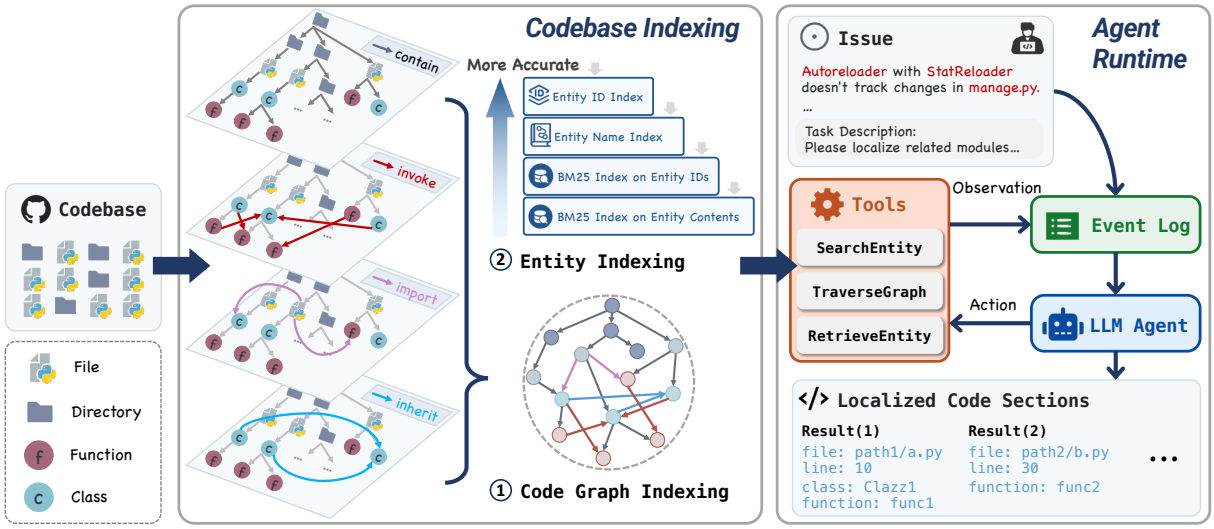

Figure 2: Overview of LocAgent framework. LocAgent first parses the given codebase to build a graph-based code representation with various entity and relation types. It then constructs sparse indexes for structure exploration and content search. Using these indexes, it performs agent-guided searches that combine the graph and tools.

2.2 LLM-based Generative Retrieval

Recently, LLMs with advanced code reasoning capabilities have demonstrated superior performance by directly processing queries and raw code for code localization (Kang et al., 2023; Wu et al., 2023; Xia et al., 2024). For example, Agentless (Xia et al., 2024), initially designed for automated program repair, uses a simplistic hierarchical localization process powered by LLM. It employs a straightforward three-phase approach that localizes relevant code sections before attempting to fix the given issues. Expanding on these techniques, agent-based methods utilize multi-step reasoning to enable automated codebase traversal. Specifically, OpenHands (Wang et al., 2025) implements a generalist coding agent that supports bash commands like `grep` and tools for viewing files. SWE-Agent (Yang et al., 2024) integrates a custom Agent-Computer Interface to support agents to navigate entire repositories. MoatlessTools (Örwall, 2024) combines an agentic searching loop and semantic search to obtain code locations.

2.3 Graph-based Code Representation

Due to the inherent structure of code, several works have employed graph-based representations to improve code understanding by capturing key relationships between components. Aider (2023) constructs a RepoMap and uses a graph ranking algorithm to identify the most significant contextual elements. Similarly, as a plugin, RepoGraph (Ouyang et al., 2025) performs subgraph retrieval–extracting ego-graphs with the search term in the centric–to provide structured context. CodexGraph (Liu et al., 2024) indexes the repository into a Neo4j graph database, where LLM agents query the database precisely using Cypher. These methods focus primarily on providing relevant context. In addition, RepoUnderstander (Ma et al., 2024) builds hierarchical and function-call graphs, using Monte Carlo Tree Search (MCTS) based repository exploration strategy to empower agents the ability of collecting repository-level knowledge. OrcaLoca (Yu et al., 2025) uses a simplified graph enhanced by priority scheduling and context pruning, which maintains efficient search. These methods build and utilize the code graph in various ways (Table 1).

3 The LocAgent Framework

We introduce LocAgent, a graph-guided LLM agent framework for code localization. Figure 2 illustrates the overall framework. Given a codebase, LocAgent can locate all the relevant code snippets at various granularities (file, class, function, or line level) for different types of GitHub issues through autonomous exploration and analysis.

3.1 Graph-based Code Representation

Codebases contain rich structural information, both explicit and implicit, that is essential for agent reasoning. Building on this insight, we develop a graph-based indexing that comprehensively captures codebase relationships while maintaining a granularity suitable for the agent to retrieve.

Code Graph Construction. We construct a heterogeneous directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ to index

the codebase, where $\mathcal{V} = \{v_i\}_{i=1}^{n}$ is the node set and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set. Each node $v \in \mathcal{V}$ and each edge $e \in \mathcal{E}$ has an associated type mapping function. For nodes, $\tau(v) : \mathcal{V} \rightarrow \mathcal{A}$ maps to types $\mathcal{A} = \{\texttt{directory}, \texttt{file}, \texttt{class}, \texttt{function}\}$. For edges, $\phi(e) : \mathcal{E} \rightarrow \mathcal{R}$ maps to relationships $\mathcal{R} = \{\texttt{contain}, \texttt{import}, \texttt{invoke}, \texttt{inherit}\}$. In this paper, we focus our study on Python repositories and leave codebases with other programming languages as future work. First, we include all directories and Python files as nodes. Then, we parse each Python file using the abstract syntax tree (AST) to identify inner functions and classes recursively as nodes. We define the function level as the finest node granularity and treat each function's code content as the document for agent retrieval.

As shown in Figure 2, all nodes with different types can be connected as a single tree using the *contain* relationship. This structure supports standard codebase-navigation operations. Our code graph further incorporates more advanced code interactions as edges: (1) the *invoke* relationship from function/class to function/class, where an invoke to a class represents class instantiation; (2) the *import* relationship from file to function/class; and (3) the *inherit* relationship between classes.

Sparse Hierarchical Entity Indexing. We treat nodes in our code graph as entities and build hierarchical indexing based on their contents. For each keyword, we lookup the indexes from top to bottom: (1) We build an entity ID index as a unique identifier for each node using its fully qualified name. For example, a function `calculate_sum` in the `MathUtils` class located in `src/utils.py` would be represented as: `src/utils.py:MathUtils.calculate_sum`. (2) We construct a global dictionary to map the entity name (e.g., `calculate_sum`) to all nodes that share the same name. (3) We index entity IDs through an inverted index (i.e., BM25) to handle keyword searches that do not exactly match the IDs or names of entities. (4) For cases where input keywords are not part of the entities' IDs (e.g., when a keyword refers to a global variable), we build an inverted index that maps code chunk(s) to each entity to cover all possible matches.

Remark. *Rather than relying solely on directory structures or hierarchical module indexing, our approach captures module dependencies that transcend directory boundaries. Two modules in distant directories (A and B) may appear unrelated in traditional navigation, but if they invoke each other or share inheritance, they're syntactically close in our graph representation. This syntactic proximity is essential for code localization because issues typically manifest through call relationships rather than directory structure.*

Tool Name	Input Params	Output
`SearchEntity`	*Keywords*	Related Entities with Code Snippets
`TraverseGraph`	*Start Entity IDs* *Direction* *Traverse Hops* *Entity Types* *Relation Types*	Traversed Subgraph, including Entites and Relations
`RetrieveEntity`	*Entity IDs*	Complete Code of Specified Entities

Table 2: List of unified APIs provided by LOCAGENT for code search and exploration.

3.2 Agent-guided Code Search

During runtime, LOCAGENT takes issue descriptions as input and launches agents that autonomously use predefined tools based on graph indexing to localize target code. While the agent may iteratively invoke multiple tools internally to explore the codebase, LOCAGENT offers a simplified interface to users, requiring only a single-turn interaction—users submit an issue statement and receive localization results without additional input. This autonomous, self-contained workflow makes LOCAGENT easy to deploy and highly practical for real-world use—with no frequent and computationally intensive code embedding generalization required beforehand.

Tool Design for Codebase Exploration. Recent works (Xie et al., 2024; Wu et al., 2024), inspired by GUI-based IDEs, have developed numerous specialized tools for agents to explore codebases. However, since these tools are primarily designed for human readability, they sacrifice the compactness and efficiency preferred by LLM agents. Building upon our graph-based code representation, we can develop tools that support efficient higher-order codebase exploration to address these challenges. We unify all codebase navigation, search, and view operations into three tools (Table 2), introduced as follows.

`SearchEntity`: This tool searches codebases using keywords to locate relevant entities through our hierarchical entity index. When an exact match is not found in the upper index, the system performs a further search using the lower index. For each en-

tity found, we return its code snippet in three detail levels: `fold`, `preview`, and `full code` (Figure 6). This effectively prevents lengthy code context and reduces noise fed into agents.

`TraverseGraph`: This tool performs a type-aware breadth-first search (BFS) on the code graph, starting from input entities and allowing control over both traversal direction and number of hops. This supports agents to perform arbitrary multi-hop codebase navigation through only one action, significantly improving the efficiency compared with existing agent systems. Note that by allowing agents to select entity types and relation types for each traversal, this tool effectively leverages the LLM agents' coding expertise to generate proper meta paths—a crucial element for heterogeneous graph analysis (Lv et al., 2021). For example, by specifying entity types to {`class`, `function`} and relation types to {`contain`, `inherit`}, this tool returns the UML diagram. Additionally, we design an expanded tree-based format for the output subgraph that encodes both relation types and directions (Figure 7). For detailed comparisons with alternative graph formats, please see §A.1.2.

`RetreiveEntity`: This tool retrieves complete entity attributes for each input entity ID, including essential information such as file path, line number, and code content.

Chain-of-Thought Agent Planning. We use chain-of-thought (CoT) prompting (shown in §A.3) to guide the agent in solving code localization problems step by step. The agent systematically follows these steps: (1) *Keyword extraction.* The agent begins by breaking down the issue statement into different categories and then extracts relevant keywords that are closely related to the problem. (2) *Linking keywords to code entities.* The agent invokes `SearchEntity` to clarify each extracted keyword. (3) *Generate the logical flow from fault to failure.* The agent first identifies the entry points that trigger the problem. Then, it iteratively traverse the codebase with `TraverseGraph`, retrieves code contents with `RetrieveEntity`, and searches new keywords with `SearchEntity`. Finally, it generates the logic flow based on the issue and additional context. (4) *Locate the target entities.* The agent pinpoints all suspicious code entities that need modification based on the logic flow. Then, it ranks these entities based on their relevance.

Confidence Estimation Based on Consistency. After generating a complete ranked list of candidate entities, to obtain a more consistent ranking, we measure the consistency (Wang et al., 2023a) of the LLM's predictions across multiple iterations. Specifically, we use the Reciprocal Rank as the initial confidence score for each predicted location. We then aggregate the scores for each entity across iterations to compute its final confidence score. The intuition behind this approach is that if the LLM consistently ranks a location higher in multiple iterations, it is more likely to be relevant.

3.3 Open-source Model Fine-tuning

Given the high costs of proprietary LLM APIs and data security concerns, we fine-tune open-source models to improve their code localization capabilities and enable local deployment. We first collecte 433 successful trajectories from `Claude-3.5`, then fine-tuned `Qwen2.5-32B` using this data. Due to budget constraints, an additional 335 successful trajectories are sampled from the fine-tuned `Qwen2.5-32B`, which is used to further refine the 32B model for self-improvement. The entire dataset, combining both successful `Claude-3.5` and `Qwen2.5-32B` trajectories, are then used to train a smaller 7B model.

Both models are fine-tuned via Supervised Fine-Tuning (SFT) with LoRA (Hu et al., 2021), using a standard next-token cross-entropy loss. See §C.3 for more training details.

4 LOC-BENCH: A New Benchmark for Code Localization

4.1 Revisiting Existing Benchmark

SWE-Bench (Jimenez et al., 2023) is a widely used benchmark that collects GitHub issues and corresponding code patches that resolve them. Xia et al. (2024) and Suresh et al. (2024) adapt its subset, SWE-bench Lite, for code localization, treating the patched files and functions as targets. However, existing datasets (Tomassi et al., 2019; Jimenez et al., 2023) present challenges for effectively evaluating code localization methods. First, they are at risk of contamination, as they may include data overlapping with repositories or issues used by modern models during pre-training. Second, existing datasets are not specifically designed for code localization. SWE-Bench, for instance, was created primarily to evaluate end-to-end bug-fixing capabilities, with localization being only an implicit intermediate step. This focus results in datasets dominated by bug reports (85% of SWE-Bench Lite examples) while severely under-representing

Dataset	Category	#Sample
SWE-Bench Lite (Total = 300)	Bug Report	254
	Feature Request	43
	Security Issue	3
	Performance Issue	0
Loc-Bench (Totoal = 560)	Bug Report	242
	Feature Request	150
	Security Issue	29
	Performance Issue	139

Table 3: Distribution of samples across different categories in the SWE-Bench Lite and Loc-Bench datasets.

other common software maintenance tasks such as security and performance issues. This imbalance fails to capture the diverse localization challenges faced in real-world software development.

4.2 Dataset Construction

To address the limitations of existing benchmarks, we introduce LOC-BENCH, a new dataset specifically designed for code localization. We collects up-to-date issues from Python repositories to mitigate potential contamination concerns in the latest LLMs. Additionally, LOC-BENCH covers wider categories, including bug reports, feature requests, security, and performance issues, enabling a more comprehensive evaluation of code localization methods. The statistics of LOC-BENCH are shown in Table 3.

For the Bug Report category, we collect GitHub issues created after October 2024, which is later than the release dates of most modern LLMs. To enrich the dataset with more instances of security and performance issues, we use the GitHub Search API to search for relevant keywords, such as "latency improvement" for performance-related issues. We exclude instances modifying more than five Python files or more than ten functions in the corresponding patch. For further details, see §B.

5 Experiments

Our experiments evaluate four key aspects of LOCAGENT: (1) the effectiveness of our graph-guided agent for code localization compared to existing methods, (2) the performance of fine-tuned open source models as cost-effective alternatives to proprietary LLMs, (3) a detailed analysis of our framework's and models' robustness and generalization, and (4) the contribution of each component in our framework through ablation studies. We also examine the impact of improved localization on downstream software maintenance tasks.

5.1 Experimental Settings

Datasets. We conduct experiments on both SWE-Bench Lite and our introduced LOC-BENCH. For SWE-Bench Lite, following Suresh et al. (2024), we treat the patched files and functions as localization targets and exclude examples where no existing functions were modified, ultimately retaining 274 out of the original 300 examples.

Metrics. To assess performance, we adopt a strict top-k accuracy metric inspired by R-Precision in information retrieval. For each example, we consider a localization successful only if *all* relevant locations are correctly identified within the top-k predictions. This evaluation measures the ability to identify all code sections that require modification. We report results across multiple k values: file-level localization using Acc@1, Acc@3, and Acc@5, and function-level localization using Acc@5 and Acc@10. To offer a more relaxed evaluation, we also assess module-level localization, which considers the prediction correct if any function within the patched class is identified.

5.2 Baselines

We evaluate LOCAGENT against three categories of competitive baselines: (a) Retrieval-based methods: including the sparse retriever BM25 (Robertson et al., 1994), the general-purpose embedding model E5-base-v2 (Wang et al., 2022), and several code-specific models such as Jina-Code-v2 (Günther et al., 2023), Codesage-large-v2 (Zhang et al., 2024a), and the current SOTA CodeRankEmbed (Suresh et al., 2024). Proprietary embeddings are excluded due to API cost. (b) Procedure-based methods: Agentless (Xia et al., 2024), which localizes code via a hierarchical procedure. (c) Agent-based methods: advanced agent frameworks for code exploration and modification, including OpenHands (Wang et al., 2025), SWE-Agent (Yang et al., 2024), and MoatlessTools (Örwall, 2024). For implementation details, see §C.1.

5.3 Evaluation Results on SWE-Bench Lite

As shown in Table 4, agent-based methods consistently outperform other approaches, and our method shows competitive performance by achieving best results across all levels of code localization. Unlike traditional retrieval-based methods, Agent-

Type	Method	Loc-Model	File (%)			Module (%)		Function (%)	
			Acc@1	Acc@3	Acc@5	Acc@5	Acc@10	Acc@5	Acc@10
Embedding-Based	BM25 (Robertson et al., 1994)		38.69	51.82	61.68	45.26	52.92	31.75	36.86
	E5-base-v2 (Wang et al., 2022)		49.64	74.45	80.29	67.88	72.26	39.42	51.09
	Jina-Code-v2 (Günther et al., 2023)		43.43	71.17	80.29	63.50	72.63	42.34	52.19
	Codesage-large-v2 (Zhang et al., 2024a)		47.81	69.34	78.10	60.58	69.71	33.94	44.53
	CodeRankEmbed (Suresh et al., 2024)		52.55	77.74	84.67	71.90	78.83	51.82	58.76
Procedure-Based	Agentless (Xia et al., 2024)	GPT-4o	67.15	74.45	74.45	67.15	67.15	55.47	55.47
		Claude-3.5	72.63	79.20	79.56	68.98	68.98	58.76	58.76
Agent-Based	MoatlessTools (Örwall, 2024)	GPT-4o	73.36	84.31	85.04	74.82	76.28	57.30	59.49
		Claude-3.5	72.63	85.77	86.13	76.28	76.28	64.60	64.96
	SWE-agent (Yang et al., 2024)	GPT-4o	57.30	64.96	68.98	58.03	58.03	45.99	46.35
		Claude-3.5	77.37	87.23	90.15	77.74	78.10	64.23	64.60
	Openhands (Wang et al., 2025)	GPT-4o	60.95	71.90	73.72	62.41	63.87	49.64	50.36
		Claude-3.5	76.28	89.78	90.15	83.21	83.58	68.25	70.07
	LocAgent (Ours)	Qwen2.5-7B(ft)	70.80	84.67	88.32	81.02	82.85	64.23	71.53
		Qwen2.5-32B(ft)	75.91	90.51	92.70	85.77	87.23	71.90	77.01
		Claude-3.5	**77.74**	**91.97**	**94.16**	**86.50**	**87.59**	**73.36**	**77.37**

Table 4: Performance comparison with baselines on code localization on SWE-bench lite. Results show accuracy at file, module, and function levels. For agent-based methods, we use GPT-4o (GPT-4o-2024-0513) and Claude-3.5 as the localization model; performance of our fine-tuned open-source models is included for comparison.

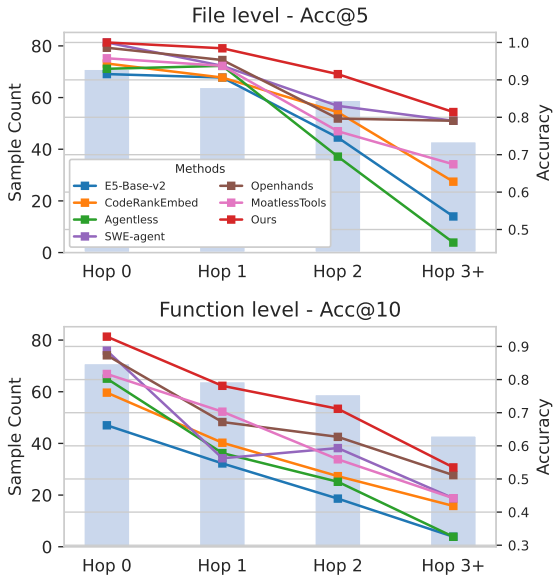

Figure 3: Performance analysis at different difficulty levels for file- and function-level localization. All agent-based methods and Agentless use Claude-3.5 as the localization model. *Hop N* refers to the distances between functions mentioned in the issue description and the ground truth patch on our code graph.

less identifies only limited locations due to its narrow repository scope, which hinders performance gains when considering a broader set of candidates. The NDCG results are presented in Table 11 in the appendix.

To further analyze the results, we examine performance across different task difficulty levels. We measure the task difficulty by calculating the shortest hops between the functions mentioned in the issue descriptions and the patched functions on our code graph (See §C.2 for more details). As shown in Figure 3, performance decreases for all methods as the task becomes more challenging. However, agent-based methods demonstrate better robustness as difficulty increases, with our method maintaining competitive performance across difficulty levels. Retrieval-based methods, such as E5-Base-v2 and CodeRankEmbed, perform poorly at the function level, even when the patched functions are explicitly mentioned in the query. This is because they treat the query as a whole, failing to capture fine-grained details. Agentless performs even worse than retrieval-based methods when exploration beyond the query is needed (*hop* ≥ 1) due to its simplistic localization process and limited view focused only on the repository structure.

5.4 Fine-tuned Open-source Models

Figure 4 shows that after fine-tuning, both the 7B and 32B models show significant improvements in this task. LocAgent with Qwen-2.5-32B(ft) achieves performance comparable to Claude-3.5, and LocAgent with Qwen2.5-7B(ft) also delivers results on par with that obtained using GPT-4o. As shown in Table 4, our method with Qwen2.5-32B(ft) outperforms nearly all baselines, including those that use larger and more powerful LLMs. The original 7B model performs poorly due to its limited tool-use capability (Chen et al., 2024). These results validate the feasibility of deploying our fine-tuned open-source models as promising alternatives to proprietary APIs, especially in resource-constrained scenarios.

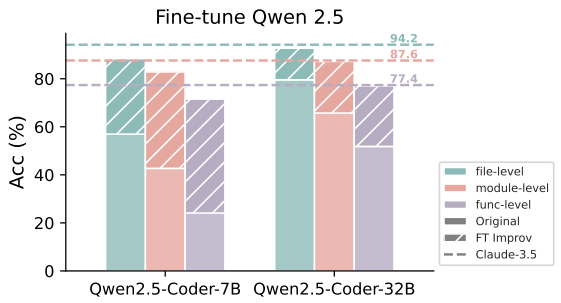

Figure 4: Performance comparison of original and fine-tuned Qwen models. Metrics include file-level Acc@5 and module/function-level Acc@10. Dashed lines indicate the performance of `Claude-3.5` for reference.

Method	LM	#Round	Cost($)	Acc@10 / Cost
MoatlessTools	`GPT-4o`	5	0.46	1.3
	`Claude-3.5`	5	0.46	1.4
SWE-agent	`GPT-4o`	8	0.56	0.8
	`Claude-3.5`	9	0.67	1.0
Openhands	`GPT-4o`	15	0.83	0.6
	`Claude-3.5`	13	0.79	0.9
Ours	`Claude-3.5`	7	0.66	1.2
	`Qwen2.5-7B(ft)`	6	0.05	**13.2**
	`Qwen2.5-32B(ft)`	9	0.09	8.6

Table 5: Efficiency analysis of average cost and agent interaction rounds across methods. Cost-efficiency is measured by the ratio of function-level Acc@10 to average cost.

5.5 Efficiency Analysis

Table 5 presents an efficiency analysis comparing agent-based methods in terms of cost and the number of agent interactions required. MoatlessTools demonstrates good cost-efficiency and requires relatively fewer rounds of interaction. However, the dense embeddings it uses make it difficult and slow to adapt to fast-evolving codebases. SWE-agent and Openhands also show moderate costs but still do not match the efficiency of LocAgent. For LocAgent with `Claude-3.5`, although more rounds of interaction are required, the cost remains lower than that of Openhands, illustrating the token efficiency of our tools' outputs. LocAgent with fine-tuned Qwen models stands out for its superior efficiency[1]. `Qwen2.5-7B(ft)` is the most cost-efficient option, requiring only $0.05 per example, while `Qwen2.5-32B(ft)` offers a more cost-effective alternative to `Claude-3.5`. These results highlight the potential of fine-tuned open-source models as efficient alternatives, providing an optimal balance of cost and performance.

[1]We calculate the cost based on the prices from AI inference providers (Hyperbolic, 2025; artificialanalysis.ai, 2025). Specifically, for the `Qwen2.5-32B(ft)` model, the cost we use is $0.20/1M tokens for both input and output. For the `Qwen2.5-7B(ft)` model, the cost we use is $0.14/1M tokens for input and $0.28/1M tokens for output.

Model Setting	File Acc@5	Module Acc@10	Function Acc@10
Ours	88.32	82.85	71.53
w/o `TraverseGraph`	86.13	78.47	66.06
Relation Types: *contain*	86.50	79.56	66.42
Traverse Hops: 1	86.86	80.29	66.79
w/o `RetrieveEntity`	87.59	81.39	69.34
w/o `SearchEntity`	68.98	61.31	53.28
w/o BM25 index	75.18	68.98	60.22

Table 6: Ablation study of our framework. Metrics are file-level Acc@5, module/function-level Acc@10.

5.6 Ablation Study

We conduct an ablation study to evaluate the effectiveness of each component of our toolsets. Due to budget constraints, we use `Qwen-2.5-7B(ft)` as the localization model for these experiments.

(1) *Each tool plays a critical role.* As shown in Table 6, removing any tool, especially `SearchEntity`, leads to varying degrees of accuracy degradation, particularly in module and function level localization.

(2) Graph structure provides essential information for localization. Removing `TraverseGraph` tool decreases module- and function-level performance since the agent lacks structure information and must infer relationships through reasoning. Adding only the *contain* relationship yields marginal improvements, emphasizing the importance of the other three types of relationship and explaining why our method surpasses baselines relying only on the repository structure.

(3) Multi-hop exploration is crucial for deep code understanding. Compared to the full setting, fixing *Hops=1* leads to a more significant decrease in function-level accuracy, underscoring the importance of multi-hop exploration for identifying relevant entities.

(4) Sparse indexing is essential for performance. Removing `SearchEntity` tool, or even partial removal of its index, causes a substantial drop in performance across all metrics. This demonstrates the effectiveness of building a sparse index on our code graph for improving localization performance.

5.7 Evaluation Results on Loc-Bench

To evaluate robustness and generalization of our methods and fine-tuned models, we test on our new dataset. Since Loc-Bench includes exam-

Method	Loc-Model	File (%)		Module (%)		Function (%)	
		Acc@5	Acc@10	Acc@10	Acc@15	Acc@10	Acc@15
IR-Based	CodeRankEmbed	74.29	80.89	63.21	67.50	43.39	46.61
Agentless	`Claude-3.5`	67.50	67.50	53.39	53.39	42.68	42.68
OpenHands	`Claude-3.5`	79.82	80.00	68.93	69.11	59.11	59.29
SWE-agent	`Claude-3.5`	77.68	77.68	63.57	63.75	51.96	51.96
LocAgent (Ours)	`Qwen2.5-7B(ft)`	78.57	79.64	63.04	63.04	51.43	51.79
	`Claude-3.5`	**83.39**	**86.07**	**70.89**	**71.07**	**59.29**	**60.71**

Table 7: Performance evaluation on the real-world LocBench dataset.

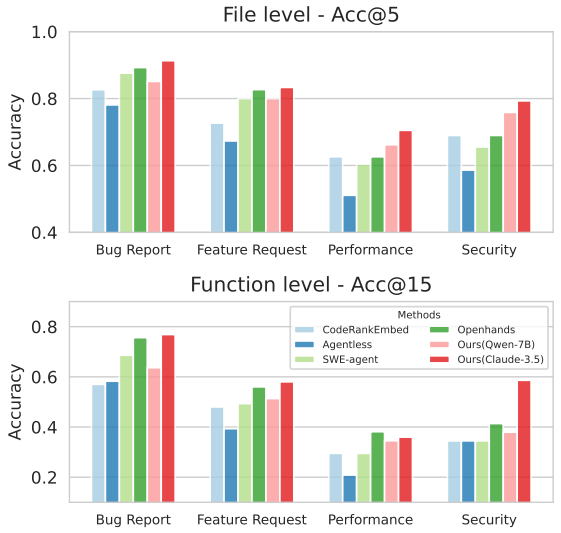

Figure 5: Performance analysis across different categories for file- and function-level localization. All agent-based baselines and Agentless use `Claude-3.5` as the localization model.

Method	Localization LM	Acc@5	Pass@1	Pass@10
Agentless	`Claude-3.5`	58.39	26.31	33.58
Ours	`Qwen2.5-32B(ft)`	69.34	26.79	36.13
	`Claude-3.5`	**73.36**	**27.92**	**37.59**

Table 8: Impact of localization accuracy on downstream bug repair tasks.

ples editing 1 to 5 files, we assess file localization at top-5 and top-10 ranks, and function/module localization at top-10 and top-15 ranks.

Table 7 shows that `Qwen2.5-7B(ft)` exhibits strong generalization capabilities, maintaining competitive performance compared to SWE-agent using more expensive and strong model. These results highlight the practicality of `Qwen2.5-7B(ft)` model for real-world applications. Despite being an open-source alternative, it achieves a performance comparable to `Claude-3.5`. We also evaluate performance across four categories. Figure 5 clearly shows that our method outperforms others in almost all categories of code localization. However, it also highlights a noticeable decrease in performance across the other three categories compared to the Bug Report category. This performance gap likely stems from the distribution of our training data, which contained more bug report examples, potentially leading to scaffolds better optimized for bug localization tasks. This trend suggests that, while our method is highly effective for bug report localization, there is still room for improvement in handling the other categories through category-specific optimization strategies and more balanced training data.

5.8 Application: Better Localization Leads to More Solved GitHub Issues

To assess the impact of localization methods on downstream tasks, we evaluated their effectiveness in solving GitHub issues. We choose Agentless as the baseline, ranking among the top-performing open-source submissions on SWE-Bench Lite. For consistency, we utilized `Claude-3.5` as the editing model in conjunction with the Agentless editing method. Table 8 shows that the success rate for solving GitHub issues improves significantly with better code localization accuracy.

6 Conclusion

In conclusion, LocAgent enhances code localization by parsing codebases as graphs, enabling efficient repository-level exploration for LLM agents. With fine-tuned open-source models, our method achieves high localization accuracy while greatly reducing costs compared to larger proprietary models. Experimental results demonstrate the effectiveness of LocAgent in identifying relevant code blocks and supporting downstream tasks.

Limitations

First, our study primarily focused on fine-tuning `Qwen-2.5-Coder` models. Exploring a broader

range of base models, including other open-source LLMs like CodeLlama, Mistral, or Yi, could provide valuable insights into model selection trade-offs. Additionally, investigating different fine-tuning approaches beyond LoRA, such as full fine-tuning or other parameter-efficient methods, could potentially yield better performance.

Moreover, though we demonstrated improved bug repair performance with better localization, we only scratched the surface of potential downstream applications. Future work should evaluate LocAgent's impact on other software engineering tasks like refactoring, feature addition, security vulnerability patching, and performance optimization. This would provide a more comprehensive understanding of the practical utility of the framework.

Finally, the current evaluation focuses primarily on Python codebases. Extending LocAgent to support other programming languages and evaluating its performance across different language paradigms would better demonstrate its generalizability.

References

Aider. 2023. Building a better repository map with tree sitter. Accessed: April 15, 2025.

Anthropic. 2023. Claude: Conversational ai by anthropic. Accessed: January 21, 2025.

artificialanalysis.ai. 2025. Artificial analysis. https://artificialanalysis.ai/models/. Accessed: 2025-04-28.

Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 117–128.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, et al. 2024. T-eval: Evaluating the tool utilization capability of large language models step by step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9510–9529.

Cognition.ai. 2024. Introducing devin, the first ai software engineer.

Gangda Deng, Ömer Faruk Akgül, Hongkuan Zhou, Hanqing Zeng, Yinglong Xia, Jianbo Li, and Viktor Prasanna. 2023. An efficient distributed graph engine for deep learning on graphs. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 922–931, New York, NY, USA. Association for Computing Machinery.

John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2002. Graphviz—open source graph drawing tools. In *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*, pages 483–484. Springer.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2023. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560*.

Jiafeng Guo, Yixing Fan, Qingyao Ai, and W Bruce Croft. 2016. A deep relevance matching model for ad-hoc retrieval. In *Proceedings of the 25th ACM international on conference on information and knowledge management*, pages 55–64.

Jiafeng Guo, Yixing Fan, Liang Pang, Liu Yang, Qingyao Ai, Hamed Zamani, Chen Wu, W Bruce Croft, and Xueqi Cheng. 2020. A deep look into neural ranking models for information retrieval. *Information Processing & Management*, 57(6):102067.

Michael Günther, Louis Milliken, Jonathan Geuter, Georgios Mastrapas, Bo Wang, and Han Xiao. 2023. Jina embeddings: A novel set of high-performance sentence embedding models. *Preprint*, arXiv:2307.11224.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Hyperbolic. 2025. Hyperbolic website. https://hyperbolic.xyz/. Accessed: 2025-04-15.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Sungmin Kang, Gabin An, and Shin Yoo. 2023. A preliminary evaluation of llm-based fault localization. *arXiv preprint arXiv:2308.05487*.

Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2024. Codexgraph: Bridging large language models and code repositories via code graph databases. *Preprint*, arXiv:2408.03910.

Qingsong Lv, Ming Ding, Qiang Liu, Yuxiang Chen, Wenzheng Feng, Siming He, Chang Zhou, Jianguo Jiang, Yuxiao Dong, and Jie Tang. 2021. Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 1150–1160.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv e-prints*, pages arXiv–2406.

Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887.

OpenAI. 2023. Chatgpt: Language model by openai. Accessed: January 21, 2025.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. Repograph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*.

PerplexityAI. 2023. Perplexity ai: An ai-powered search engine. Accessed: January 21, 2025.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. 1994. Okapi at trec-3. In *Text Retrieval Conference*.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2024. Cornstack: High-quality contrastive data for better code ranking. *arXiv preprint arXiv:2412.01007*.

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. *Preprint*, arXiv:1903.06725.

VoyageAI. 2024. Voyage-code-2: Elevate your code retrieval. Accessed: 2024-02-02.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024b. Coderag-bench: Can retrieval augment code generation? *Preprint*, arXiv:2406.14497.

Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276*.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*.

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350*.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024a. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

Albert Örwall. 2024. Moatless tools.

A LocAgent Design Details

A.1 Tool Output Design

A.1.1 Three-level format for `SearchEntity` output

Once invoked by the LLM agent, the retrieval APIs search for files, classes, methods, and code snippets in the codebase, and return the results back to the agent. To avoid forming a very lengthy code context that may contain noisy information to LLM, we return only necessary information as API outputs. To achieve this, we designed four granular standard output formats (Figure 6): `fold`, `preview`, `full code`. Specifically, the API adapts its output based on the number and size of retrieved entities: when the number of matched entities is small (≤ 3), it returns their `full code`; for large files, it provides a `preview`, summarizing the module structure (e.g., class/method signatures); and when the number of matches exceeds 3, it outputs a compact `fold` format that only lists entity identifiers. This design ensures that the agent receives sufficient context without overwhelming its input window.

A.1.2 Tree-based Subgraph Formatting for `TraverseGraph` Output

The `TraverseGraph` tool performs breadth-first traversal over the code graph, returning a local subgraph for each input entity. These subgraphs help the agent reason about complex dependencies surrounding the entity. However, reasoning over graphs remains a challenge for LLMs. Prior work (Fatemi et al., 2023) shows that LLM performance can vary significantly depending on how graphs are encoded into text, making the design of the subgraph output format a critical factor. We leave the exploration of applying more advanced and efficient graph traversal strategies (Deng et al., 2023) for LLM Agents as future work.

We have developed a new tree-based format, shown in Figure 7, with several features that enhance LLM reasoning: (1) We represent subgraphs as trees, allowing LLMs to use indentation to determine a node's distance from the root, (2) We display complete entity IDs for each node (e.g., `django/core/validators.py:RegexValidator`) to help LLMs locate nodes easily, and (3) We explicitly specify relation types for each edge, including reversed relations

To evaluate how different graph formats impact code localization performance, we conducted an experiment using 37 challenging samples from SWE-Bench-Lite. These samples were considered "challenging" because they could not be solved by any baseline agent methods. Using `Claude-3.5` as the Localization Model across all settings, we compared various output formats. Table 9 presents our findings. The baseline output formats we tested are described below:

- **row**: For each line, list one row of the adjacency matrix. For example,

  function "fileA.py:funcA" invokes function "fileA.py:funcB", "fileA.py:funcC"

- **row (w/ entity attributes)**: Additionally include entity attributes (e.g., code content) for format **row**.

- **incident**: The incident format mentioned in (Fatemi et al., 2023). An integer instead of entity ID is used to represent each node. For example,

  Map function "fileA.py:funcA" to index 0. Map function "fileA.py:funcB" to index 1. Map function "fileA.py:funcC" to index 2.

  function 0 invokes function 1,2.

- **Graphviz DOT**: Represent graph in Graphviz DOT language (Ellson et al., 2002).

- **JSON**: Expand the subgraph as a tree, and convert it to JSON format.

As shown in Table 9, expanding subgraphs as trees (i.e., **JSON**, **tree-based**) can significantly improve the performance. Our **tree-based** format achieves the best overall performance across different levels of localization tasks. We also test returning entity attributes along with subgraphs. We notice that **row (w/ entity attributes)** consistently underperforms **row**, indicating the attributes for all the nodes may be very noisy. Besides, although using incident format can simplify the output and show improvements in file-level localization, it degradation the module- and file-level localization.

A.2 Implementation

To enable the LLM agent to invoke the Code Localization APIs, we handle the interaction differently based on the LLM's capabilities. For LLMs that support tool-calling features, we define the tools as a list of JSON objects, which are then used as parameters for the API calls. For LLMs that do

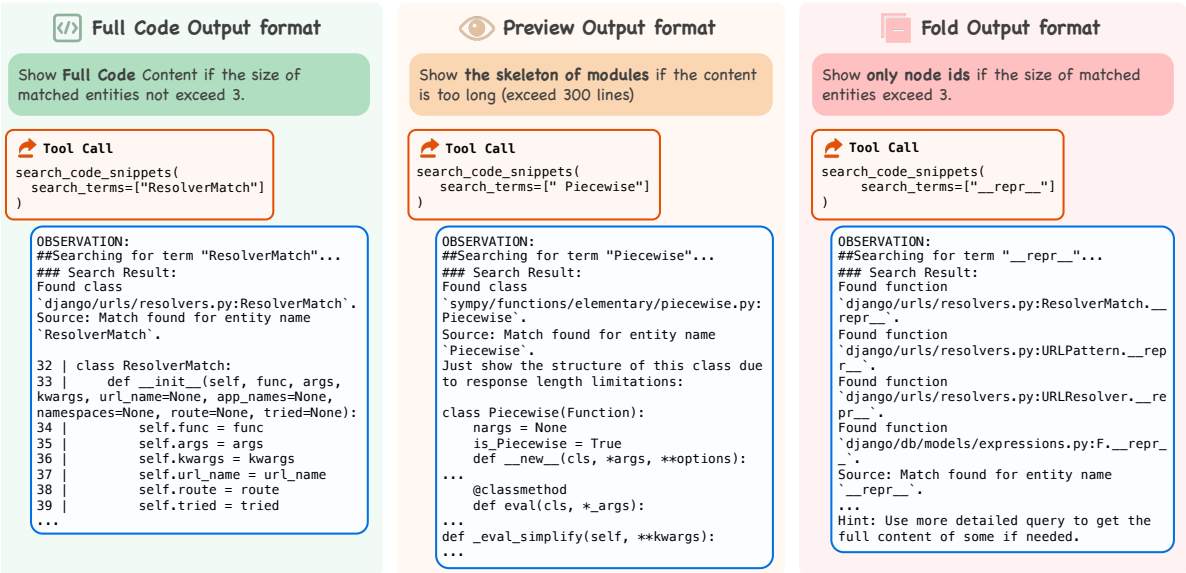

Figure 6: Different output formats designed for efficient agent-code interaction. Left: Full code output when matched entities ≤ 3. Middle: Preview output showing module skeleton for large files. Right: Fold output showing only entity IDs when matches > 3.

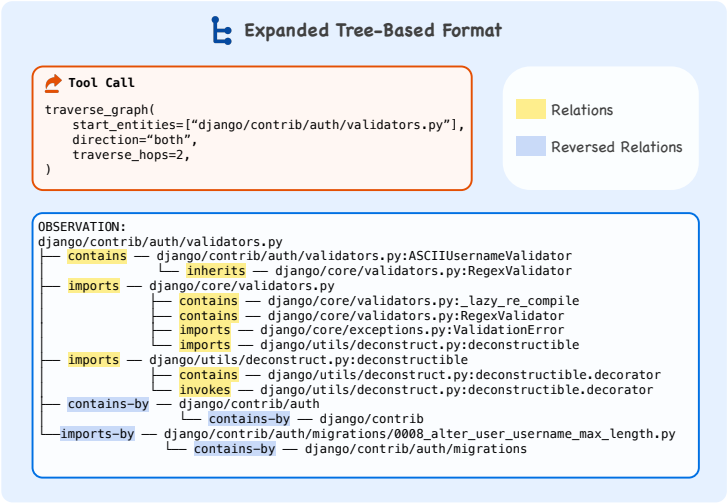

Figure 7: A truncated example of the expanded tree-based format for the output subgraph of tool `TraverseGraph`.

not support tool-calling (such as Qwen), we provide the description of the API and the expected output as part of the LLM's prompt. When the agent decides to invoke a set of retrieval APIs, it responds with a list of API call names and their corresponding arguments. These retrieval API requests are processed locally by searching over the built code graph. The results from executing these APIs locally are returned to the agent.

By default, we query the LLM with a temperature setting of 1.0. We conduct **two** interactions, after which we rerank the results based on mean reciprocal rank (MRR) scores. We also leverage multiprocess execution to speed up the process. Since all of our tools are read-only, LocAgent does not require a specialized Docker environment to operate.

A.3 Prompt

The prompt for LocAgent is shown in Figure 8.

A.4 Supported Programming Languages

While our current implementation focuses on Python codebases, our method is not inherently tied to Python. LocAgent constructs a lightweight graph representation of the codebase, which is built from abstract syntax trees (ASTs) generated from the source code. Specifically, we employ *tree-sitter*

Output Format	File(%)			Module(%)		Function(%)	
	Acc@1	Acc@3	Acc@5	Acc@5	Acc@10	Acc@5	Acc@10
row	41.18	67.65	70.59	61.76	61.76	35.29	38.24
row (w/ entity attributes)	41.18	64.71	64.71	50.00	50.00	32.35	32.35
incident	41.18	70.59	73.53	55.88	55.88	29.41	32.35
Graphviz DOT	41.18	73.53	**82.35**	64.71	64.71	35.29	35.29
JSON	41.18	67.65	76.47	**67.65**	**70.59**	**38.24**	**41.18**
tree-based (Ours)	**47.06**	**79.41**	79.41	64.71	64.71	**38.24**	**41.18**

Table 9: Localization performance under different `TraverseGraph` output formats.

library to generate ASTs, and *tree-sitter* supports a wide range of programming languages beyond Python, including C, C++, Java, Go, and many others. Thus, extending our approach to other object-oriented or procedural languages would be relatively straightforward.

B Dataset construction details

We provide a more thorough clarification of the dataset construction process below:

Issue Sourcing. For the Bug Report and Feature Request categories, we select open-source Python repositories on GitHub with over 5,000 stars to ensure code quality and project reliability. We then collect associated pull requests (PRs), each associated with a codebase specified by its base commit following (Jimenez et al., 2023). To gather security and performance issues, we use the GitHub Search API with targeted keywords (listed in Table 10) to identify relevant PRs.

Issue Filtering. We filter out PRs that do not explicitly resolve any linked GitHub issue. In addition, we exclude large-scope changes involving more than 5 Python files or more than 10 functions, as such large-scale changes are often too broad and may not represent a single, isolated bug fix or improvement. PRs that do not involve any function-level edits are also discarded to ensure meaningful localization targets.

Category Labeling. we use `GPT-4o-2024-0513` to classify each issue based on its description. To ensure reliability, we sample the result three times for each issue and apply manual review in cases where the outputs are inconsistent, reducing potential noise and labeling errors.

Ground Truth Locations. The affected files or functions in the original codebase, as identified in the patches, are considered the target locations for the given issue. While it is possible to fix a bug in a location different from the ground truth, the extracted ground-truth locations still serve as approximate targets for localization. Additionally, edited code such as documents, import statements, and comments are excluded from the localization target. These elements are not considered relevant for code localization, as they do not directly impact the functionality of the code or its execution. By filtering out these elements, the focus is maintained on the core code changes that are relevant for localization.

C Implementation Details

C.1 Baselines Implementation

Regarding the embedding-based methods in our evaluation, while these approaches could theoretically index nodes at different levels (file, module, or function) to compute corresponding metrics, the standard implementations we evaluated operate at the function level, embedding each function as a single unit, following (Suresh et al., 2024).

We use OpenHands with its default generalist agent CodeActAgent (Wang et al., 2024a) from its agenthub. We use Openhands version 0.12.0 released on October 31, 2024. We employ OpenHands's remote runtime feature to parallelize evaluation on OpenHands (with CodeActAgent) and SWE-agent.

C.2 Quantifying Task Difficulty Based on Code Graph Distance

We measure task difficulty by computing the average shortest hop distance between the functions mentioned in the issue descriptions and the patched functions within our code graph. Specifically, we first extract potential function names from each issue description using `GPT-4o-2024-0513`, and identify their corresponding nodes in the code graph using the global dictionary. These identified nodes form the set of predicted nodes, denoted as $\mathcal{C}$.

Category	Keywords
Performance	bottleneck, performance improvement, memory usage optimization, time complexity reduction, latency improvement, scalability improvement, CPU usage reduction, caching improvement, concurrency optimization
Security	Out-of-bounds Write, Out-of-bounds Read, NULL Pointer Dereference, Missing Authorization, memory leak fix, security vulnerability, security issue, authentication bypass, authentication issue, better maintained, buffer overflow, denial of service, security hardening, security patch, unsafe deserialization, Use After Free, Integer Overflow or Wraparound, Uncontrolled Resource Consumption, Missing Authentication for Critical Function

Table 10: We use these Keywords to search for Performance and Security related issues with Github Search APIs.

Type	Method	Loc-Model	File (%)			Module (%)		Function (%)	
			NDCG@1	NDCG@3	NDCG@5	NDCG@5	NDCG@10	NDCG@5	NDCG@10
Embedding-Based	BM25 (Robertson et al., 2009)		38.69	46.5	50.61	37.31	39.86	26.15	27.92
	E5-base-v2 (Wang et al., 2022)		49.64	64.19	66.6	53.15	54.45	31.39	35.3
	Jina-Code-v2 (Günther et al., 2023)		43.43	59.93	63.7	51.02	54.13	33.28	36.44
	Codesage-large-v2 (Zhang et al., 2024a)		47.81	60.82	64.39	49.38	52.22	27.03	30.74
	CodeRankEmbed (Suresh et al., 2024)		52.55	67.54	70.39	57.51	59.76	40.28	42.55
Procedure-Based	Agentless (Xia et al., 2024)	`GPT-4o`	67.15	71.76	71.76	64.31	64.31	53.81	53.81
		`Claude-3.5`	72.63	76.72	76.87	67.36	67.36	57.55	57.55
Agent-Based	MoatlessTools (Örwall, 2024)	`GPT-4o`	73.36	80.03	80.33	68.57	69.09	49.77	50.62
		`Claude-3.5`	72.63	80.73	80.88	69.11	69.11	53.03	53.16
	SWE-agent (Yang et al., 2024)	`GPT-4o`	57.3	63.96	64.12	53.95	53.95	42.32	42.44
		`Claude-3.5`	77.37	84.32	84.93	72.77	72.9	59.67	59.79
	Openhands (Wang et al., 2025)	`GPT-4o`	60.95	67.62	68.39	58.18	58.6	44.34	44.66
		`Claude-3.5`	76.28	84.27	84.43	75.79	75.92	63.13	63.8
	LocAgent (Ours)	`Qwen2.5-7B(ft)`	70.80	79.36	80.9	70.99	71.68	55.62	58.09
		`Qwen2.5-32B(ft)`	75.91	84.74	85.64	76.28	76.77	64.27	65.93
		`Claude-3.5`	**77.74**	**86.19**	**87.14**	**77.73**	**78.1**	**64.34**	**65.57**

Table 11: NDCG scores comparison showing ranking quality of different methods.

Similarly, we link the ground truth functions from the patch to their corresponding nodes in the code graph, forming the set of target nodes, denoted as $\mathcal{T}$. To quantify the difficulty δ, we calculate the average shortest hop distance between the predicted nodes $\mathcal{C}$ and the target nodes $\mathcal{T}$, defined as:

$$\delta = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} min_{c \in \mathcal{C}} d(c, t)$$

where $d(c, t)$ represents the shortest hop distance between nodes c and t in the graph. For performance analysis stratified by difficulty, we round δ down to $\lfloor \delta \rfloor$ to group samples by difficulty levels, and we exclude samples where the LLM fails to extract any valid function names.

C.3 Fine-tuning details

We do not use explicit teacher-student distillation techniques (e.g., loss alignment or intermediate representation matching); instead, our data-centric distillation strategy effectively transfers task-specific reasoning abilities to the smaller model. We use `Qwen-2.5-Coder-Instruct` (Hui et al., 2024) 7B and 32B variants as our base models. We fine-tuned `Qwen-2.5-Coder-Instruct` 7B and 32B models on 768 training samples from the SWE-Bench training dataset, leveraging LoRA for efficient adaptation. The training set included 447 samples generated by `Claude-3.5`, while the remaining samples were iteratively generated using the fine-tuned `Qwen2.5-32B` model. The fine-tuning process was conducted over 5 epochs with *max_token* set to $128k$ and a learning rate of 2×10^{-4}.

Prompt

```
Given the following GitHub problem description, your objective is to localize the specific files, classes or functions, and lines
of code that need modification or contain key information to resolve the issue.

Follow these steps to localize the issue:
## Step 1: Categorize and Extract Key Problem Information
- Classify the problem statement into the following categories:
  Problem description, error trace, code to reproduce the bug, and additional context.
- Identify modules in the '{{package_name}}' package mentioned in each category.
- Use extracted keywords and line numbers to search for relevant code references for additional context.

## Step 2: Locate Referenced Modules
- Accurately determine specific modules
  - Explore the repo to familiarize yourself with its structure.
  - Analyze the described execution flow to identify specific modules or components being referenced.
- Pay special attention to distinguishing between modules with similar names using context and described execution flow.
- Output Format for collected relevant modules:
  - Use the format: 'file_path:QualifiedName'
  - E.g., for a function `calculate_sum` in the `MathUtils` class located in `src/helpers/math_helpers.py`, represent it as:
    'src/helpers/math_helpers.py:MathUtils.calculate_sum'.

## Step 3: Analyze and Reproducing the Problem
- Clarify the Purpose of the Issue
  - If expanding capabilities: Identify where and how to incorporate new behavior, fields, or modules.
  - If addressing unexpected behavior: Focus on localizing modules containing potential bugs.
- Reconstruct the execution flow
  - Identify main entry points triggering the issue.
  - Trace function calls, class interactions, and sequences of events.
  - Identify potential breakpoints causing the issue.
  Important: Keep the reconstructed flow focused on the problem, avoiding irrelevant details.

## Step 4: Locate Areas for Modification
- Locate specific files, functions, or lines of code requiring changes or containing critical information for resolving the issue.
- Consider upstream and downstream dependencies that may affect or be affected by the issue.
- If applicable, identify where to introduce new fields, functions, or variables.
- Think Thoroughly: List multiple potential solutions and consider edge cases that could impact the resolution.

## Output Format for Final Results:
Your final output should list the locations requiring modification, wrapped with triple backticks ```
Each location should include the file path, class name (if applicable), function name, or line numbers, ordered by importance.
Your answer would better include about 5 files.

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

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

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

Return just the location(s)
Note: Your thinking should be thorough and so it's fine if it's very long.
```

Figure 8: The task instruction prompt for LocAgent.