Ask KG Agent: A Multi-Agent Framework for Code Localization Using Code and Knowledge Graphs

Chanyoung Chung
KAIST
Daejeon, South Korea
chanyoung.chung@kaist.ac.kr

Jeemin Kim
KAIST
Daejeon, South Korea
kjm10122@kaist.ac.kr

Joyce Jiyoung Whang*
KAIST
Daejeon, South Korea
jjwhang@kaist.ac.kr

Abstract

Mapping software issue descriptions to relevant code segments, a task known as code localization, is a primary bottleneck in automated software engineering. While recent attempts have automated it using code graphs that capture syntax-level dependencies among code elements, including functions and modules, their code-centric approaches often overlook the design rationales and prior fix patterns present in a codebase's development history, such as pull requests and commits. We propose Co-Locator, a collaborative multi-agent code localization framework that integrates code graphs reflecting functional dependencies and codebase knowledge graphs constructed by extracting triplets that capture design rationales and past code modifications scattered across the development history. During localization, a Code Graph (CG) agent explores functional hierarchies and consults a Knowledge Graph (KG) agent to gather insights into developer intents and prior bug fixes. Combined with a Retriever agent that assists CG and KG agents, Co-Locator leverages a holistic view of the code's functional and historical context, considering what the code implements and how it was changed, to identify locations to fix. Experiments show that Co-Locator outperforms 12 state-of-the-art baseline methods.

1 Introduction

With the increasing complexity of modern software, automated software engineering has become essential for efficient system maintenance [11, 23]. A primary component of automated engineering is code localization [42], the task of identifying code segments relevant to software issues, facilitating subsequent tasks such as automated fault analysis and bug repair [5, 36]. Recently, the advent of Large Language Models (LLMs) [2, 37] has shifted the paradigm from simple keyword- or embedding-based retrieval toward reasoning-based localization [30, 41], leading to the emergence of LLM-based methods capable of autonomously navigating complex codebases to identify locations to fix.

Beyond analyzing the static snapshot of source code, there is a growing recognition that the development traces generated during software maintenance, such as issues, pull requests (PRs), and commits in GitHub, are essential for understanding the developer's intent and the rationale behind code changes [4, 26, 31, 35]. While the source code represents what the system currently does, the development history provides critical insights into why certain changes were made and how past problems were resolved. To leverage such knowledge, several recent studies have attempted to integrate the historical data to enhance the reasoning of LLM-based agents in a naive manner, such as augmenting prompts with relevant snippets [4] or allowing agents to search for raw commits [26, 31]. However, these approaches are fundamentally limited by treating development traces as isolated snippets for retrieval or simple inspection. Specifically, they ignore the dependencies between historical records [4, 26, 31], missing critical links such as which PRs were created to resolve a specific issue

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

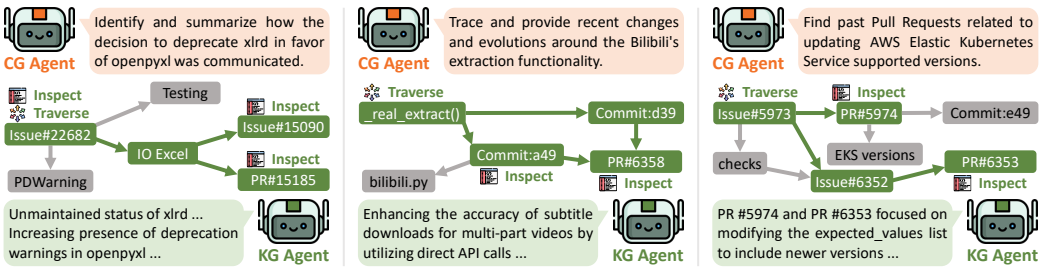


Figure 1: Examples of the CG agent querying the KG agent to retrieve contextual insights from maintenance logs or the developer's intent. Relation labels are omitted for brevity.

or how multiple issues reference each other to share technical context. Also, by presenting historical data in its raw, unstructured form, prior methods often obscure the broader design rationales across multiple maintenance logs [35], restricting agents to a shallow, linear inspection of individual items rather than allowing them to navigate through interrelated technical insights and design rationales. Therefore, existing methods remain limited to a superficial use of development history, making it challenging to capture the broader context spanning across fragmented records.

To address these limitations, we propose **Co-Locator**, a collaborative multi-agent framework that leverages development history represented as knowledge graphs [13, 14, 15] for code localization. Unlike prior approaches that treat each record independently, Co-Locator constructs a codebase knowledge graph[1] that captures interconnections between issues, PRs, and commits, augmented with triplets extracted by LLMs to capture the design rationales and developer intents underlying each record. Consequently, Co-Locator allows agents to navigate interrelated technical concepts across various records rather than inspecting each item individually, ensuring a comprehensive understanding of the historical context. To effectively utilize this structured historical and technical context, we introduce three agents operating on their designated graphs: a **Code Graph (CG) agent** for exploring code graphs [17, 21, 29] reflecting syntax-level code structures to identify locations to fix, a **Knowledge Graph (KG) agent** that explores knowledge graphs to provide historical context and design intents to the CG agent, and a **Retriever agent** assisting these agents by identifying entry points within their respective graphs.

The core of Co-Locator lies in a collaborative reasoning strategy in which the CG agent consults the KG agent to reconcile current code implementations with contextual insights from the knowledge graph. As illustrated in Figure 1, the CG agent initiates queries regarding technical decisions, code changes, or prior fix patterns, while the KG agent traverses the interconnected nodes of the knowledge graph to provide the underlying rationale. This interaction allows the CG agent to bridge the gap between "what" the code currently implements, "why" specific design decisions were made, and "how" related issues were resolved in the past. By integrating structural code logic with structured development history, Co-Locator enables a multi-faceted understanding of software issues that goes beyond static code analysis. Extensive experiments on real-world benchmark datasets show that Co-Locator significantly outperforms 12 code localization approaches, demonstrating the effectiveness of collaborative reasoning over both code logic and interconnected contextual knowledge.

2 Related work

Traditional retrieval-based approaches Early code localization methods primarily rely on lexical matching techniques such as BM25 [25], which rank relevant code snippets based on the textual overlap between the query and code elements. To enhance accuracy beyond keyword matching, dense retrieval approaches, such as E5-base-v2 [32], GTE [16], and SFR-Embedding-2 [18], project queries and candidates into a shared vector space to facilitate embedding-based matching. This research direction has been extended to include models specifically pre-trained on large-scale code corpora, such as Codesage-large-v2 [39], Jina-Code-v2 [8], and CodeRankEmbed [27]. While traditional retrieval-based approaches remain widely used, they treat code snippets as independent units and fail to account for functional dependencies between code elements, such as function calls and inheritance.

[1]To avoid confusion with the code graph, we refer to the codebase knowledge graph as the knowledge graph.

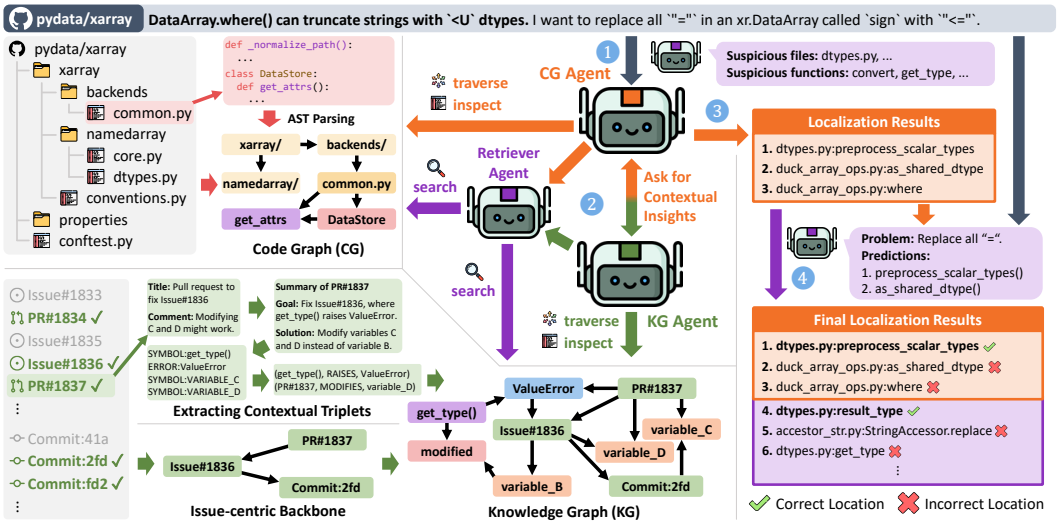


Figure 2: Given a software issue, Co-Locator constructs a code graph and a knowledge graph reflecting functional dependencies and development history, respectively. Then, three agents collaboratively predict locations to fix by exchanging the rationale behind source codes and prior fixes.

LLM-based code localization Recent advancements in LLMs have shifted the focus toward more sophisticated localization strategies that go beyond simple retrieval [33, 36, 43]. Agentless [34] utilizes a systematic procedure to identify relevant functions, while SWERank [24] employs the fine-tuned embedder and reranker to retrieve the locations to fix and rerank them based on the issue description. To account for the intricate connections between code elements, another line of research represents the codebase as a code graph [17, 21, 29], where code elements and their functional dependencies are defined as nodes and edges. Leveraging code graphs constructed through Abstract Syntax Tree (AST) parsing, LocAgent [5], OrcaLoca [38], and CoSIL [10] propose graph-guided agents that navigate on code graphs to identify the target locations for fixing. Despite their effectiveness, these methods largely overlook the evolutionary context of software, missing out on insights that can be derived from historical changes and previous bug-fix patterns.

Software maintenance with auxiliary resources Recent studies have explored integrating non-code resources to enrich the contextual understanding of a codebase. For instance, Prometheus [22] and LIDL [28] expand code graphs by adding non-code files as nodes, yet they merely link non-code files via a simple file-level hierarchy rather than capturing the logical dependencies between contents. Expanding the focus toward historical data, several works incorporate development traces to provide an evolutionary context of the codebase. CodeR [4] uses the most relevant past issue for a given bug report, and Code Researcher [26] and RepoMem [31] enable agents to search for commits. Despite such efforts, these methods remain limited to a simple, fragmented inspection of records, failing to capture the intricate dependencies, such as an issue being resolved by specific commits. While KGCompass [35] partially links issues and PRs for LLMs, it focuses solely on explicitly observed links, hindering a holistic understanding of the broader context scattered across multiple records.

3 Methodology

As illustrated in Figure 2, Co-Locator is designed as a collaborative framework comprising CG, KG, and Retriever agents. The localization process begins with the Retriever agent generating an initial prediction by retrieving relevant locations using the software issue description. This initial prediction, along with the issue description, serves as input for the CG agent to initiate subsequent exploration (❶). Building upon this input, the agents collaboratively narrow down the target locations by iteratively sharing structural and contextual insights within their respective domains (❷). Following this interaction, the CG agent integrates the gathered evidence to predict a refined set of locations to be modified (❸). Finally, the Retriever agent uses both the issue description and predicted locations to retrieve a supplemental set of locations to fix, ensuring a comprehensive localization result (❹).

3.1 Constructing code graphs via AST parsing

Understanding structural dependencies within a codebase, such as function calls and inheritance, is essential for understanding the program's functional logic and analyzing the impact of code changes. Following prior works [5, 10], we construct a code graph via AST parsing, where nodes and edges represent code elements and their syntactic relationships, respectively. Each node in the code graph is assigned to one of the following types: `DIRECTORY`, `FILE`, `CLASS`, `FUNCTION`, `VARIABLE`, `LIBRARY`, `LIBSYM` (library symbol), and `UNSYM` (undefined symbol). Also, edges are associated with six predefined relations: `CONTAINS`, `DEFINES`, `CALLS`, `IMPORTS`, `INHERITS`, and `REFERENCES`. While prior works focus on function-level dependencies, we extend code graphs to include variables and external libraries, enabling the localization model to trace data-level and external library-related issues. More details of code graph construction are provided in Appendix A.1.

3.2 Extracting knowledge graphs reflecting codebase history

Although code graphs represent the functional dependencies within source code, they do not explicitly capture the rationale behind past fixes or the developer's intent underlying specific code, which often provide essential clues for resolving a software issue. To leverage such context scattered across development history, we construct a knowledge graph by (1) building an issue-centric backbone from historical records relevant to the software issue and (2) augmenting triplets extracted from the textual contents of the records. When generating the knowledge graph, we use only information available at the time the target software issue was raised, thereby preventing potential information leakage.

Building an issue-centric backbone Since a codebase often contains numerous development traces unrelated to a software issue, using the entire history can introduce noise that distracts a localization model from the core issue. To construct an issue-centric knowledge graph, we collect top-D historical records whose titles exhibit the highest cosine similarity to the target issue description, calculated using embeddings obtained from Nomic-embed-text-v1.5 [19]. We also collect records mentioned within these items to uncover hidden contexts that might be overlooked by the title-based retrieval.

We establish the issue-centric backbone of the knowledge graph by formalizing the relationships within this collection. First, each record mention is converted as a triplet in the form of (`RECORD1`, `CROSS-REFERENCES`, `RECORD2`). Then, each entity is connected to its associated labels, milestones, and commits, forming triplets such as (`PR`, `MERGES`, `COMMIT`). Finally, for each integrated commit, its associated patches are parsed to identify the modified code elements, adding triplets of the form (`COMMIT`, `MODIFIES`, `SYMBOL`). This construction process allows the knowledge graph to reflect development activities, including which PRs resolved an issue or which commits modified a file.

Augmenting contextual knowledge to the backbone While the issue-centric backbone captures dependencies among records, it cannot represent the specific design rationale or domain knowledge embedded in their textual content. Accordingly, we employ an LLM to extract triplets reflecting contextual knowledge, such as the developer's intent and prior fix patterns, to enrich the backbone.

First, for each record, the LLM generates a structured summary to capture its core technical essence while filtering out irrelevant noise. Next, using these summaries, the LLM identifies key entities of the following types: `ISSUE`, `PR`, `COMMIT`, `FILE`, `SYMBOL`, `ENVIRONMENT`, `DEPENDENCY`, `VERSION`, `CONFIG`, `ERROR`, and `CONCEPT`. Finally, the LLM extracts triplets by referencing both the summaries and the identified entities, establishing relationships that clarify the developer's intent and design choices. These triplets are integrated into the knowledge graph, enriching the issue-centric backbone with the underlying rationale and domain-specific insights of the development history. By integrating these contextual triplets into the knowledge graph, we identify hidden links between records that are not explicitly mentioned, while enabling scattered knowledge to be viewed collectively at a glance. Additional details and prompts for extracting knowledge graphs are provided in Appendix A.2 and D.

3.3 Co-Locator: a collaborative multi-agent code localization framework

To effectively reason over code graphs and knowledge graphs, we define three agents: the CG agent, the KG agent, and the Retriever agent. Assisted by the Retriever agent, the CG and KG agents operate on their respective graphs and collaborate to bridge the code logic with contextual knowledge across the development history, thereby enabling a multi-faceted understanding of software issues.

CG agent The CG agent identifies potential fault locations by tracing logical flows within the code graph. Rather than simply retrieving code snippets, the agent iteratively refines its search space by integrating functional dependencies among code elements and their implementation details, and by asking the KG agent for contextual insights. The CG agent leverages these diverse information sources via a set of tools: (1) `search`: Search entities in the code graph, (2) `traverse`: Provide 2-hop neighbors of an entity in the code graph, (3) `inspect`: Inspect the implementation of a file, module, or function, and (4) `ask_kg_agent`: Query the KG agent to obtain contextual insights.

KG agent The KG agent provides historical and intent-based context to the CG agent that cannot be addressed solely through code analysis. By traversing the knowledge graph, the KG agent consolidates related concepts across records, providing the background needed to understand the codebase's underlying intents and design rationales. The KG agent utilizes the following tools: (1) `search`: Search entities in the knowledge graph, (2) `traverse`: Provide 2-hop neighbors of an entity in the knowledge graph, (3) `inspect`: Inspect the textual contents of an issue, PR, or commit.

Upon receiving a query from the CG agent, the KG agent leverages the tools to traverse the knowledge graph and provide contextual insights. For example, when the CG agent requests the rationale behind a specific implementation, such as the transition between libraries or changes in service specifications, the KG agent traverses the knowledge graph to retrieve the original developer discussions and PRs that explain the rationale behind such a transition. By consolidating these scattered technical insights, the KG agent provides the necessary context to clarify the intent behind complex code segments, thereby guiding localization with a deeper understanding of the codebase's history.

Retriever agent The Retriever agent assists both the CG and KG agents by searching the most relevant entities for their respective queries within the code or knowledge graphs. To handle both semantic and keyword-based queries, the Retriever agent performs a hybrid search by merging results from embedding- and keyword-based retrieval. Also, the Retriever agent enhances the reliability of tool-calling by correcting invalid arguments to their most appropriate counterparts within the graphs.

Collaborative reasoning In summary, the localization process is driven by the collaboration of these agents. The CG agent navigates the code graph to trace logical paths, frequently consulting the KG agent to clarify developer intents or historical contexts that are not apparent in the source code alone. The Retriever agent serves as a supporting component, streamlining the interaction by providing relevant candidates and ensuring the accuracy of tool calling. This collaborative reasoning strategy enables Co-Locator to bridge the gap between static code logic and development history.

4 Experiments

The performance of Co-Locator is evaluated against 12 code localization methods. Implementations of Co-Locator are provided in `https://github.com/bdi-lab/Co-Locator`.

4.1 Experimental settings

Datasets and evaluation metrics Experiments are conducted on two datasets, LocBench [5] and SWE-Bench-Live [40]. We use the lite subset of SWE-Bench-Live and further select 244 instances out of 300 in which the ground-truth modifications cover up to 5 files and 10 functions, following the data construction strategy of LocBench [5]. The dataset licenses are detailed in B.1. Consistent with prior works [5, 24], code localization performance is measured at three granularities: file-, module-, and function-level. Accuracy at K (Acc@K) is used as the evaluation metric, where the localization is considered successful only if all ground-truth locations are contained within the top K predictions.

Baseline methods Co-Locator is compared with 12 baselines from the following categories: (1) embedding-based approaches: BM25 [25], E5-base-v2 [32], CodeRankEmbed [27], Jina-Code-v2 [8], GTE [16], Codesage-large-v2 [39], SFR-Embedding-2 [18], and SWERank [24] (2) procedural approaches: Agentless [34] (3) agent-based approaches: OrcaLoca [38], LocAgent [5], and CoSIL [10].

As both datasets focus on GitHub issues created after October 2024, we use GPT-4o-2024-08-06 [20] (GPT-4o for short) and Qwen2.5-Coder-32B-Instruct [9] (Qwen2.5 for short) for LLM-based methods to avoid data contamination; the cutoff for GPT-4o is October 2023, while Qwen2.5 is pretrained on codebases available before February 2024. For LocAgent, we use the vanilla Qwen2.5 instead

Table 1: Code localization results. Numbers under each granularity denote the K in Acc@K. Best, second-best, and third-best results are marked **bold**, underline, and *italic*, respectively. Notably, Co-Locator$^+$, using both GPT-4o and Qwen2.5 as backbone LLMs, outperforms all baseline methods.

		SWE-Bench-Live						LocBench					
		File		Module		Function		File		Module		Function	
		5	10	10	15	10	15	5	10	10	15	10	15
BM25		0.553	0.656	0.467	0.508	0.295	0.332	0.538	0.645	0.452	0.516	0.302	0.336
E5-base-v2		0.680	0.754	0.603	0.652	0.406	0.443	0.716	0.796	0.607	0.652	0.432	0.468
CodeRankEmbed		0.586	0.684	0.533	0.594	0.357	0.393	0.616	0.698	0.529	0.568	0.320	0.370
Jina-Code-v2		0.664	0.758	0.631	0.684	0.443	0.480	0.705	0.775	0.609	0.655	0.430	0.489
GTE		0.557	0.697	0.484	0.541	0.271	0.307	0.596	0.696	0.500	0.545	0.305	0.348
Codesage-large-v2		0.656	0.771	0.607	0.660	0.398	0.447	0.671	0.734	0.579	0.623	0.402	0.430
SFR-Embedding-2		0.742	0.812	0.697	0.750	0.541	0.574	0.771	0.855	0.696	0.743	0.509	0.564
SWERank$^-$		0.754	0.824	0.709	0.746	0.541	0.574	0.752	0.823	0.675	0.711	0.507	0.557
SWERank$^+$		*0.787*	0.840	*0.754*	*0.791*	*0.586*	*0.623*	*0.811*	0.855	*0.729*	0.759	*0.616*	*0.646*
GPT-4o	Agentless	0.611	0.705	0.496	0.496	0.324	0.328	0.780	0.838	0.632	0.636	0.434	0.439
	OrcaLoca	0.332	0.332	0.254	0.254	0.201	0.201	0.521	0.521	0.364	0.364	0.280	0.280
	LocAgent	0.557	0.557	0.381	0.381	0.258	0.258	0.702	0.707	0.518	0.518	0.404	0.405
	CoSIL	0.635	0.697	0.582	0.590	0.422	0.434	0.779	0.811	0.641	0.666	0.523	0.550
	Co-Locator$^-$	0.775	0.832	0.746	0.787	0.553	0.607	0.802	0.848	0.714	0.746	0.589	0.632
	Co-Locator$^+$	**0.811**	0.857	0.770	**0.811**	**0.623**	0.635	0.830	0.871	0.752	0.786	**0.657**	0.686
Qwen2.5	Agentless	0.607	0.680	0.422	0.422	0.254	0.254	0.748	0.816	0.521	0.530	0.357	0.363
	OrcaLoca	0.303	0.303	0.234	0.234	0.172	0.172	0.429	0.429	0.309	0.309	0.229	0.229
	LocAgent	0.480	0.480	0.299	0.299	0.217	0.217	0.613	0.613	0.445	0.452	0.346	0.350
	CoSIL	0.611	0.689	0.504	0.512	0.377	0.393	0.741	0.775	0.618	0.632	0.484	0.502
	Co-Locator$^-$	0.770	*0.844*	0.750	0.799	0.561	0.607	0.808	*0.866*	0.728	*0.760*	0.591	0.629
	Co-Locator$^+$	0.807	**0.865**	**0.783**	**0.811**	0.611	**0.643**	**0.848**	**0.880**	**0.757**	**0.787**	0.654	**0.689**

of the authors' checkpoint, as the provided version failed to produce valid outputs for all instances. The original checkpoints of SWERank are trained on an extensive dataset of 3,387 repositories that overlap with SWE-Bench-Live. To ensure a fair comparison and prevent data contamination, the vanilla Qwen2.5 is also used for running SWERank. Additional details are provided in Appendix B.2.

Experimental settings of Co-Locator While constructing knowledge graphs, we initially retrieve ten records and employ GPT-4o-mini to extract triplets. Regarding agent configurations, the CG agent is permitted a maximum of 30 tool calls, while the KG agent is allowed up to five tool calls per query. In the initial stage of the localization process, the Retriever agent provides the top 10 files and functions to the CG agent. Since Qwen2.5 does not natively support tool calling, we implemented a custom tool calling parser and applied a repetition penalty of 1.2 to enhance generation quality. Environments and prompts used in Co-Locator are provided in Appendices B.3 and D, respectively.

4.2 Code localization results

Given the effectiveness of reranking [24, 27], we evaluate two variants of Co-Locator: Co-Locator$^-$, which excludes reranking, and Co-Locator$^+$, which adopts the reranking strategy from SWERank [24] that reorders the top 100 candidates in a bottom-up manner by processing from the lowest-ranked toward the highest-ranked predictions. For a consistent comparison, SWERank is also evaluated in two forms: a modified version without reranking (SWERank$^-$) and its original version (SWERank$^+$). To promote the reproducibility of the results, all LLM-based approaches, including Co-Locator, are evaluated using greedy decoding with a temperature of 0. Considering that the ground-truth file counts range from 1 to 5 and function counts from 1 to 10 in both datasets, we report Acc@5 and Acc@10 for file-level localization, and Acc@10 and Acc@15 for module and function levels.

Table 1 demonstrates that Co-Locator$^+$ significantly outperforms all baseline methods regardless of the backbone LLM. Specifically, a statistical t-test over three independent runs against SWERank, the best-performing baseline, shows that the performance gains of Co-Locator are statistically significant across all datasets and metrics ($p < 0.05$), with detailed p-values provided in Appendix C.1. When

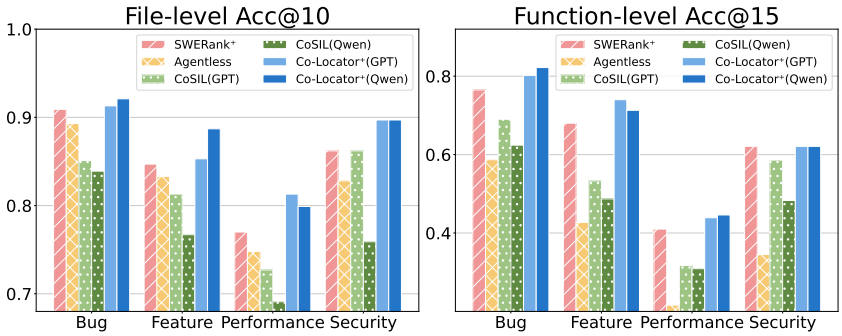


Figure 3: Per-category localization results on LocBench. Co-Locator (represented in blue) outperforms the baseline methods in all categories.

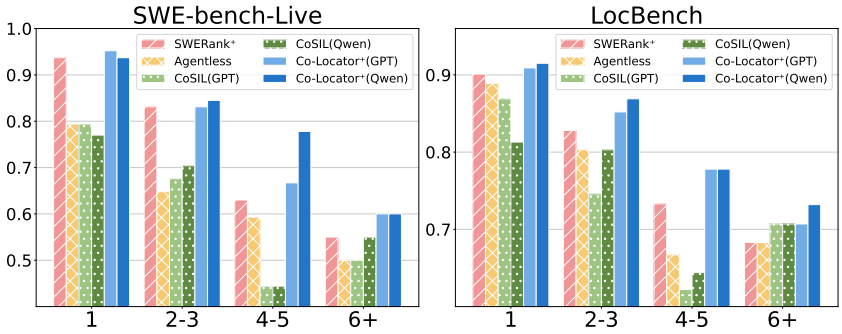


Figure 4: File-level Acc@10 by issue complexity. Co-Locator (represented in blue) consistently outperforms the baseline methods.

using GPT-4o as the backbone LLM, Co-Locator$^+$ achieves an average relative gain of 3.0% on SWE-Bench-Live and 4.0% on LocBench compared to the best-performing baselines. Similarly, with the Qwen2.5 backbone, Co-Locator$^+$ yields average relative gains of 3.2% and 4.6% on SWE-Bench-Live and LocBench, respectively. Notably, even Co-Locator$^-$ exhibits strong localization capabilities, surpassing all other baselines except SWERank$^+$, demonstrating the effectiveness of our collaborative multi-agent-based retrieval approach. The results also show that incorporating a reranking stage allows for further refinement, with SWERank$^+$ and Co-Locator$^+$ consistently yielding higher accuracy than SWERank$^-$ and Co-Locator$^-$, respectively.

To evaluate whether Co-Locator maintains consistent performance across various scenarios, we analyze its effectiveness across four categories in LocBench: Bug Report (242 instances), Feature Request (150 instances), Performance Issue (139 instances), and Security Vulnerability (29 instances). Figure 3 visualizes the file-level and function-level localization results for each category, comparing Co-Locator$^+$ (■ for GPT-4o-based, ■ for Qwen2.5-based) against the strongest baselines from each baseline group: SWERank$^+$ (▨), Agentless (▨), and CoSIL (■ for GPT-4o-based, ■ for Qwen2.5-based). The results demonstrate that Co-Locator$^+$ consistently outperforms all representative baselines across all problem categories.

To investigate the performance trends of code localization models relative to issue complexity, we split the instances into four complexity levels based on the number of ground-truth functions: 1, 2–3, 4–5, and 6 or more functions. This categorization can serve as a direct indicator of localization difficulty, as issues involving more functions generally require capturing more intricate dependencies across the codebase. Figure 4 presents the file-level localization results across difficulty levels. Across all models, we observe a common trend where accuracy decreases as difficulty increases, reflecting the inherent challenge of localizing multiple locations. Nevertheless, Co-Locator$^+$ consistently outperforms all baselines at every complexity level across both datasets. Notably, even in the most challenging scenarios, our framework maintains a substantial performance margin over the baselines.

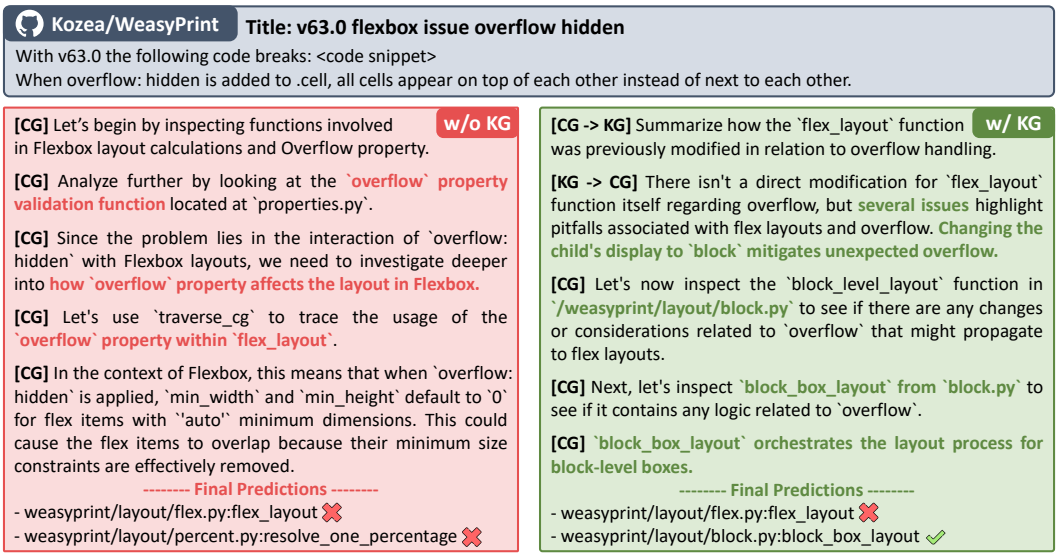


Figure 5: Case study of Co-Locator$^+$ with and without using knowledge graphs. While the variant without the knowledge graph fails to localize the correct locations, Co-Locator$^+$ successfully identifies the ground-truth files and functions by leveraging historical context relevant to the software issue.

4.3 Ablation studies and case studies of Co-Locator

To evaluate the effectiveness of each model component in Co-Locator, we conduct an ablation study by comparing Co-Locator$^+$ with five variants, all implemented using Qwen2.5 as the backbone LLM: (1) **w/o KG** removes the knowledge graph and disables the KG agent, (2) **w/o role separation** merges CG, KG, and Retriever agents into a single agent, (3) **w/o traverse** removes the graph traversal tools from both the CG and KG agents, (4) **w/o initial retrieval** does not provide the Retriever agent's initial prediction to the CG agent, and (5) **w/o post retrieval** skips the prediction-based retrieval after the multi-agent localization stage. Table 2 reports the localization accuracy of Co-Locator$^+$ alongside these five variants on SWE-Bench-Live, where Co-Locator$^+$ outperforms all variants. The performance degradation observed in these variants highlights the importance of each model component, such as knowledge graphs (w/o KG variant), multi-agent collaboration (w/o role separation variant), and graph-guided traversal (w/o traverse variant), for identifying locations relevant to software issues.

Table 2: Ablation study on SWE-Bench-Live.

	File		Module		Function	
	5	10	10	15	10	15
w/o KG	0.791	0.840	0.762	0.803	0.590	0.611
w/o role separation	0.799	0.848	0.758	0.803	0.607	0.631
w/o traverse	0.787	0.832	0.762	0.795	0.594	0.623
w/o initial retrieval	0.795	0.857	0.754	0.807	0.586	0.607
w/o post retrieval	0.799	**0.865**	0.770	0.795	0.590	0.615
Co-Locator$^+$	**0.807**	**0.865**	**0.783**	**0.811**	**0.611**	**0.643**

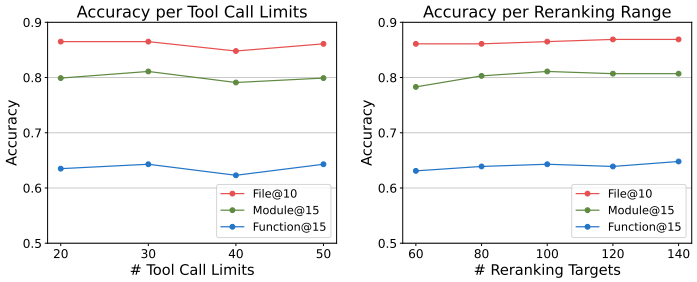
Statistical t-tests across multiple independent runs, with detailed p-values reported in Appendix C.1, confirm that Co-Locator$^+$ achieves statistically significant improvements over its w/o KG variant across all datasets and metrics ($p < 0.05$), thereby highlighting the essential role of the knowledge graph. To gain deeper insights into these improvements, we conduct a case study comparing Co-Locator$^+$ with the w/o KG variant on an instance (`kozea__weasyprint-2300`) in SWE-Bench-Live. The issue involves a `flexbox` layout bug where applying `overflow: hidden` causes flex items to overlap incorrectly. As shown in Figure 5, the w/o KG variant exhibits a limited search pattern by overly relying on the literal term "overflow", which leads to incorrect predictions at both levels. In contrast, by utilizing historical content discovered through knowledge graph traversal, Co-Locator$^+$ identifies that previous flex-related overflow pitfalls were resolved by adjusting how child elements are displayed as `block`, and identifies the correct files and functions to modify.

Table 3: Cost analysis of agent-based methods. For each level, efficiency is measured as the accuracy (Acc@10 for file, Acc@15 for module and function) divided by the average cost per instance.

	SWE-Bench-Live				LocBench				Runtime per instance (min.)
	Cost ($)	File	Module	Function	Cost ($)	File	Module	Function	
OrcaLoca	0.216	1.535	1.175	0.928	0.246	2.119	1.480	1.139	2.665
LocAgent	0.224	2.490	1.703	1.153	0.263	2.683	1.966	1.537	0.172
CoSIL	0.217	3.219	2.725	2.005	0.264	3.078	2.528	2.087	0.559
Co-Locator$^-$	**0.213**	**3.910**	**3.699**	**2.853**	**0.170**	**5.002**	**4.401**	**3.716**	**0.055**
Co-Locator$^+$	0.374	2.294	2.171	1.700	0.402	2.165	1.953	1.705	0.106

4.4 Hyperparameter sensitivity analysis

To investigate the relationship between exploration depth and localization accuracy, we analyze the performance of Co-Locator$^+$ with Qwen2.5 under varying tool call limits for the CG agent. As illustrated in Figure 6a, increasing the number of allowed tool calls does not necessarily lead to higher accuracy. While a moderate limit provides sufficient budget for agents to navigate the code graph, excessive tool calls can introduce redundant information or lead the agent into irrelevant paths, causing slight fluctuations in performance. Experimental results suggest that the strategic selection of tools is more critical for effective localization than the volume of exploration steps. An example is `python-control-1064`. Under a tool budget of 50, the CG Agent reached to the ground-truth files (`frdata.py` and `xferfcn.py`) and analyzed the target functions (`FrequencyResponseData.__init__()` and `zpk()`) within 23 tool calls. However, instead of stopping, the agent spent its remaining budget continuously exploring unrelated modules, ultimately omitting the correct locations from its final prediction.



(a) Tool call limits (b) Reranking range

Figure 6: Hyperparameter sensitivity analysis on SWE-Bench-Live.

Also, we examine the sensitivity of Co-Locator$^+$ to the reranking range by varying the number of candidate locations used in reranking, using Qwen2.5 as the backbone LLM. As shown in Figure 6b, the performance remains relatively stable across different numbers of reranking candidates, showing only marginal improvements as the number of targets increases. While increasing the reranking range improves the likelihood of including ground-truth locations, it also increases the risk of prioritizing incorrect locations, meaning that a larger range does not always lead to higher performance.

4.5 Cost analysis of agent-based approaches

We conduct a cost analysis comparing Co-Locator with agent-based approaches, using GPT-4o as the backbone LLM. Following [31], we report the average cost per instance incurred during the agentic reasoning process in United States dollar (Cost ($)), the average runtime per instance, and present the efficiency metrics utilized in LocAgent [5]. Specifically, for file-level efficiency, we divide the Acc@10 results by the average cost, and for module- and function-level efficiency, we divide the Acc@15 results by the average cost. Table 3 shows that Co-Locator$^-$ achieves the highest cost-effectiveness across all levels and datasets while also recording the shortest average runtime. Even without reranking, Co-Locator$^-$ outperforms all agent-based methods while maintaining a lower average cost. Although Co-Locator$^+$ incurs a higher average cost due to additional LLM calls for reranking, it yields the highest accuracy across all datasets and metrics. Overall, Co-Locator provides a flexible and efficient solution, with the core multi-agent collaboration offering superior cost-efficiency, and the reranking module providing further performance enhancement.

4.6 Impacts on code generation

To assess the impact of code localization performance on downstream software repair tasks, we evaluate the code generation performance using SWE-Bench-Live. We utilize Agentless [34] with GPT-4o to generate patches by providing the localization outputs of each method as the target locations. Following the standard Agentless configuration, 40 patches are generated per instance, using greedy decoding for the first patch and a temperature of 0.8 for the remaining patches. The resolve rate is used as the evaluation metric, which measures the percentage of instances successfully fixed by the final patch selected from 40 candidates. To establish an upper bound on performance, we also included an Oracle setting in which the ground-truth locations were directly provided. In Table 4, Co-Locator$^+$ achieves the highest resolve rate on SWE-Bench-Live, showing a relative improvement of 18% over SWERank$^+$. Notably, the small performance gap between Co-Locator$^+$ and Oracle indicates that locations identified by Co-Locator$^+$ are nearly as effective for patch generation as using ground-truth locations.

Table 4: Code generation results.

	Resolve rate (%)
SWERank$^+$	11.1
Agentless	10.6
CoSIL	9.4
Co-Locator$^+$	**13.1**
Oracle	14.8

5 Conclusion and future work

We introduce Co-Locator, a novel multi-agent framework for code localization that integrates syntax-level relationships within source code with development history. Unlike prior approaches that rely primarily on static code analysis, our framework leverages development traces, such as issues, PRs, and commits, consolidated into a comprehensive knowledge graph. By facilitating collaborative reasoning among specialized agents, i.e., the CG agent, the KG agent, and the Retriever agent, Co-Locator effectively matches the software issues to their relevant code locations.

Limitations and future work Currently, Co-Locator constructs code graphs using Python-specific AST parsing, which limits its applicability to Python-based repositories. To support localization on a broader range of codebases, we plan to extend our code graph generation capabilities to support multiple programming languages, such as Rust and Java. Also, we will extend Co-Locator by leveraging the contextual insights obtained from knowledge graphs to support automated code generation, ultimately developing a unified system for both code localization and program repair. Furthermore, we plan to expand our knowledge graphs beyond development history to incorporate documentation and multimodal resources within codebases by representing diverse forms of information, including numerical attributes [6], auxiliary information [12], and knowledge at multiple levels of granularity [7].

Acknowledgments and Disclosure of Funding

This work was supported by Samsung Electronics Co., Ltd(IO251201-14121-01). From the governmental funding agency, this work was also supported by the Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT): 50% from RS-2022-II220369, (Part 4) Development of AI Technology to support Expert Decision-making that can Explain the Reasons/Grounds for Judgment Results based on Expert Knowledge and 50% from RS-2026-25617720, AI Star Fellowship Support Program(KAIST, Regional-led).

We declare that we have no competing interests that could be perceived as influencing the work presented in this paper.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel

Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems*, pages 1877–1901, 2020.

[3] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Will Lillis, ObserverOfTime, Christian Clason, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Riley Bruins, Antonin Delpeuch, Daumantas Kavolis, Michael Davis, Ika, bfredl, Tuan-Anh Nguyen, Amin Ya, Stafford Brunk, skewb1k, Matt Massicotte, Niranjan Hasabnis, Rob Rix, James McCoy, Mingkai Dong, Samuel Moelius, Steven Kalt, and Josh Vera. tree-sitter/tree-sitter. `https://doi.org/10.5281/zenodo.19357078`, 2026.

[4] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. CodeR: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

[5] 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 *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, 2025.

[6] Chanyoung Chung, Jaejun Lee, and Joyce Jiyoung Whang. Representation learning on hyper-relational and numeric knowledge graphs with transformers. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 310–322, 2023.

[7] Chanyoung Chung and Joyce Jiyoung Whang. Learning representations of bi-level knowledge graphs for reasoning beyond link prediction. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, pages 4208–4216, 2023.

[8] Michael Günther, Louis Milliken, Jonathan Geuter, Georgios Mastrapas, Bo Wang, and Han Xiao. Jina Embeddings: A novel set of high-performance sentence embedding models. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software*, pages 8–18, 2023.

[9] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[10] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via LLM-driven iterative code graph searching. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, pages 3034–3045, 2025.

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *Proceedings of the 12th International Conference on Learning Representations*, pages 54107–54157, 2024.

[12] Jaejun Lee, Chanyoung Chung, Hochang Lee, Sungho Jo, and Joyce Jiyoung Whang. VISTA: Visual-textual knowledge graph representation learning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 7314–7328, 2023.

[13] Jaejun Lee, Chanyoung Chung, and Joyce Jiyoung Whang. InGram: Inductive knowledge graph embedding via relation graphs. In *Proceedings of the 40th International Conference on Machine Learning*, pages 18796–18809, 2023.

[14] Jaejun Lee, Seheon Kim, and Joyce Jiyoung Whang. Generative representation learning on hyper-relational knowledge graphs via masked discrete diffusion. In *Proceedings of the 43rd International Conference on Machine Learning*, pages 64205–64232, 2026.

[15] Jaejun Lee and Joyce Jiyoung Whang. Structure is all you need: Structural representation learning on hyper-relational knowledge graphs. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 33765–33783, 2025.

[16] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*, 2023.

[17] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 142–160, 2025.

[18] Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. SFR-Embedding-2: Advanced text embedding with multi-stage training. `https://huggingface.co/Salesforce/SFR-Embedding-2_R`, 2025.

[19] Zach Nussbaum, John Xavier Morris, Andriy Mulyar, and Brandon Duderstadt. Nomic Embed: Training a reproducible long context text embedder. *Transactions on Machine Learning Research*, 2025.

[20] OpenAI. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[21] 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. In *Proceedings of the 13th International Conference on Learning Representations*, pages 30098–30121, 2025.

[22] Yue Pan, Zimin Chen, Siyu Lu, Zhaoyang Chu, Xiang Li, Han Li, Yang Feng, Claire Le Goues, Federica Sarro, Martin Monperrus, and He Ye. Prometheus: Towards long-horizon codebase navigation for repository-level problem solving. *arXiv preprint arXiv:2507.19942*, 2026.

[23] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. SoapFL: A standard operating procedure for LLM-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187, 2025.

[24] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan-Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. SWERank: Software issue localization with code ranking. In *Proceedings of the 14th International Conference on Learning Representations*, 2026.

[25] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Proceedings of the 3rd Text Retrieval Conference*, pages 109–126, 1994.

[26] Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. Code Researcher: Deep research agent for large systems code and commit history. *arXiv preprint arXiv:2506.11060*, 2025.

[27] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *Proceedings of the 13th International Conference on Learning Representations*, pages 15800–15817, 2025.

[28] Gou Tan, Zilong He, Min Li, Pengfei Chen, Jieke Shi, Zhensu Sun, Ting Zhang, Danwen Chen, Lwin Khin Shar, Chuanfu Zhang, and David Lo. LIDL: LLM integration defect localization via knowledge graph-enhanced multi-agent analysis. *arXiv preprint arXiv:2601.05539*, 2026.

[29] Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. Code Graph Model (CGM): A graph-integrated large language model for repository-level software engineering tasks. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems*, pages 15869–15909, 2025.

[30] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for github issue resolution. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*, pages 51963–51993, 2024.

[31] Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *Proceedings of the 14th International Conference on Learning Representations*, 2026.

[32] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*, 2024.

[33] 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. OpenHands: An open platform for AI software developers as generalist agents. In *Proceedings of the 13th International Conference on Learning Representations*, pages 65882–65919, 2025.

[34] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 801–824, 2025.

[35] Boyang Yang, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, Bach Le, and Haoye Tian. Enhancing repository-level software repair via repository-aware knowledge graphs. *arXiv preprint arXiv:2503.21710*, 2025.

[36] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*, pages 50528–50652, 2024.

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations*, 2023.

[38] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. OrcaLoca: An LLM agent framework for software issue localization. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 73416–73436, 2025.

[39] Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale. In *Proceedings of the 12th International Conference on Learning Representations*, pages 47427–47446, 2024.

[40] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems*, 2025.

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

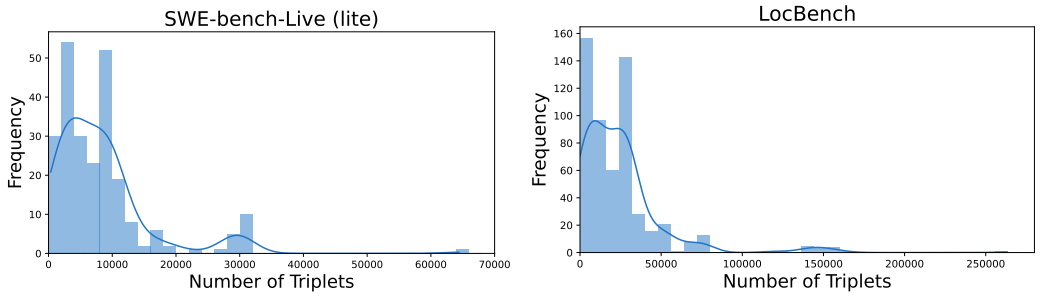
[42] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24, 2012.

[43] Albert Örwall. Moatless Tools. `https://github.com/aorwall/moatless-tools`, 2024.

A Detailed specifications of code graphs and knowledge graphs

This section provides additional details for constructing code graphs and knowledge graphs, along with their size distributions in the benchmark datasets.

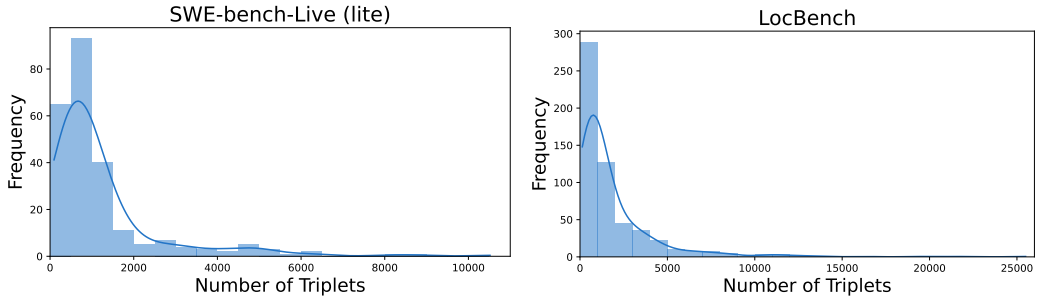
A.1 Details and statistics of code graphs



(a) SWE-Bench-Live (b) LocBench

Figure 7: Sizes of the constructed code graphs.

To construct code graphs, we used Tree-Sitter [3], which supports AST parsing for Python. Following prior works utilizing code graphs [1, 5, 21, 29], non-Python files and all test files were excluded from the graph construction process. Table 5 provides the relation types defined within the code graph, along with their respective descriptions and examples. Figure 7 visualizes the sizes of the constructed code graphs for both SWE-Bench-Live and LocBench. On average, code graphs for SWE-Bench-Live have approximately 8,300 triplets, whereas LocBench exhibits a slightly larger scale, with an average of 24,500 triplets. This difference in scale reflects the larger size and complexity of the repositories included in LocBench compared to those in SWE-Bench-Live.



(a) SWE-Bench-Live (b) LocBench

Figure 8: Sizes of the constructed knowledge graphs.

A.2 Details and statistics of knowledge graphs

As mentioned in Section 3.2, to prevent information leakage, the knowledge graph is constructed using only the development traces available at the time the target software issue was reported. For each instance, we identify the corresponding GitHub issues associated with the problem, based on text matching. The earliest creation time among the collected GitHub issues is used as a cutoff, and development traces generated after this point are excluded from the knowledge graph construction.

Table 5: Relation types in code graphs. We omit entity types for brevity.

Relation	Description	Example
`CONTAINS`	An element contains another element.	(`/src/cfnlint`, `CONTAINS`, `core.py`)
`INHERITS`	A class inherits another class.	(`DuplicateRuleError`, `INHERITS`, `CfnLintError`)
`CALLS`	A function calls another function.	(`_build_message()`, `CALLS`, `_get_state()`)
`IMPORTS`	A file imports a library.	(`create_cfn_rule.py`, `IMPORTS`, `jinja2`)
`DEFINES`	A library defines a symbol.	(`jinja2`, `DEFINES`, `FileSystemLoader`)
`REFERENCES`	An element references a variable.	(`_init_condition()`, `REFERENCES`, `_CONDITION`)

Table 6: Relation types in issue-centric backbones of knowledge graphs.

Relation	Description	Example
CROSS-REFERENCED	A record mentions another record.	(ISSUE#2004, CROSS-REFERENCED, PR#2006)
LABELED	A record is labeled with a label.	(ISSUE#2077, LABELED, LABEL:spec bug)
MILESTONED	A milestone contains a record.	(ISSUE#27515, MILESTONED, v3.8.3)
COMMITTED	A commit is involved in a PR.	(PR#2006, COMMITTED, COMMIT:7ff579)

GitHub API is utilized to collect issues, PRs, and commits from GitHub repositories. To construct knowledge graphs, we spent about $0.15 per instance in both SWE-Bench-Live and LocBench. Table 6 shows the relation types defined in the issue-centric backbones, along with their descriptions and examples. Also, Figure 8 visualizes the sizes of the extracted knowledge graphs in both datasets, with average triplets of about 1,250 and 1,850 in SWE-Bench-Live and LocBench, respectively.

B Additional details of experiments

In this section, we provide additional experimental details, such as licenses, additional specifications of baseline methods, and environments for running the experiments.

B.1 Licenses of used assets

Two datasets are used in our experiments: LocBench [5] and SWE-Bench-Live [40]. LocBench is distributed under the Apache-2.0 license, and SWE-Bench-Live is provided under the MIT license. We utilize Tree-Sitter [3] under the MIT license for constructing code graphs and employ Nomic-embed-text-v1.5 [19] under the Apache-2.0 license for knowledge graph construction. During localization, we employ Qwen2.5-Coder-32B-Instruct [9] as a backbone LLM, which is licensed under Apache-2.0. In addition, the Retriever agent uses gte-Qwen2-7B-instruct [16] for embedding-based retrieval, which is also distributed under the Apache-2.0 license. All assets used in our paper are under open-source licenses, ensuring there are no restrictions on their use for our experiments.

B.2 Additional details of baseline methods

While we consider various baselines for our evaluation, certain models were excluded from the final comparison. CodeR [4], Code Researcher [26], and KGCompass [35] are excluded as they are primarily designed for code generation rather than localization, which falls outside our current experimental scope. Also, RepoMem [31] is excluded due to the unavailability of its source code and high costs. RepoMem requires generating summaries for 200 files per instance using LocAgent [5], incurring a cost of approximately $0.23 per file and $46 per instance with GPT-4o as a backbone LLM. Considering that RepoMem yields a maximum improvement of 4.9% in file-level accuracy over LocAgent [31], while Co-Locator outperforms LocAgent by approximately 30% in average, its exclusion would not affect the overall performance trends observed in our evaluation.

B.3 Environmental settings and runtime

All experiments are conducted using NVIDIA RTX 6000 and NVIDIA H200 with Intel Xeon Gold 6330 CPU. We use vLLM to serve Qwen2.5 in our local environment, while GPT-4o is accessed via the OpenAI API. A fixed random seed is set across all experiments for reproducibility. Under this setup, the average runtime for Co-Locator$^-$ is 0.383 minutes per instance using Qwen2.5 and 0.055 minutes per instance using GPT-4o. For Co-Locator$^+$, the additional reranking process requires 0.597 and 0.051 minutes per instance for Qwen2.5 and GPT-4o, respectively.

C Additional experiments

C.1 Statistical significance of the empirical results

As shown in Table 1 of Section 4.2, Co-Locator outperforms the baseline methods in code localization across all datasets and all evaluation metrics. To verify the statistical significance of Co-Locator's

Table 7: Statistical significance of Co-Locator's improvements over SWERank.

	SWE-Bench-Live						LocBench					
	File		Module		Function		File		Module		Function	
	5	10	10	15	10	15	5	10	10	15	10	15
SWERank$^-$	0.754	0.824	0.709	0.746	0.541	0.574	0.752	0.823	0.675	0.711	0.507	0.557
Co-Locator$^-$	0.765	0.840	0.751	0.795	0.566	0.607	0.813	0.865	0.725	0.758	0.600	0.639
p-value	0.029	0.029	0.009	0.008	0.005	0.000	0.005	0.005	0.001	0.001	0.001	0.002
SWERank$^+$	0.787	0.840	0.754	0.791	0.586	0.623	0.811	0.855	0.729	0.759	0.616	0.646
Co-Locator$^+$	0.813	0.864	0.779	0.808	0.610	0.642	0.841	0.882	0.751	0.787	0.648	0.685
p-value	0.007	0.002	0.004	0.003	0.002	0.003	0.012	0.003	0.010	0.000	0.007	0.005

Table 8: Statistical significance of Co-Locator$^+$'s improvements over its w/o KG variant.

	SWE-Bench-Live						LocBench					
	File		Module		Function		File		Module		Function	
	5	10	10	15	10	15	5	10	10	15	10	15
w/o KG	0.798	0.841	0.749	0.798	0.591	0.614	0.833	0.876	0.742	0.773	0.627	0.663
Co-Locator$^+$	0.813	0.864	0.779	0.808	0.610	0.642	0.841	0.882	0.751	0.787	0.648	0.685
p-value	0.024	0.022	0.012	0.029	0.003	0.009	0.010	0.038	0.029	0.017	0.002	0.006

improvements over baseline methods, we conduct repeated evaluations against SWERank, the strongest baseline. Specifically, we run each model three times and perform statistical t-tests. As SWERank uses Qwen2.5 as its backbone LLM, we evaluate Co-Locator under the same backbone LLM to ensure a fair comparison. As detailed in Table 7, Co-Locator$^+$ and Co-Locator$^-$ consistently outperform their corresponding baselines, SWERank$^+$ and SWERank$^-$, respectively. Furthermore, the performance gaps are statistically significant with $p < 0.05$ across all datasets and metrics.

In Section 4.3, Table 2 demonstrates the benefits of leveraging knowledge graphs, with Co-Locator$^+$ achieving higher localization accuracy than its w/o KG variant. To assess the statistical reliability of these gains, we evaluate both models across three independent runs with different random seeds and conduct statistical t-tests. Table 8 shows that Co-Locator$^+$ consistently outperforms its w/o KG variant, with statistically significant improvements ($p < 0.05$) across all datasets and metrics.

C.2 Alternative designs of KG agent

While the w/o KG variant serves as our main reference point for evaluating the overall impact of the knowledge graph, we also conduct a fine-grained analysis to explore alternative design choices for the KG Agent. Specifically, we evaluate the following three alternatives that represent different strategies for integrating historical knowledge into code localization:

- Under **Dense Retrieval over Records**, the KG Agent performs standard dense semantic retrieval over issues, PRs, and commits.
- Under **Historical Relationships Only**, the knowledge graph includes only structural links among historical records and omits contextual triplets extracted via an LLM. In this setting, the KG Agent still has access to the contents of issues, PRs, and commits.
- Under **Contextual Triplets Only**, the knowledge graph consists solely of contextual triplets extracted from historical records via an LLM.

To ensure a fair comparison, all methods are evaluated on SWE-Bench-Live using Qwen2.5 as the backbone LLM. Table 9 shows the average localization results across three independent runs, where the performance of all three alternatives lies between that of the w/o KG variant and Co-Locator. Interestingly, the performance of Historical Relationships Only is comparable to that of Dense Retrieval over Records. This occurs because the textual content of historical records often contains implicit references to related records, enabling text retrieval to capture context similar to that of

Table 9: Results of Co-Locator$^+$, w/o KG variant, and alternative KG agent designs.

	File		Module		Function	
	5	10	10	15	10	15
w/o KG	0.798	0.841	0.749	0.798	0.591	0.614
Dense Retrieval over Records	0.809	0.850	0.765	0.800	0.592	0.627
Historical Relationships Only	0.802	0.852	0.758	0.799	0.594	0.623
Contextual Triplets Only	0.803	0.848	0.755	0.801	0.593	0.627
Co-Locator$^+$	**0.813**	**0.864**	**0.779**	**0.808**	**0.610**	**0.642**

structural links. Meanwhile, the results of Contextual Triplets Only show that fine-grained semantic relations can provide effective localization clues. Ultimately, Co-Locator$^+$ achieves the highest accuracy across all evaluation metrics, demonstrating that integrating historical relationships with fine-grained contextual triplets is beneficial. These findings suggest that leveraging development history itself provides valuable complementary evidence for code localization. Our framework further builds on this benefit by integrating historical relationships and fine-grained contextual knowledge into a unified knowledge graph for code localization.

D Prompts

This section presents the prompts used for knowledge graph construction, including summary generation (§ D.1), entity extraction (§ D.2), and triplet extraction (§ D.3). Prompts for the CG, KG, and Retriever agents are also provided (§ D.4).

D.1 Prompts for summary generation

System prompt for issue and PR summary generation.

You are an expert software engineer. Generate a detailed but grounded technical report of a GitHub Issue or Pull Request. The report will later be used to build a Knowledge Graph for debugging and repair planning.

STRICT RULES

---

1. Use the exact Markdown template below.
2. Do NOT output JSON.
3. Preserve exact spelling and casing of identifiers (files, functions, classes, variables, APIs).
4. Do not invent information. If something is not stated in the text, write "None".
5. You may summarize technical context in your own words only when it is clearly supported by the text.
6. Do not add new sections or modify the template structure.

OUTPUT MARKDOWN TEMPLATE

ISSUE_REPORT
Problem
Provide a clear description of the problem at the system behavior level (2–4 sentences). Explain what functionality is affected and what kind of failure occurs.

Failure Context

Describe where the failure occurs in the system (1–3 sentences). Mention the relevant component, workflow step, or runtime context if known.

Intended Behavior
Explain the expected behavior or invariant of the system (1–2 sentences).

Actual Behavior
Describe the incorrect behavior or failure that occurs (1–3 sentences). Mention symptoms such as incorrect outputs, thrown exceptions, or unexpected states.

Trigger
Describe the conditions or sequence of events that lead to the issue (1–3 sentences).

Technical Clues
Summarize important hints that help understand the failure (2–4 sentences). This may include logs, stack traces, observations from discussion, or suspected components.

CODE_CONTEXT
Files:
- <exact file paths mentioned. one per line.>

Symbols:
- <exact class/function/module/variable names. one per line.>

Evidence:
- <error message, stack trace fragment, log snippet, or line number. one per line.>

REPAIR_HINT
Describe the design-level repair direction discussed in the issue or PR comments (1–3 sentences). If no repair strategy is mentioned, write "None".

User prompt for issue and PR summary generation.

You are given a structured timeline of a GitHub Issue or Pull Request.

Entries may include:
- the original report
- comments or discussion
- code snippets
- logs or stack traces
- patch explanations

DATA:
{content}

TASK:
Extract the key technical information needed to understand the problem and locate the related code.

Focus on:
- the system behavior being discussed
- expected vs actual behavior
- triggers or runtime conditions
- technical clues such as logs or stack traces
- file paths and identifiers
- any repair strategy mentioned in the discussion

Provide explanatory context when it helps clarify the failure mechanism, but only when it is clearly supported by the text. Only include information supported by the provided text. If a section cannot be filled, write "None". Produce the result using the Markdown template defined in the system prompt.

System prompt for commit summary generation.

You are an expert software engineer. Generate a detailed but grounded technical report of a Git commit. The report will be used to construct a Knowledge Graph describing code changes and repair patterns.

STRICT RULES

1. Use the exact Markdown template below.
2. Do NOT output JSON.
3. Preserve exact spelling and casing of identifiers (files, functions, classes, variables).
4. Do not invent information. If something is not explicitly present in the commit message or diff, write "None".
5. You may summarize technical context in your own words only when it is clearly supported by the commit message or diff.
6. Only describe component roles when they are clearly implied by the code or commit message.
7. Do not add new sections or modify the template structure.

OUTPUT MARKDOWN TEMPLATE

COMMIT_REPORT
Change
Describe the main change introduced by the commit (2–4 sentences). Explain what functionality or behavior is modified.

Intent
Explain why the change was made (1–3 sentences). If the intent is not explicitly stated, write "None".

Change Type
<bug fix / refactor / feature / documentation / configuration / other>

Component Role
Describe the role of the modified component or symbol in the system (1–3 sentences). If the role cannot be inferred from the commit message or diff, write "None".

Modification Semantics
Explain the semantic nature of the code change (2–4 sentences). Describe how the logic, conditions, validation, or behavior of the code was altered.

Key Modifications
Describe the most important concrete code modifications. Use multiple bullet points if several changes occur.

CODE_CONTEXT
Files:
- <exact file paths modified. one per line.>

Symbols:

- <exact class/function/module/variable names. one per line.>

Evidence:
- <important diff fragment or commit message phrase. one per line.>

User prompt for commit summary generation.

You are given information about a Git commit.

The input may include:
- the commit message
- modified file paths
- code diffs

DATA:
content

TASK:
Extract the key technical information describing the change.

Focus on:
- what code behavior or logic was changed
- why the change was made (if stated)
- where the change occurs in the codebase
- which files and identifiers were modified
- technical signals from the diff such as condition changes, validation logic, boundary checks, or error handling

Describe the role of modified components and the semantic type of modification when this can be clearly inferred from the commit message or diff. Provide explanatory context when it helps clarify the code change. Only include information supported by the provided content. If a section cannot be filled, write "None". Produce the result using the Markdown template defined in the system prompt.

D.2 Prompts for entity extraction

System prompt for entity extraction.

You are an expert data engineer specialized in Knowledge Graph construction for code localization. Your task is to extract as many distinct ENTITIES as possible from a GitHub Issue/PR/Commit summary in order to maximize Knowledge Graph coverage for debugging and repair reasoning.

GOAL:
Identify identifiers, technical elements, software concepts, operations, and failure signals that can serve as nodes in a Knowledge Graph. Prioritize coverage and completeness when extracting entities.

OUTPUT FORMAT (STRICT)

- Group entities by their TYPE.
- Each entity must follow the format: PREFIX:CANONICAL_ENTITY_NAME
- If no entity exists for a TYPE, skip that section entirely.
- Use the following Markdown format:

TYPE_NAME
- PREFIX:ENTITY_NAME
- PREFIX:ENTITY_NAME

- Do NOT introduce new types or prefixes.
- Do NOT output JSON.
- Do NOT add explanations.

_____

ENTITY TYPES & NAMING CONVENTIONS

_____

1) ISSUE / PR / COMMIT (ISSUE:*, PR:*, COMMIT:*)
- Use the identifier mentioned in the summary.
2) FILE (FILE:*)
- Use the exact full file path exactly as written.
3) SYMBOL (SYM:*)
- Use the exact code identifier for functions, classes, methods, or modules. Do NOT change spelling or casing.
4) ENVIRONMENT (ENV:*)
- Use short canonical tokens describing runtime or operating systems.
5) DEPENDENCY (DEP:*)
- Use short canonical tokens for external libraries or frameworks.
6) CONFIG (CFG:*)
- Use configuration flags, environment variables, or runtime settings.
7) ERROR / SYMPTOM (ERR:*)
- Use short canonical tokens representing failures or incorrect behavior.
8) CONCEPT / OPERATION (CONCEPT:*, OP:*)
- CONCEPT represents software or system concepts. OP represents logical transformations or operations.

_____

ENTITY EXTRACTION STRATEGY

_____

Extract entities from all relevant parts of the summary.

For each sentence, attempt to identify:
- code artifacts (files, symbols)
- failures and error conditions
- operations and logic changes
- software concepts
- configuration or environment context
- dependencies or frameworks

_____

STRICT RULES

_____

1. **CANONICAL IDENTIFIERS:** Normalize phrases with the same meaning into one canonical entity.
2. **EXACT CODE IDENTIFIERS:** For FILE and SYMBOL entities, preserve the exact spelling and casing.
3. **NO DUPLICATES:** Each entity must appear only once.
4. **ATOMICITY:** Each entity should represent a single concept.
5. **PREDEFINED TYPES ONLY:** Use only the entity types and prefixes defined above.
6. **NO REPETITION:** Do not repeat yourself.

User prompt for entity extraction.

Extract as many distinct entities as possible from the following GitHub Issue/PR/Commit summary to provide a comprehensive foundation for triplet extraction. Follow the entity types, prefixes, and canonicalization rules defined in the system prompt.

Focus on identifying entities related to:
- file paths
- code symbols (functions, classes, modules)
- errors or failure modes
- software concepts
- logical operations or code modifications
- environment or dependency context

Convert natural language descriptions into canonical entity tokens using the prefixes: ERR:*, CONCEPT:*, OP:*. If a sentence describes multiple technical ideas, extract multiple entities.

SUMMARY TO ANALYZE:
{target}

{summary}

List all extracted entities using the Markdown template defined in the system prompt. Prefer extracting many valid entities rather than missing important ones. Do not repeat entities. Do not introduce new prefixes or types.

D.3 Prompts for triplet extraction

System prompt for triplet extraction.

You are an expert knowledge engineer specializing in semantic Knowledge Graph construction for software debugging and repair planning. Your task is to extract as many semantic TRIPLETS as possible from a GitHub Issue/PR/Commit summary for Knowledge Graph construction.

STRICT RULES

---

1. **Entity Anchor**: Prioritize entities from the provided entity list. Use the format 'PREFIX:IDENTIFIER'.
2. **New Entities**: You may introduce additional entities if they appear explicitly; they must follow the same prefix rules (e.g., SYM:new_func).
3. **Relation Format**: Relations MUST be a single word or underscore-separated word in **ALL UPPERCASE**. Do NOT include TYPE prefixes in the relation (e.g., Use `REPORTS` instead of `ISSUE_REPORTS_ERR`).
4. **Semantic Focus**: Focus on behavioral, causal, or repair-related relationships (e.g., TRIGGERS, FIXES, DEPENDS_ON).
5. **No Conversational Filler**: Output ONLY the triplets. No explanations, headers, or markdown formatting outside the triplets.

ALLOWED ENTITY PREFIXES

ISSUE, PR, COMMIT, FILE, SYM, ENV, DEP, CFG, ERR, CONCEPT, OP

RELATION EXAMPLES

_____

- Contextual: REPORTS, ADDRESSES, MODIFIES, IMPLEMENTS
- Behavioral/Causal: OCCURS_IN, TRIGGERS, RELATED_TO, AFFECTS
- Repair-oriented: FIXES, MITIGATES, IMPROVES, RESOLVES
- Dependency/Env: USES, DEPENDS_ON, TARGETS, REQUIRES

If no example fits, create a new relation.

_____

FORMAT VALIDATION

_____

Each output line MUST follow exactly:
ENTITY<|#|>RELATION<|#|>ENTITY

Where:
ENTITY must start with one of the allowed prefixes above.

VALID EXAMPLES (FORMAT ILLUSTRATIVE ONLY):
ISSUE:123<|#|>REPORTS<|#|>ERR:ValueError
ERR:ValueError<|#|>OCCURS_IN<|#|>SYM:replace
COMMIT:abc123<|#|>MODIFIES<|#|>FILE:src/write.py

If a triplet does not satisfy this format, DO NOT output it.

INVALID EXAMPLES (DO NOT PRODUCE):
SYM:replace<|#|>USE (Not a triplet) PR:2006<|#|>MODIFIES<|#|>OP (Unclear entities)
OCCURS_IN<|#|>ERR:NotImplementedError<|#|>SYM:MyModule (each triplet should be written as `head<|#|>relation<|#|>tail`)

_____

OUTPUT FORMAT

_____

One triplet per line:
HEAD_ENTITY<|#|>RELATION<|#|>TAIL_ENTITY

User prompt for triplet extraction.

Extract as many semantic triplets as possible from the following GitHub Issue/PR/Commit summary for Knowledge Graph construction. Follow the entity prefixes, relation rules, and output format defined in the system prompt.

Focus on identifying relationships related to:
- what error or failure occurred
- where the failure occurred
- which file or symbol is affected
- what concept or operation is involved
- what change was introduced by the commit or PR
- what concept or operation fixes or mitigates the error

Use the provided entity list as the primary reference when constructing triplets. If an important entity appears in the summary but is missing from the entity list, you may introduce that entity.

SUMMARY TO ANALYZE:
{target}

{summary}

PROVIDED ENTITY LIST:

{entities}

List all extracted triplets using the output format defined in the system prompt. Do not repeat triplets. Do not introduce new entity prefixes.

D.4 System prompts of agents

System prompt of the CG agent.

You are a Senior Software Engineer specializing in **Fault Localization**. Your mission is to pinpoint the exact python functions that must be modified to fix a reported bug.

STRICT RULE: NO BLIND GUESSING
- You do not have direct access to the source code or repository history.
- You MUST confirm every hypothesis by querying the **KG Agent** (for history/context).

Strategic Tool Usage
1. `search_cg(query)`:
- **Semantic Search**: Search by name (e.g., `process_data`) or description (e.g., "error handling in upload").
- If results are weak, rephrase the query into a descriptive sentence.

2. `traverse_cg(node, query)`:
- **Contextual Expansion**: Trace relationships like "Who calls this?" (inverse relations) or "What methods are in this class?".
- Crucial for tracing bug propagation through the system.

3. `inspect_file(node)` / `inspect_class(node)` / `inspect_func(node)`:
- **Detailed Analysis**: Use these to get the signature or implementation of a specific node.
- **Strict Input**: You MUST enter the node name exactly as represented in the Code Graph.

4. `ask_kg_agent(query)`:
- You can ask the KG agent to find the historical context and provide relevant contents.

Tool Usage Rules
1. **One-at-a-Time**: Call only ONE tool per turn.
2. **Sequential Reasoning**: Wait for the tool response to update your mental model before the next move.
3. Do not batch multiple tool calls in a single message.
4. You can use tools up to 30 times.

Inter-Agent Communication Rules
- Ask the **KG Agent** to understand the **historical context**, **evolution**, and **developer intent**. Example queries include:
- Locate Issues, PRs, or Commits that are relevant to an error.
- Identify relevant issues and summarize their problem settings.
- Identify and summarize how a specific problem was previously resolved.
- Trace the previous changes and evolution of a specific symbol.

Step-by-Step Localization Process
Step 1: Contextual Grounding (KG + CG)
- Identify the core symbols and entry points.
- **KG Inquiry**: Ask the KG Agent for similar past issues, bug reports, or recent PRs that modified the suspicious area.

Step 2: Structural & Historical Mapping

- Use the KG Agent to see who usually fixes this area.
- Use tools to see how components interact.
- Identify candidate modules in: `FUNC:file_path:QualifiedName` format.

Step 3: Deep Logic & Intent Analysis
- Inspect the code of top candidates, and decide whether the changes should be made.
- **Intent (KG)**: If the logic seems intentional but causes a bug, ask the KG Agent to understand the developer's reasoning.

Step 4: Comprehensive Candidate Identification
- Rank candidates by **Probability (Importance)**:
1. **Direct Cause**: Where the logic fails.
2. **Regression/Historical**: Where similar bugs occurred.
3. **Indirect/Data Flow**: Where bad data originates or propagates.
- Provide a comprehensive list; do not prematurely narrow down your search.

Output Format for Final Results:
Your final output should list **all identified python functions** requiring modification or investigation, wrapped with triple backticks ‘‘‘. Order them by **importance/probability**, starting with the most likely cause. Provide an extensive list (aim for a thorough coverage of the issue).

Examples (format illustrative only):
‘‘‘
file: path/to/file1.py
class: MyClass1
function: my_function1

file: path/to/file2.py
class: MyClass2
function: my_function2

file: path/to/file3.py
function: my_function3
‘‘‘

Return just the location(s).

System prompt of the KG agent.

You are the **Knowledge Graph (KG) Navigator**. Your mission is to act as the memory and reasoning interface between the Localization Agent and the repository's historical knowledge graph. You provide **evolutionary context** by connecting Issues, Pull Requests, Commits, and Semantic Entities.

Your Goal
Help the Localization Agent understand the "Why" and "How" behind the code by exploring historical patterns, previous bug fixes, and developer intentions stored in the Knowledge Graph.

Strategic Tool Usage
1. `search_kg(query)`:
- **Semantic Search**: Search relevant entities based on natural language descriptions (e.g., "memory leak in async loops" or "regression after v2.1.0").
- If results are weak, rephrase the query into a descriptive sentence.

2. `traverse_kg(node, query)`:

- **Contextual Expansion**: Trace semantic relationships between issues, PRs, commits, symbols, and concepts.
- Find which PR resolved a specific Issue, or which Commits are linked to a suspicious discussion. This is essential for finding "Precedent Fixes."

3. `inspect_issue(node)` / `inspect_pr(node)` / `inspect_commit(node)`:
- **Detailed Analysis**: Use these to get the summary of a specific issue/PR or a patch of a specific commit.
- **Input Format**: To inspect an issue or a pr, the input format must be `ISSUE:#` or `PR:#`. To inspect a commit, the input format must be `COMMIT:#`.

Execution Strategy
- **Sequential Execution**: Call only one tool at a time. Do not attempt to call multiple search or inspect tools in a single turn.
- **Reasoning First**: Before calling a tool, briefly state why it is necessary.
- **Synthesize**: Don't just dump output. Explain **why** the results are relevant to the bug.
- **Tool Call Limit**: You can use tools up to 5 times.

Intent Interpretation
The Localization Agent will seek historical or semantic clues. Your job is to:
1. Identify the **Goal**: (e.g., "Finding similar past bugs", "Understanding PR rationale", "Tracing entity evolution")

2. Choose the **Best Tool**:
- To find concepts or specific discussions, `search_kg`
- To see semantic relationships among issue/pr/commits, concepts, and symbols, `traverse_kg`
- To deep-dive into a specific issue/pr/commit, `inspect_issue`, `inspect_pr`, or `inspect_commit`

3. **Execute and Summarize**: Provide a concise answer for the Localization Agent's question.

Example Reasoning Flows: KG Navigator (Knowledge Graph)
Use these flows to understand "Why" the code exists and "What" has happened historically.
1. Goal: Locate Issues, PRs, or Commits that are relevant to an error.
- Use `search_kg` to find historical entities by error messages, symptoms, or feature keywords. Use `traverse_kg` to find commits that resolve the identified Issue or are semantically connected to the identified Issue/PR. Use `inspect_issue`, `inspect_pr`, or `inspect_commit` to read 2-3 most relevant events and summarize.

2. Goal: Identify relevant issues and summarize their problem settings.
- Use `search_kg` to find relevant issues. Use `inspect_issue` to identify the problem setting of issues and summarize the observed results.

3. Goal: Identify and summarize how a specific problem was previously resolved.
- Use `search_kg` to find "Precedent Fixes" by searching for a past bug. Use `traverse_kg` to identify prs or commits that resolve the problem. Use `inspect_pr` or `inspec_commit` to analyze the fix strategy.

4. Goal: Trace the previous changes and evolution of a specific symbol.
- Use `search_kg` to identify the exact name of a specific symbol in knowledge graph. Use `traverse_kg` to identify the relationship between a symbol and the historical events. Use `inspect_commit` to view the specific code patches (diffs) and understand how the symbol has evolved over time.

You are the institutional memory of the Localization Agent. Be insightful, historical, and connective.

System prompt of the Retriever agent in argument correction.

Task
Identify the single most relevant node from the candidate list that matches the user's input, considering the given context.

Rules
1. Provide the **index number** of the selected node only.
2. If no candidate node is a relevant match, you must select the index for "None of the above".
3. Strictly forbidden: Do not output any text, node names, or punctuation other than the single integer.
4. If there are multiple candidates, pick the one with the highest semantic similarity.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We clearly state our main contributions in both Abstract and Introduction: (1) the construction of knowledge graphs integrating interrelationships among development traces (issues, PRs, commits) with contextual triplets extracted by LLMs, and (2) Co-Locator, a collaborative multi-agent code localization framework that consists of three agents: the CG agent, the KG agent, and the Retriever agent. We provide experimental results in Section 4 to demonstrate that Co-Locator significantly outperforms 12 state-of-the-art baselines.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discussed the limitation of Co-Locator in Section 5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The manuscript does not include any theoretical claims.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide all technical specifications necessary for replication, including data processing procedures and detailed hyperparameter configurations (§ 4.1) and experimental environments (§ B.3). Full implementations of Co-Locator are released via the GitHub, with the link specified in Section 4. In addition, our methodology's core operational logic is described in detail to allow for a complete reconstruction of the framework (§ 3.3), and all prompts used in Co-Locator are provided (§ D). In all experiments, we fixed the random seeds and used greedy decoding for the reproducibility of our results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We use two benchmark datasets, LocBench and SWE-Bench-Live, both of which are publicly available and provide the implementations of Co-Locator with detailed instructions via the GitHub. Also, all essential resources required to reproduce the experimental results, including the detailed model architecture (§ 3.3), datasets and hyperparameter configurations (§ 4.1), environmental settings (§ B.3), and prompts (§ D), are provided in both the main paper and technical appendix.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Details regarding the datasets used, the evaluation metrics and their justifications, baseline methods for comparison, and the hyperparameters of Co-Locator are included in Section 4. In-depth technical specifications and additional experimental details are provided in Section B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We conduct statistical t-tests on multiple independent runs to show that our experimental results are statistically significant.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide a comprehensive summary of the computing resources used for our experiments. Specifically, GPT-4o was accessed via the OpenAI API, while Qwen2.5 was served on local GPUs. Detailed hardware specifications for the local environment are specified in Appendix B. Furthermore, the cost analysis, including the average API cost per instance and the overall efficiency of Co-Locator, is presented in Section 4.5 and the average runtime of Co-Locator is provided in Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: All authors have thoroughly reviewed the NeurIPS Code of Ethics and confirm that our research adheres to all its guidelines. Our study utilizes publicly available benchmarks and open-source GitHub repositories, ensuring that no human subjects or private/sensitive data were involved. Furthermore, we have taken all necessary precautions to maintain anonymity throughout the submission process.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: Co-Locator is a foundational tool for automated software maintenance and code localization. As our work focuses on improving developer productivity through technical means within a controlled software engineering context, its societal impacts do not extend beyond the general scope of automated programming research.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper does not release high-risk generative models or uncurated datasets. We use established benchmarks and standard LLM interfaces for code localization in controlled environments, posing no significant risk of misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.

- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We have properly cited all existing assets, including baselines, backbone LLMs, embedding models, and benchmark datasets, along with their specific versions in Sections 2 and 4. Furthermore, the licenses and terms of use for each asset are explicitly detailed in Appendix B.1 to ensure full compliance with their original requirements.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide the implementations of Co-Locator via GitHub. The repository includes a README file that contains detailed documentation and instructions for reproducing Co-Locator.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

Justification: This paper does not involve any crowdsourcing experiments or research conducted with human subjects. All experiments were performed using automated frameworks and publicly available datasets.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: This research does not involve human subjects, and therefore, Institutional Review Board (IRB) approval was not needed. All experiments were performed using automated frameworks and publicly available datasets.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: We utilize GPT-4o-mini for triplet extraction during knowledge graph construction, and employ GPT-4o-2024-08-06 and Qwen2.5-Coder-32B-Instruct as the backbone models for our CG, KG, and Retriever Agents. The specific roles, prompts, and configurations for each LLM are detailed in Section 4 and Appendices B.3 and D. Additionally, LLMs were used for minor grammatical corrections during the writing process.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.