SweRank: Software Issue Localization with Code Ranking

Anonymous authors
Paper under double-blind review

Abstract

Software issue localization, the task of identifying the precise code locations (files, classes, or functions) relevant to a natural language issue description (e.g., bug report, feature request), is a critical yet time-consuming aspect of software development. While recent LLM-based agentic approaches demonstrate promise, they often incur significant latency and cost due to complex multi-step reasoning and relying on closed-source LLMs. Alternatively, traditional code ranking models, typically optimized for query-to-code or code-to-code retrieval, struggle with the verbose and failure-descriptive nature of issue localization queries. To bridge this gap, we introduce SweRank, an efficient and effective retrieve-and-rerank framework for software issue localization. To facilitate training, we construct SweLoc, a large-scale dataset curated from public GitHub repositories, featuring real-world issue descriptions paired with corresponding code modifications. Empirical results on SWE-Bench-Lite and LocBench show that SweRank achieves state-of-the-art performance, outperforming both prior ranking models and costly agent-based systems using closed-source LLMs like Claude-3.5. Further, we demonstrate SweLoc's utility in enhancing various existing retriever and reranker models for issue localization, establishing the dataset as a valuable resource for the community.

1 Introduction

The scale and complexity of modern software systems continue to grow exponentially, with a significant portion of development effort dedicated to identifying and resolving software issues. This has fueled growth in automated software issue fixing (Cognition AI, 2024), with recent LLM-based patch generation (Yang et al., 2024a; Gauthier, 2024) solving real-world issues on benchmarks such as SWE-Bench (Jimenez et al., 2023), and commercial copilots integrating "one-click" quick-fix suggestions directly into IDEs (Microsoft, 2023; Cursor, 2025; Windsurf, 2025). Central to the process of fixing software issues is the task of **issue localization**: accurately identifying *where* in the codebase the necessary changes should be made. This involves pinpointing the specific files, classes, or functions relevant to a given issue description, typically provided in natural language (e.g., a bug report). Effective localization is critical; without correctly identifying the relevant code segments, any subsequent attempt at automated repair is likely to fail or, worse, introduce new faults.

Given the importance of localization, recent work treats it as an agentic reasoning problem (Yao et al., 2023) and has investigated the use of sophisticated LLM-based agents (Yang et al., 2024b; Yu et al., 2025; Chen et al., 2025) that issue commands such as 'read-file', 'grep' and 'traverse-graph' to iteratively explore codebases, navigate file structures, search for code patterns, and analyze dependencies. While powerful, these agent-based compound systems often involve multiple rounds of interaction (≈7–10 on average) with large models and complex reasoning processes, which can incur considerable API costs (≈$0.66 per example with Claude-3.5) at high latency. Moreover, agent traces are brittle: they rely on temperature sampling and require complex tool orchestration.

An alternative, more efficient strategy is to frame issue localization as an information retrieval problem, specifically using code ranking models (Yue et al., 2021; Zhang et al., 2024; Suresh et al., 2024). Such models can directly rank candidate code snippets (e.g., functions or files) based on their relevance to a given natural language query, and quickly score and sort potential locations within a large codebase. However, prior code ranking models are still inferior in performance as they have predominantly been optimized for tasks distinct from issue localization.

These typically include query-to-code retrieval (Li et al., 2024a), which aims to find code implementing a described functionality, and code-to-code retrieval (Wang et al., 2023a; Li et al., 2024b), focused on identifying semantically similar code fragments. The task of issue localization presents unique characteristics; input queries (issue descriptions) are often substantially more verbose than typical NL-to-code queries[1] and, more crucially, issues tend to describe observed erroneous behavior or system failures rather than specifying desired functionality. This fundamental difference in query nature and intent suggests that models trained on conventional code retrieval data (Husain et al., 2019; Suresh et al., 2024) may not be optimally suited for issue localization.

To bridge this gap, we introduce SWERANK, a code ranking framework trained specifically for software issue localization. SWERANK employs a standard yet effective retrieve-and-rerank architecture, comprising two core components: (1) SWERANKEMBED, a bi-encoder embedding model serving as the code retriever; and (2) SWERANKLLM, an instruction-tuned LLM serving as a code reranker. To train SWERANK, we construct SWELOC, a new large-scale issue localization dataset curated from public Github repositories, providing realistic training examples. SWERANKEMBED is trained using a contrastive objective, where the issue descriptions serve as queries, the known localized functions act as positive examples, and carefully mined code snippets from the same repository function as hard negatives. Subsequently, SWERANKLLM is trained as a list-wise reranker (Reddy et al., 2024); it takes as input the issue description alongside the top-K candidates retrieved by SWERANKEMBED and predicts an improved ranking permutation, thereby enhancing the final localization.

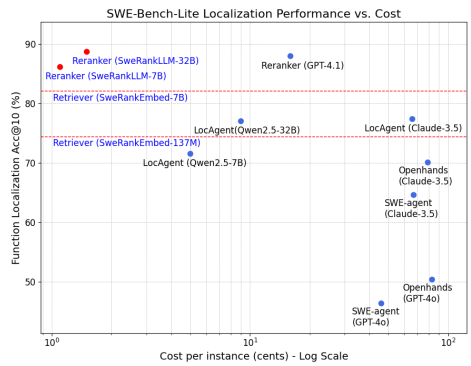


Figure 1: Comparison of localization performance versus cost per instance on SWE-Bench-Lite. Our proposed SWERANKEMBED retriever and SWERANKLLM reranker models achieve superior accuracy at a significantly lower cost compared to agent-based localization methods.

Empirical results demonstrate that SWERANK achieves state-of-the-art performance for file, module and function-level localization on Swe-Bench-Lite (Jimenez et al., 2023) and LocBench (Chen et al., 2025). Further, we show that SWERANK, built on open-source models, has a considerably better performance to cost ratio compared to agent-based approaches that employ closed-source LLMs like Claude-3.5 (Anthropic, 2023), as illustrated in Figure 1. Finally, we demonstrate the effectiveness of our SWELOC data by showing that it consistently improves localization performance when used for finetuning a variety of text and code-pretrained retriever and reranker models.

2 Related Work

2.1 Software Issue Localization

Software issue localization or Fault Localization (FL) aims to identify the specific code locations responsible for reported bugs. Traditional fault localization methods (Wong et al., 2016) can be grouped into spectrum-based and program-analysis approaches. Spectrum-based fault localization (SFL) (de Souza et al., 2016; Amario de Souza et al., 2024) statistically associates test outcomes with executed code elements to rank statements or functions by their 'suspiciousness' based on passing and failing test coverage. Complementary static and dynamic analyses exploit program structure–through call-graph traversal (Adhiselvam et al., 2015), dependency analysis (Elsaka, 2017), or program slicing (Soremekun et al., 2021)–to constrain the search space of potential bug locations. While these methods provide a statistical basis for finding faults, they require precise program models and cannot leverage the rich natural language context in bug reports.

Modern approaches instead use LLM-based agent frameworks that treat bug localization as a planning and searching problem. AgentFL (Qin et al., 2024) incorporates a multi-agent system with a three step procedure involving interpreting the bug context, traversing the codebase and verifying the suspected fault. OpenHands (Wang et al., 2025) and SWE-Agent (Yang et al., 2024b) use bash commands

[1] 460 tokens in SWE-Bench (Jimenez et al., 2023) issues vs 12 tokens in CSN (Li et al., 2024a) queries.

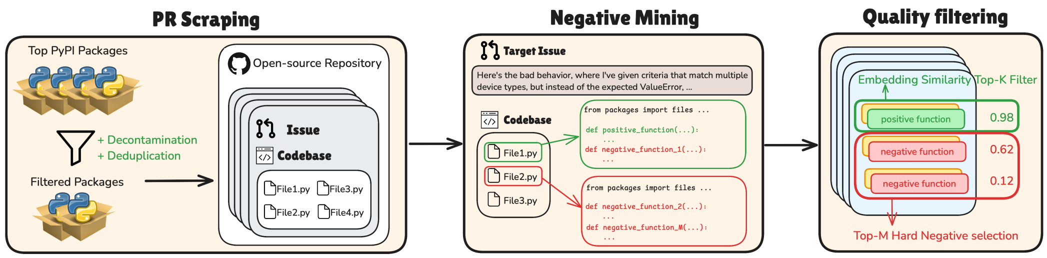


Figure 2: Overview of SweLoc data construction pipeline, illustrating the three main stages.

or custom interfaces to navigate repositories and access files. Other agentic systems combine IR with tool use: MoatlessTools (Örwall, 2024) integrates a semantic code search engine into an agent's loop to guide it to relevant files. More recently, LocAgent (Chen et al., 2025) constructs a graph of the codebase for an LLM agent to do multi-hop reasoning over code dependencies. While these agent-driven approaches have achieved impressive results, they incur substantial costs and have high latency. Agent-based methods must orchestrate multiple steps of reasoning and tool use, which makes them brittle; a single failure in the chain (e.g., a misleading intermediate query or an incomplete code observation) can derail the entire localization process. SweRank instead formulates issue localization as a single-shot ranking problem, which is highly efficient and cost-effective.

2.2 Code Ranking

Transformer-based code ranking models (Wang et al., 2023c; Zhang et al., 2024; Günther et al., 2023; Suresh et al., 2024) have set state-of-the-art on a variety of code retrieval tasks (Li et al., 2024a;b) by learning joint embeddings of text and code. Wang et al. (2023c) and Zhang et al. (2024) learn improved code representations by incorporating a mix of training objectives, such as span denoising, text-code matching and causal LM pretraining, over large-scale code corpora such as CodeSearchNet (Husain et al., 2019) and The Stack (Kocetkov et al., 2022). Suresh et al. (2024) improve the contrastive training process between function snippets and associated doc-strings with better consistency filtering and harder negative mining. Liu et al. (2024b) incorporate multi-task contrastive data that includes code contest generation (Billah et al., 2024), code summarization (Sontakke et al., 2022), code completion (Liu et al., 2024a), code translation (Pan et al., 2024) and code agent conversation (Jin et al., 2024). However, prior code ranking models rarely include error logs in their training data and are not optimized for issue localization, where queries are verbose bug reports rather than precise functionality requests. In contrast, SweRank is explicitly trained on SweLoc, a new automatically collected set of real-world issue reports paired with known buggy functions. By optimizing a bi-encoder retriever and a listwise LLM reranker on this task-specific data, SweRank directly aligns verbose bug descriptions with faulty code, thereby improving localization accuracy.

3 SweLoc: Issue Localization Data

Existing code retrieval datasets (Husain et al., 2019; Suresh et al., 2024) are generally valuable for tasks like NL-to-code search which mainly requires functionality matching. However, they are sub-optimal for training models aimed at software issue localization. The nature of software issues–often detailed descriptions of failures rather than concise functional specifications–necessitates a dataset that accurately reflects this challenge of precisely identifying the problematic functions. To address this gap and provide a suitable training ground for our SweRank framework, we constructed SweLoc, a novel large-scale dataset specifically curated for the task of localizing code snippets relevant to software issues. SweLoc is derived from real-world software development activities captured in public GitHub repositories. Our methodology comprises three main phases: (1) identifying and filtering relevant pull requests (PRs) from popular Python repositories (§3.1), (2) processing these PRs to extract issue descriptions paired with their corresponding code modifications (§3.2), and (3) applying consistency filtering and hard-negative mining to enhance the quality of training instances (§3.3). An overview of this process is shown in Figure 2.

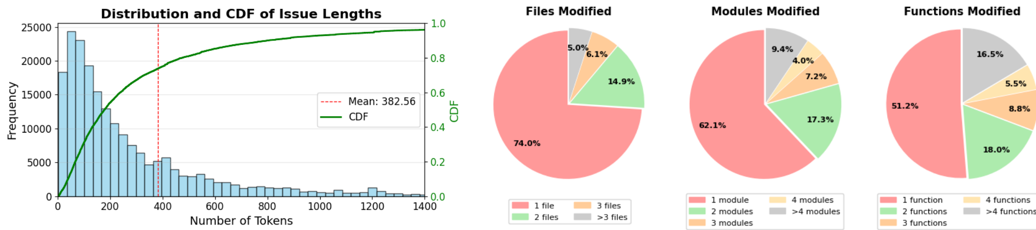


Figure 3: (Left) Distribution of query lengths in the SWELOC dataset. The red dashed line indicates a mean query length of 382.56 tokens, underscoring the detailed nature typical of issue reports. (Right) Distribution of the number of (a) files, (b) modules, and (c) functions modified per GitHub issue. This highlights that while many localizations are concentrated, a significant number span multiple code units, particularly at finer granularities.

3.1 IDENTIFYING RELEVANT PRS

Our data collection involves selecting the repositories associated with the top 11,000 PyPI packages on GitHub. To ensure repository quality and relevance to our task, we apply several filtering criteria. Repositories are required to contain at least 80% Python code. To prevent data leakage and overlap with existing benchmarks, we exclude repositories already present in SWE-Bench (Jimenez et al., 2023) and LocBench (Chen et al., 2025). Finally, we perform deduplication based on source code overlap to remove near-identical repositories. This process results in a curated set of 3387 repositories.

Following the SWE-Bench methodology, we identify pull requests (PRs) within these repositories that (1) resolve a linked GitHub issue and (2) include modifications to test files, indicating the issue resolution was verified. For each such PR, we collect the issue description and the codebase snapshot at the PR's base commit. This procedure results in 67,341 initial $(\text{PR}, \text{codebase})$ pairs. Figure 3 provides further details on the dataset's composition, including query and repository edit distributions.

3.2 LOCALIZATION PROCESSING

Using the collected $(\text{PR}, \text{codebase})$ pairs, we create contrastive training data in the form of $\langle$query, positive, negatives$\rangle$ tuples. For each tuple, the issue description serves as the query. Each function modified within the PR is designated as a positive example, corresponding to a distinct training instance. Thus, a PR modifying N functions yields N training instances. The negatives for each instance come from the unmodified functions within the corresponding codebase. This initial set of instances are further refined via consistency filtering and hard-negative mining, as described next.

3.3 CONSISTENCY FILTERING AND HARD NEGATIVES

The quality of $\langle$query, positive, negatives$\rangle$ tuples used for training significantly impacts the ranking model performance (Suresh et al., 2024). Effective contrastive learning requires relevant positives and challenging negatives (semantically similar to the positive but irrelevant to the query). However, issue descriptions in open-source repositories can be vague, leading to noisy signals for relevance between the issue descriptions and associated code modifications when directly used for training.

To mitigate this, we employ filtering and mining techniques following recent work (Günther et al., 2023; Suresh et al., 2024). First, we apply top-K consistency filtering (Suresh et al., 2024) to retain only instances where the positive code snippet is semantically close to the query relative to other code snippets in the repository. Formally, given an instance i with issue description t_i, a positive function c_i, and the set of other unrelated functions F_i in the repository, we use a pre-trained embedding model (CODERANKEMBED (Suresh et al., 2024)) to compute similarities between t_i, c_i and all functions in F_i. Instance i is retained only if c_i ranks within the top K functions in F_i, based on similarity to t_i. We set $K = 20$, with ablation studies in §5.3.1.

Beyond filtering for relevance of positive pairs, incorporating challenging negatives is crucial for enabling the model to distinguish between semantically similar instances (Moreira et al., 2024). To this end, we employ a hard negative mining strategy that leverages the previously computed similarities to select a set of hard negatives $B_i = \{c_j^-\}_{j=1}^{M}$ for each instance i. These negatives c_j^- are chosen from F_i such that they are among the top M (=15) most similar functions to the query t_i.

4 SWERANK METHODOLOGY

In this section, we present our proposed ranking framework for software issue localization. SWERANK adopts a two-stage retrieve-and-rerank approach with two key components: (1) SWERANKEMBED, a bi-encoder retriever that efficiently narrows down candidate code snippets from large codebases; and (2) SWERANKLLM, a listwise LLM reranker that refines these initial results for improved localization accuracy. Next, we elaborate on the architecture and training objectives for these components.

4.1 SWERANKEMBED

The retriever component, SWERANKEMBED, utilizes a bi-encoder architecture (Reimers & Gurevych, 2019) to generate dense vector representations for GitHub issues and code functions within a shared embedding space. Let (t_i, c_i^+) represent a positive pair from the SWELOC dataset, consisting of an issue t_i and the corresponding code function modified c_i^+. The bi-encoder maps these to embeddings (h_i, h_i^+), derived from the last hidden layer of the encoder. For a training batch of size N, let $H = \{h_i^+\}_{i=1}^N$ denote the set of positive code embeddings. Let $H_B = \bigcup_{i=1}^n \{h_{ij}^-\}_{j=1}^M$ be the set of embeddings for the M hard negatives mined for each issue t_i in the batch (as described in §3.3).

SWERANKEMBED is trained using an InfoNCE contrastive loss (Oord et al., 2018). The objective encourages the embedding h_i of an issue to have a higher similarity with its corresponding positive code embedding h_i^+, compared to its similarity with all other h_k^+ embeddings $(k \neq i)$ and all hard negative embeddings h_{kj}^- within the batch. The loss for a single positive pair (h_i, h_i^+) is:

$$\mathcal{L}_{\mathcal{CL}} = -\log\left(\frac{\exp(\mathbf{h}_i \cdot \mathbf{h^+}_i)}{\sum_{\mathbf{h}_k \in (\mathbf{H_B} \cup \mathbf{H})} \exp(\mathbf{h}_i \cdot \mathbf{h}_k)}\right) \tag{1}$$

The denominator sums over the positive embedding h_i^+ itself and $N(M+1)-1$ negative embeddings relative to h_i. During inference, candidate code functions for a given issue description are ranked based on the cosine similarity between their respective embeddings and the issue embedding.

4.2 SWERANKLLM

For the reranking stage, we employ SWERANKLLM, an instruction-tuned LLM for reranking. SWERANKLLM adopts a listwise ranking approach (Pradeep et al., 2023b), which offers better performance than pointwise methods by considering the relative relevance of candidates. Typically, listwise LLM rerankers are trained to process an input consisting of the query and a set of candidate documents, each associated with a unique identifier. The model's training objective is then to generate the full sequence of identifiers, ordered from most to least relevant according to the ground-truth ranking. However, since SWELOC does not provide a ground-truth ranking among the negative functions for the issue t_i, generating a complete target permutation for training is not feasible.

To adapt listwise reranking training to our setting where only the positive is known, we modify the training objective. Formally, let $\mathcal{D} := \{d_i\}_{i=1}^{|\mathcal{D}|}$ be a training dataset of triplets, where each sample $d_i := (t_i, c_i^+, \{c_{i,j}^-\}_{j=1}^M)$ includes a GitHub issue t_i, a relevant positive code c_i^+, and a set of M irrelevant negative codes $\{c_{i,j}^-\}_{j=1}^M$. We first assign a unique numerical identifier from 1 to M+1) to each function in the set $c_i^+ \cup \{c_{i,j}^-\}_{j=1}^M$. Let I_i^+ be the identifier assigned to the positive function c_i^+. Instead of training the model to predict the full ranked list of identifiers, we train it to correctly generate the identifier corresponding to the single positive function, I_i^+. Thereby, the training objective for a given sample d_i is thus simplified to maximizing the likelihood of the first generated (i.e. top-ranked) identifier:

$$\mathcal{L}_{LM} = -\log(P_\theta(I_i^+|x)) \tag{2}$$

where x is the input prompt constructed from the issue t_i and the set of candidate functions $c_i^+ \cup \{c_{i,j}^-\}_{j=1}^M$ along with their assigned identifiers, and P_θ represents the listwise LLM reranker.

During training, we omit the end-of-sequence token after predicting I_i^+ to retain the model's capability to generate full ranked lists for inference, as required by the listwise format. As we show later in our experiments in §5.3.2, our approach enables finetuning any listwise reranker for the software issue localization task, without needing the full candidate ranking ordering for training supervision.

Type	Method	Model	File (%)			Module (%)		Function (%)	
			Acc@1	Acc@3	Acc@5	Acc@5	Acc@10	Acc@5	Acc@10
Agent	MoatlessTools Örwall (2024)	`GPT-4o`	73.36	84.31	85.04	74.82	76.28	57.30	59.49
		`Claude-3.5`	72.63	85.77	86.13	76.28	76.28	64.60	64.96
	SWE-agent Yang et al. (2024b)	`GPT-4o`	57.30	64.96	68.98	58.03	58.03	45.99	46.35
		`Claude-3.5`	77.37	87.23	90.15	77.74	78.10	64.23	64.60
	Openhands Wang et al. (2025)	`GPT-4o`	60.95	71.90	73.72	62.41	63.87	49.64	50.36
		`Claude-3.5`	76.28	89.78	90.15	83.21	83.58	68.25	70.07
	LocAgent Chen et al. (2025)	`Qwen2.5-7B(ft)`	70.80	84.67	88.32	81.02	82.85	64.23	71.53
		`Qwen2.5-32B(ft)`	75.91	90.51	92.70	85.77	87.23	71.90	77.01
		`Claude-3.5`	77.74	91.97	94.16	86.50	87.59	73.36	77.37
Retriever	BM25 (Robertson et al., 1994)		38.69	51.82	61.68	45.26	52.92	31.75	36.86
	Jina-Code-v2 (161M) (Günther et al., 2023)		43.43	71.17	80.29	63.50	72.63	42.34	52.19
	Codesage-large-v2 (1.3B) (Zhang et al., 2024)		47.81	69.34	78.10	60.58	69.71	33.94	44.53
	CodeRankEmbed (137M) (Suresh et al., 2024)		52.55	77.74	84.67	71.90	78.83	51.82	58.76
	SFR-Embedding-2 (7B) (Meng et al., 2024)		58.03	80.29	83.94	70.07	79.20	56.20	64.23
	GTE-Qwen2-7B-Instruct (7B) (Li et al., 2023)		65.33	82.85	89.78	76.28	83.58	63.14	70.44
	SweRankEmbed-Small (137M) (Ours)		66.42	86.50	90.88	79.56	85.04	63.14	74.45
	SweRankEmbed-Large (7B) (Ours)		72.63	91.24	94.16	84.31	89.78	71.90	82.12
+ Reranker	CodeRankLLM (7B) (Suresh et al., 2024)		72.99	89.78	93.80	85.04	90.88	71.90	83.58
	GPT-4.1		82.12	95.62	97.08	93.07	93.43	81.75	87.96
	SweRankLLM-Small (7B) (Ours)		78.10	92.34	94.53	89.05	92.70	79.56	86.13
	SweRankLLM-Large (32B) (Ours)		**83.21**	**94.89**	**95.99**	**90.88**	**93.43**	**81.39**	**88.69**

Table 1: Performance (in %) on SWE-Bench-Lite. The rerankers use SweRankEmbed-Large as the retriever. Gray corresponds to results with closed-source models. Best retriever numbers are in blue, while best overall numbers (except GPT-4.1) are in **bold**.

5 Experiments

The experiments compare SweRank's performance against state-of-the-art agent-based localization methods, and other code ranking models (§5.2). Furthermore, we investigate the impact of our SweLoc dataset, analyzing how its quality controls (such as consistency filtering) and size influence model performance (§5.3.1), and examining its generalizability by evaluating effectiveness in fine-tuning various pre-existing retriever and reranker models for the issue localization task (§5.3.2).

5.1 Setup

Model Training: We train the SweRank models in two sizes: *small* and *large*. All models are finetuned using our SweLoc dataset. SweRankEmbed-Small is initialized with CodeRankEmbed (Suresh et al., 2024), a SOTA 137M code embedding model, while the large variant is initialized with GTE-Qwen2-7B-Instruct (Li et al., 2023), a 7B parameter text embedding model employing Qwen2-7B-Instruct as its encoder. The small version of SweRankLLM is initialized with CodeRankLLM (Suresh et al., 2024), a 7B parameter code-pretrained listwise reranker. The large version is initialized with Qwen-2.5-32B-Instruct that is pretrained using text listwise reranking data (Pradeep et al., 2023b). More details in Appendix A.

Baselines: Our primary comparison is against prior agent-based localization methods. Specifically, we include OpenHands (Wang et al., 2025), SWE-Agent (Yang et al., 2024b), MoatlessTools (Örwall, 2024) and LocAgent (Chen et al., 2025), the current SOTA agent-based approach. Notably, these methods predominantly use closed-source models, with LocAgent also finetuning open-source models for this task. For the retrieve-and-rerank framework, we compare SweRankEmbed-Small against BM25 (Robertson et al., 1994) and several code embedding models of comparable size, including Jina-Code-v2 (Günther et al., 2023), Codesage-large-v2 (Zhang et al., 2024), and CodeRankEmbed (Suresh et al., 2024). For the 7B parameter embedding model comparison, we include GTE-Qwen2-7B-Instruct, which ranks third on the MTEB leaderboard (Muennighoff et al., 2023) at the time of evaluation. For the reranker comparison, we include CodeRankLLM and other closed source-models such as GPT-4.1. Due to the larger size of LocBench, comparisons on this benchmark are limited to a subset of the best-performing baselines.

Method	Loc Model	File (%)		Module (%)		Function (%)	
		Acc@5	Acc@10	Acc@10	Acc@15	Acc@10	Acc@15
Agentless	`Claude-3.5`	67.50	67.50	53.39	53.39	42.68	42.68
OpenHands	`Claude-3.5`	79.82	80.00	68.93	69.11	59.11	59.29
SWE-agent	`Claude-3.5`	77.68	77.68	63.57	63.75	51.96	51.96
LocAgent	`Qwen2.5-7B(ft)`	78.57	79.64	63.04	63.04	51.43	51.79
	`Claude-3.5`	83.39	86.07	70.89	71.07	59.29	60.71
Retriever	CodeRankEmbed (137M)	74.29	80.36	63.93	67.86	47.86	50.89
	GTE-Qwen2-7B-Instruct (7B)	75.54	82.50	67.14	71.61	51.79	57.14
	SweRankEmbed-Small (137M)	80.36	84.82	71.43	75.00	58.57	63.39
	SweRankEmbed-Large (7B)	82.14	86.96	75.54	78.93	63.21	67.32
+ Reranker	CodeRankLLM (7B)	83.93	88.21	76.96	80.89	64.64	69.29
	GPT-4.1	85.89	88.75	79.64	82.50	71.61	74.64
	SweRankLLM-Small (7B)	85.54	88.39	79.11	82.14	69.46	74.46
	SweRankLLM-Large (32B)	**86.61**	**89.82**	**81.07**	**83.21**	**71.25**	**76.25**

Table 2: Performance (in %) on LocBench. The rerankers use SweRankEmbed-Large as the retriever. Gray correspond to results with closed-source models. Best retriever model numbers are in blue, while best overall numbers (except GPT-4.1) are in **bold**.

Datasets & Metrics: We evaluate on SWE-Bench-Lite (Jimenez et al., 2023) and LocBench (Chen et al., 2025). Following Suresh et al. (2024), we exclude examples from SWE-Bench-Lite where no existing functions were modified by the patch, resulting in 274 retained examples out of 300. While SWE-Bench-Lite primarily consists of bug reports and feature requests, LocBench (560 examples) also includes security and performance issues. Consistent with Chen et al. (2025), we measure localization performance at three granularities: file, module (class) and function, with Accuracy at k (Acc@k) as the evaluation metric. This metric deems localization successful if all relevant code locations are correctly identified within the top-k results. The relevance score for a specific file or module is determined by the maximum score of any function contained within that file or module.

5.2 Localization Results

Table 1 compares performance of different localization methods on the SWE-Bench-Lite benchmark. The results indicate that our SweRank models surpasses the performance of all evaluated agent-based methods. Furthermore, the SweRankEmbed-Small model, despite its relatively small size of 137M parameters, demonstrates highly competitive performance, outperforming prior 7B parameter embedding models. Notably, SweRankEmbed-Large achieves higher Acc@10 for function localization than LocAgent with Claude-3.5. Employing the SweRankLLM reranker subsequently enhances the retriever's output, establishing a new SOTA for localization performance on this benchmark across all granularities. Qualitative examples are provided in Appendix G.

Table 2 shows results on LocBench. A similar trend is observed, with the large variants of SweRankEmbed and SweRankLLM setting new SOTA performance. Figure 4 provides a detailed breakdown of localization accuracy across the four distinct difficulty categories within LocBench. Despite being primarily trained with bug reports in SweLoc, the SweRank models demonstrate impressive generalizability across other categories.

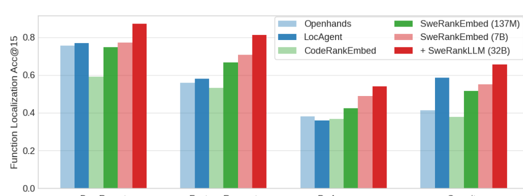
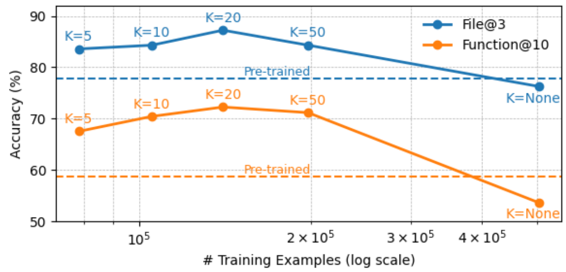


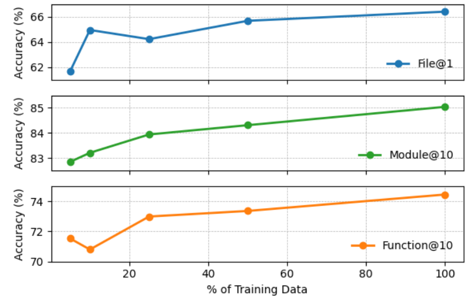
Figure 4: Localization performance across different categories within LocBench. SweRank considerably outperforms Agent-based methods using Claude-3.5.

5.3 Analysis

Our analysis presented in this section aims to demonstrate the following key points: 1) the impact of SweLoc data quality and size on final model performance (§5.3.1); 2) the utility of SweLoc for finetuning various retriever and reranker models (§5.3.2; and 3) the cost-effectiveness of the proposed SweRank framework (§5.3.3). Unless otherwise mentioned, the results are on SWE-Bench-Lite.



(a) While accuracy improves from doing consistency filtering, i.e. discarding instances where the positive's rank among negatives is $>K$, no filtering (K=*None*) hurts performance.



(b) All metrics show a general upward trend as the percentage of training data (K=20) increases.

Figure 6: Impact of (a) training data filtering and (b) data size on SWERANKEMBED-SMALL performance.

5.3.1 DATA QUALITY AND SIZE

Public GitHub repositories, as a source for contrastive data, often contain noisy instances. This study first examines the effectiveness of consistency filtering (§3.3), specifically the influence of the positive-rank threshold, K. This parameter dictates the minimum rank of the instance's positive (relative to negatives, based on similarity with the issue description) for inclusion of the instance in the training set. Increasing K relaxes the filtering, yielding more training instances but potentially introducing more noise. As shown in Figure 6a, finetuning SWERANKEMBED-SMALL with SWELOC data filtered by different K values reveals that optimal performance is achieved with a moderate K (e.g., K=20), striking a balance between instance quality and dataset size. The absence of filtering (K=None) proves detrimental as performance drops after finetuning compared to pre-trained model.

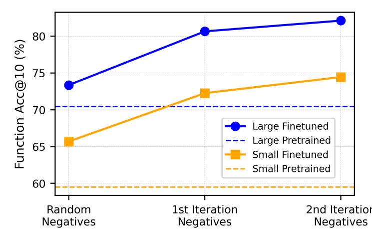


Figure 5: Plot showing SWERANKEMBED performance against increasingly hard negatives in SWELOC. Finetuned models notably improve from an additional iteration of negative mining.

Controlling for data quality (by fixing K=20), the impact of dataset size is investigated. Figure 6b illustrates that training with varying proportions of the filtered data yields considerable performance improvements, even with only 5% of the data. Generally, larger dataset sizes correspond to further performance gains. These experiments underscore the significance of both data quality and quantity, demonstrating that merely increasing data volume without quality control can be detrimental. Further, the impact of negative hardness on SWERANKEMBED performance is examined. Figure 5 shows localization accuracy for Large and Small variants (finetuned and pretrained) with increasingly hard negatives. In an iterative mining approach, 1st iteration negatives are mined using the small pretrained model, and 2nd iteration negatives use the small model from 1st iteration. Results indicate that finetuning with random negatives yields smaller gains, while using 2nd iteration negatives yields notably improves performance over the 1st iteration.

5.3.2 CHOICE OF RETRIEVER AND RERANKER

Here, we demonstrate the effectiveness of SWELOC by showing improvements for a variety of retriever and reranker models from finetuning. First, the following embedding models, pre-trained on different data types, are finetuned for one epoch on SWELOC: Arctic-Embed (Merrick et al., 2024), primarily pre-trained on English text retrieval data; CodeRankEmbed, pre-trained on 22 million NL-to-Code examples (Suresh et al., 2024); and Arctic-Embed-v2.0 (Yu et al., 2024), pre-trained on a mix of English and multilingual data. From Table 3, we see all models showing significant performance improvement from finetuning. Notably, models that initially performed weaker (e.g., Arctic-Embed) showed greater gains. This outcome validates that SWELOC can substantially improve the performance of *any* embedding model for software issue localization.

Base Retriever	**Pretrain**	**Func. Acc@10** (%)
CodeRankEmbed	English+Code	59.5→**72.3** (+12.8)
Arctic-Embed	English	53.7→71.9 (+17.4)
Arctic-Embed-v2.0	Multilingual	62.0→70.1 (+8.1)

Table 3: Accuracy (Before→After) from finetuning different retrievers with SWELOC data.

Next, text- and code-instruction-tuned LLMs of different sizes from the Qwen2.5 family (Yang et al., 2024c; Hui et al., 2024) are finetuned as listwise LLM rerankers using SWELOC data. Since we only apply loss on the first generation token, to ensure compatibility with the listwise output format, all models were initially pretrained on listwise text reranking data (Pradeep et al., 2023b), which provides the full ranking order to use for supervision. The results, shown in Table 4, indicate that rerankers across different model sizes universally benefit from finetuning on SWELOC. An interesting observation is that the code-pretrained model performs marginally better at the 7B scale, while the text-pretrained models achieve better results at the 3B and 32B scales. Results with finetuning Llama-3.1 are in Appendix B.

Base LLM Reranker	**Func. Acc@5** (%)	**Func. Acc@10** (%)
Qwen-2.5-Text (32B)	77.0→**81.4** (+4.4)	82.5 →**86.1** (+3.6)
Qwen-2.5-Code (32B)	76.3→79.9 (+3.6)	81.8 →84.7 (+2.9)
Qwen-2.5-Text (7B)	75.2→75.6 (+0.4)	81.4 →82.5 (+1.1)
Qwen-2.5-Code (7B)	75.5→75.9 (+0.4)	81.0 →83.6 (+2.6)
Qwen-2.5-Text (3B)	68.3→73.7 (+4.6)	76.6→82.5 (+5.9)
Qwen-2.5-Code (3B)	71.2→71.9 (+0.7)	80.3→81.0 (+0.7)

Table 4: Localization accuracy (Before→After) from finetuning different listwise rerankers with SWELOC.

5.3.3 INFERENCE COST ANALYSIS

Agent-based localization approaches typically involve multiple iterations, each requiring extensive chain-of-thought generation (Wang et al., 2023b), incurring considerable cost at inference. In contrast, SWERANK offers significant cost-effectiveness as the SWERANKLLM reranker only needs to generate output candidate identifiers to determine the ranking order. Furthermore, the SWERANKEMBED output embeddings can be pre-computed, resulting in negligible extra cost. Table 5 compares the inference costs of SWERANKLLM with other agent-based methods. Clearly, agent-based approaches, often relying on closed-source models for better performance, are highly cost-intensive. SWERANK is substantially cheaper while providing significantly better performance, with up to *6X* better performance-cost tradeoffs compared to LocAgent.

Method	**Model**	**Cost($)** ↓	**Acc@10 / Cost** ↑
SWE-agent	`GPT-4o`	0.46	0.8
	`Claude-3.5`	0.67	1.0
Openhands	`GPT-4o`	0.83	0.6
	`Claude-3.5`	0.79	0.9
LocAgent	`Claude-3.5`	0.66	1.2
	`Qwen2.5-7B(ft)`	0.05	13.2
	`Qwen2.5-32B(ft)`	0.09	8.6
Reranker	`GPT-4.1`	0.16	5.9
	SWERANKLLM (7B)	0.011	**79.0**
	SWERANKLLM (32B)	0.015	57.5

Table 5: SWERANKLLM has considerably better inference cost-efficiency than agent-based methods while being more performant.

5.3.4 IMPACT ON DOWNSTREAM ISSUE RESOLUTION

This section analyzes the impact of improved localization on downstream code repair performance. To evaluate issue resolution, we utilize SWE-Fixer (Xie et al., 2025), a two-step pipeline consisting of code file retrieval (localization) followed by code editing. We compare the repair outcomes on SWE-Bench-Lite when employing different localization methods: the native localization mechanism of SWE-Fixer, LocAgent (with Claude-3.5), our SWERANK (large variant), and an oracle. The oracle simulates perfect localization by using the ground-truth edited file, thereby providing an upper bound for the repair framework. From Table 6, we see that better localization provided by SWERANK yields improved issue resolution, with oracle results showing that repair performance is currently constrained by the code editing model.

Localization	**File Acc@1**	**Repair Pass@1**
SWE-Fixer	69.7	21.0
LocAgent	78.5	22.6
SWERank	**83.2**	**24.5**
Oracle	100	25.9

Table 6: Impact of localization accuracy on downstream issue resolution.

5.3.5 PERFORMANCE ANALYSIS BY ISSUE COMPLEXITY

Aggregate metrics often obscure performance variance on complex issues. To address this, we stratify the test set by `num_gold` (the number of functions modified in the ground truth patch) as a proxy for issue complexity. We compare our approach against LocAgent and Gemini-Embedding, with results summarized in Figure 7.

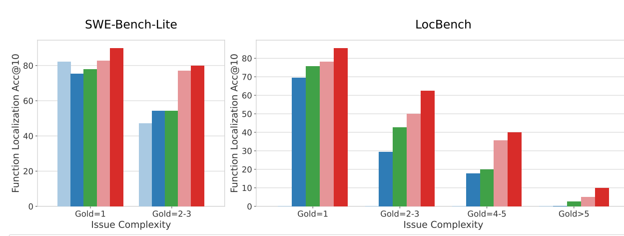


Figure 7: Function Acc@10 breakdown by issue complexity.

As expected, performance degrades as the number of modified functions increases. Instances requiring changes to > 5 functions are extremely difficult for all models, resulting in single-digit accuracy. However, SWERANKEMBED-LARGE demonstrates significantly better scaling on complex issues (`num_gold=2–3`) compared to Agentic approaches; on SWE-Bench-Lite, LocAgent drops from 82.0% to 47.1%, while our retriever maintains 77.1%. Furthermore, the reranker consistently improves performance across all complexity levels, confirming that it successfully captures cross-function dependencies that the bi-encoder might miss.

5.3.6 PERFORMANCE ANALYSIS BY LEXICAL AND SEMANTIC OVERLAP

To further dissect the model's capabilities, we analyzed performance by grouping instances based on **Lexical Overlap** (Rouge-1) and **Semantic Overlap** (Cosine Similarity).

Lexical Overlap. We bucket instances into four groups using the Rouge-1 score between the issue description and the ground-truth localized functions. A high Rouge score indicates significant keyword overlap. Figure 8 summarizes the results. We observe that performance generally degrades as lexical overlap decreases. However, even in the lowest overlap bucket (0.0–0.1), SWERANKEMBED-LARGE outperforms LocAgent (65.2% vs 60.9% on SWE-Bench-Lite), demonstrating that our model does not rely solely on keyword matching. Furthermore, SWERANKLLM consistently improves performance, with significant gains seen specifically for instances with low lexical overlap.

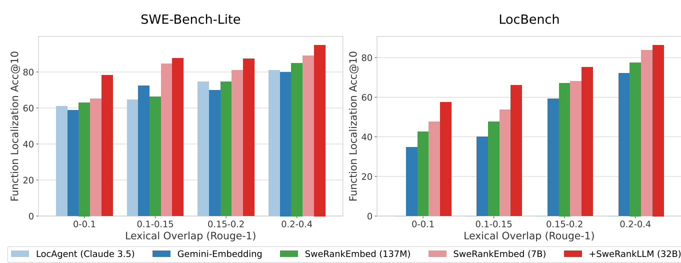


Figure 8: Performance breakdown by **Lexical Overlap** (Rouge-1).

Semantic Overlap. We also categorize instances based on the mean cosine similarity (computed via Gemini-Embedding) of the issue description and the ground-truth functions. As shown in Figure 9, performance is directly correlated with semantic overlap, achieving near-perfect accuracy for high-similarity instances (> 0.8). Notably, SWERANKLLM reranker considerably boosts performance over the retriever in low-similarity buckets (0.65–0.75), outperforming the multi-turn LocAgent approach. This suggests that while agentic tool-use can bridge the semantic gap, our training on SWELOC—which incorporates hard negatives—enables the SWERANK framework to learn these non-obvious mappings effectively without the cost of agentic inference.

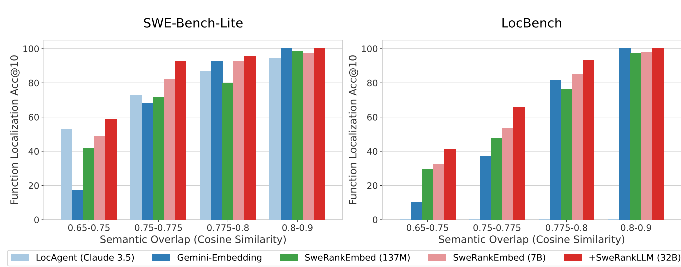


Figure 9: Performance breakdown by **Semantic Overlap**.

6 CONCLUSION

This paper frames software issue localization as a specialized ranking task and introduces SWERANK, a highly performant and cost-effective retrieve-and-rerank framework. To effectively train SWERANK models, we construct SWELOC, a large-scale contrastive training dataset derived from real-world GitHub issues, employing consistency filtering and hard-negative mining for quality. Empirical evaluations on SWE-Bench-Lite and LocBench demonstrate state-of-the-art localization performance using SWERANK, significantly outperforming costly closed-source agent-based systems. The introduction of SWELOC dataset provides a valuable resource for advancing research in this domain.

Reproducibility Statement

We plan to release the dataset publicly for the benefit of the community. The supplementary material attached provides scripts for model training, in addition to the dataset construction process. More details about model training necessary for reproducing experiments are provided in Appendix A.

References

A Adhiselvam, E Kirubakaran, and R Sukumar. An enhanced approach for software bug localization using map reduce technique based apriori (mrtba) algorithm. *Indian Journal of Science and Technology*, 8:35, 2015. 2

Higor Amario de Souza, Marcelo de Souza Lauretto, Fabio Kon, and Marcos Lordello Chaim. Understanding the use of spectrum-based fault localization. *Journal of Software: Evolution and Process*, 36(6):e2622, 2024. 2

Anthropic. Claude: Conversational ai by anthropic, 2023. URL `https://www.anthropic.com/claude`. Accessed: January 21, 2025. 2

Md Mustakim Billah, Palash Ranjan Roy, Zadia Codabux, and Banani Roy. Are large language models a threat to programming platforms? an exploratory study. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 292–301, 2024. 3

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided llm agents for code localization. *arXiv preprint arXiv:2503.09089*, 2025. 1, 2, 3, 4, 6, 7

Cognition AI. Devin: The First AI Software Engineer. `https://devin.ai/`, 2024. Accessed: 2025-04-22. 1

Cursor. Cursor: The AI Code Editor. `https://www.cursor.com/`, 2025. Accessed: 2025-04-22. 1

Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016. 2

E. Elsaka. Chapter three - fault localization using hybrid static/dynamic analysis. volume 105 of *Advances in Computers*, pp. 79–114. Elsevier, 2017. doi: https://doi.org/10.1016/bs.adcom.2016.12.004. URL `https://www.sciencedirect.com/science/article/pii/S0065245816300778`. 2

Paul Gauthier. How aider scored sota 26.3% on swe bench lite — aider, 2024. URL `https://aider.chat/2024/05/22/swe-bench-lite.html`. Accessed: January 21, 2025. 1

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 16

Michael Günther, Georgios Mastrapas, Bo Wang, Han Xiao, and Jonathan Geuter. Jina embeddings: A novel set of high-performance sentence embedding models. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pp. 8–18, 2023. 4, 6

Michael Günther, Louis Milliken, Jonathan Geuter, Georgios Mastrapas, Bo Wang, and Han Xiao. Jina embeddings: A novel set of high-performance sentence embedding models, 2023. URL `https://arxiv.org/abs/2307.11224`. 3, 6

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186, 2024. URL `https://api.semanticscholar.org/CorpusID:272707390`. 9

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. 2, 3

Neel Jain, Ping yeh Chiang, Yuxin Wen, John Kirchenbauer, Hong-Min Chu, Gowthami Somepalli, Brian R. Bartoldson, Bhavya Kailkhura, Avi Schwarzschild, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Neftune: Noisy embeddings improve instruction finetuning, 2023. URL https://arxiv.org/abs/2310.05914. 16

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023. 1, 2, 4, 7

Hyoungwook Jin, Seonghee Lee, Hyungyu Shin, and Juho Kim. Teach ai how to code: Using large language models as teachable agents for programming education. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–28, 2024. 3

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*, 2022. 3

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. Matryoshka representation learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 30233–30249. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf. 17

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019. 16

Jinhyuk Lee, Feiyang Chen, Sahil Dua, Daniel Cer, Madhuri Shanbhogue, Iftekhar Naim, Gustavo Hernández Ábrego, Zhe Li, Kaifeng Chen, Henrique Schechter Vera, et al. Gemini embedding: Generalizable embeddings from gemini. *arXiv preprint arXiv:2503.07891*, 2025. 16, 18

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Yichun Yin, Hao Zhang, Yong Liu, Yasheng Wang, and Ruiming Tang. Csn: A comprehensive benchmark for code information retrieval models. *arXiv preprint arXiv:2407.02883*, 2024a. 2, 3

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Yichun Yin, Hao Zhang, Yong Liu, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. *arXiv preprint arXiv:2407.02883*, 2024b. 2, 3

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. 6, 16

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024a. 3

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024b. 3

Rui Meng, Ye Liu, Shafiq Rayhan Jotya, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Sfr-embedding-2: Advanced text embedding with multi-stage training, 2024. URL https://huggingface.co/Salesforce/SFR-Embedding-2_R. 6

Luke Merrick, Danmei Xu, Gaurav Nuti, and Daniel Campos. Arctic-embed: Scalable, efficient, and accurate text embedding models. *arXiv preprint arXiv:2405.05374*, 2024. 8, 16

Microsoft. GitHub Copilot—Your AI pair programmer, 2023. URL https://github.com/features/copilot. 1

Gabriel de Souza P Moreira, Radek Osmulski, Mengyao Xu, Ronay Ak, Benedikt Schifferer, and Even Oldridge. Nv-retriever: Improving text embedding models with effective hard-negative mining. *arXiv preprint arXiv:2407.15831*, 2024. 4

Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 2014–2037, 2023. 6

Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic embed: Training a reproducible long context text embedder, 2024. 16

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. In *Advances in Neural Information Processing Systems*, pp. 10203–10213, 2018. 5

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024. 3

Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. Rankvicuna: Zero-shot listwise document reranking with open-source large language models, 2023a. URL `https://arxiv.org/abs/2309.15088`. 16

Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. Rankzephyr: Effective and robust zero-shot listwise reranking is a breeze! *arXiv preprint arXiv:2312.02724*, 2023b. 5, 6, 9, 16

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*, 2024. 2

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020. 16

Revanth Gangi Reddy, JaeHyeok Doo, Yifei Xu, Md Arafat Sultan, Deevya Swain, Avirup Sil, and Heng Ji. First: Faster improved listwise reranking with single token decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8642–8652, 2024. 2

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL `https://aclanthology.org/D19-1410`. 5

Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3. In *Text Retrieval Conference*, 1994. URL `https://api.semanticscholar.org/CorpusID:41563977`. 6

Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. Code summarization: Do transformers really understand code? In *Deep Learning for Code Workshop*, 2022. 3

Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26:1–45, 2021. 2

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code ranking. *arXiv preprint arXiv:2412.01007*, 2024. 1, 2, 3, 4, 6, 7, 8, 16

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. FEVER: a large-scale dataset for fact extraction and VERification. In *NAACL-HLT*, 2018. 16

Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pp. 146–158, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616256. URL `https://doi.org/10.1145/3611643.3616256`. 2

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 *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=OJd3ayDDoF`. 2, 6

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023b. URL `https://arxiv.org/abs/2203.11171`. 9

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023c. URL `https://arxiv.org/abs/2305.07922`. 3

Windsurf. Windsurf Editor: The AI-Native IDE. `https://windsurf.com/editor`, 2025. Accessed: 2025-04-22. 1

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. 2

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025. 9

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024a. 1

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024b. 1, 2, 6

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL `https://arxiv.org/abs/2504.21798`. 18

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024c. URL `https://api.semanticscholar.org/CorpusID:274859421`. 9

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2018. 16

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. 1

Puxuan Yu, Luke Merrick, Gaurav Nuti, and Daniel Campos. Arctic-embed 2.0: Multilingual retrieval without compromise. *arXiv preprint arXiv:2412.04506*, 2024. 8

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350*, 2025. 1

Wang Yue, Wang Weishi, Shafiq Joty, and Steven C.H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021. 1

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=vfzRRjumpX`. 1, 3, 6

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL `https://arxiv.org/abs/2506.05176`. 17

Albert Örwall. Moatless tools, 2024. URL `https://github.com/aorwall/moatless-tools`. 3, 6

A Training Details

A.1 SweRankEmbed

Our data filtering, negative mining, and model finetuning are implemented using the contrastors package (Nussbaum et al., 2024). The SweRankEmbed-Small encoder uses CodeRankEmbed, which was initialized with Arctic-Embed-M (Merrick et al., 2024), a text encoder supporting an extended context length of 8,192 tokens and pretrained on large-scale web query- document pairs, along with public text retrieval datasets (Yang et al., 2018; Kwiatkowski et al., 2019; Thorne et al., 2018). The encoder supports a query prefix "*Represent this query for searching relevant code:* ", as set by (Suresh et al., 2024). The model is finetuned using 8 GH200 GPUs for two epochs with a learning rate of 2e-5, a batch size of 64 and 15 hard negatives per example.

The SweRankEmbed-Large encoder uses GTE-Qwen2-7B-Instruct (Li et al., 2023), which was pretrained on a large corpora of text retrieval data. For this model, we use a custom query prefix "*Instruct: Given a github issue, identify the code that needs to be changed to fix the issue. Query:* ". The model is finetuned using 8 GH200 GPUs for 1 epoch with a learning rate of 8e-6, a batch size of 64 and 7 hard negatives per example.

A.2 SweRankLLM

Training data: For each <query, positive, negatives> tuple from SweLoc, we randomly sample 9 negative codes to fit the listwise reranking window size of 10 along with the positive code. To prevent the positional bias from affecting the reranker and ensure model robustness (Pradeep et al., 2023a), we shuffle the order of candidate codes for each training example. Since the combined length of a GitHub issue and corresponding candidate codes may exceed the model's maximum embedding size, we set the maximum length per candidate code to 1024 and the total length limit to 16348. For overlong prompts, we truncate the query to reach the maximum total length. This preserves meaningful context for issue localization as much as possible within the limited context window size for effective model training. The rerankers are all first pretrained with text listwise reranking data (Pradeep et al., 2023b) to teach the model to follow the listwise output format.

Hyperparameters: For the LLM reranker training, with both text reranking and SweLoc data, we trained for one epoch with a global batch size of 128, an initial learning rate of 5e-6 with 50 warmup steps, cosine learning rate scheduler, bfloat16 precision, and noisy embeddings (Jain et al., 2023) with a noise scale $\alpha = 5$. For efficient long-context, multi-gpu training, we used DeepSpeed (Rasley et al., 2020) ZeRO stage 3 with 16 GH200 GPUs.

B Experiments with More Reranker Models

To demonstrate the broader applicability of our dataset, we conduct experiments with finetuning Llama-3.1 8B Instruct (Grattafiori et al., 2024) as a listwise reranker. The models are first pre-trained on general text reranking data from RankZephyr (Pradeep et al., 2023b) and subsequently finetuned on our SweLoc dataset. Results, shown in Table 7, demonstrate significant performance gains on both SWE-Bench-Lite and LocBench after fine-tuning on SWE-Loc. This confirms that our dataset is a valuable resource for improving the issue localization capabilities of various LLM families, not just Qwen 2.5.

Method Type	SWE-Bench-Lite		LocBench	
	Acc@5	Acc@10	Acc@5	Acc@10
Zeroshot Reranker	60.22	81.39	61.96	69.11
RankZephyr finetune	72.99	80.29	64.11	70.00
+ SweLoc finetune	**77.01**	**85.77**	**68.04**	**73.04**

Table 7: Function localization accuracy of Llama-3.1 8B Instruct as a listwise LLM reranker.

C Retriever Ceiling Analysis

To assess the upper bound (performance ceiling) provided by the retrieval stage, we report extended metrics (Acc@20, @50, and @100) in Table 8. We compare our models against Gemini-Embedding (Lee et al., 2025). On SWE-Bench-Lite, SweRankEmbed-Large achieves a retrieval

Model	SWE-Bench-Lite (Acc@K)				LocBench (Acc@K)			
	Acc@10	**Acc@20**	**Acc@50**	**Acc@100**	**Acc@10**	**Acc@20**	**Acc@50**	**Acc@100**
SweRankEmbed-Small	74.45	81.75	87.96	91.97	58.57	67.50	75.71	82.32
SweRankEmbed-Large	**82.12**	**86.50**	**90.88**	**93.43**	**63.21**	**71.25**	**80.71**	**84.29**
Gemini-Embedding	72.26	79.20	87.96	90.88	51.43	60.18	70.00	78.39

Table 8: Extended retrieval metrics. Acc@K indicates the percentage of instances where all ground-truth functions are within the top-K retrieved candidates. The Acc@100 score indicates a performance ceiling for the subsequent reranker.

Model	Depth	Width	Dim	Acc@5	Acc@10
SweRankEmbed-Small (137M)	12	768	768	51.82 → 63.14	58.76 → 74.45
Qwen3-Embedding-0.6B	28	2048	1024	52.55 → 66.79	62.77 → 75.18
SweRankEmbed-Large (7B)	28	3072	3584	63.14 → 71.90	70.44 → 82.12
Qwen3-Embedding-8B	36	3072	4096	60.95 → **73.72**	71.53 → **83.94**

Table 9: Impact of retriever model capacity. We compare different variants of the SWERANKEMBED and Qwen3-Embedding models. Performance is reported as (Before → After) finetuning on SWELOC.

ceiling (Acc@100) of 93.43%. Given that SWERANKLLM-LARGE achieves 88.7% Acc@10, the gap suggests that the retrieval stage is not the primary bottleneck. Furthermore, this retrieval ceiling is significantly higher than the best performance achieved by agentic methods like LocAgent (∼78%).

D ABLATION STUDIES ON MODEL CAPACITY

While the experimental results in the main text demonstrates that performance generally improves with model size for rerankers, we provide additional experiments here to analyze the sensitivity of the retriever to model capacity and architecture design.

Impact of Encoder Depth & Width: To isolate the effects of model architecture, we compare our SWERANKEMBED variants against the recently released Qwen3-Embedding models (Zhang et al., 2025) (0.6B and 8B variants). We finetuned the Qwen3 models on the SWELOC dataset using the exact same procedure as SWERANKEMBED. Table 9 details the model specifications and performance. Comparing SWERANKEMBED-SMALL to Qwen3-0.6B, we observe moderate gains from utilizing a significantly deeper and wider encoder. However, comparing SWERANKEMBED-LARGE to Qwen3-8B suggests diminishing returns from further increasing depth (28 vs. 36 layers). Conversely, the considerable performance gap between Qwen3-0.6B and SWERANKEMBED-LARGE appears driven by the larger embedding dimension, which we investigate below.

Impact of Embedding Dimension: To strictly isolate the impact of embedding dimension, we performed a controlled ablation using the Qwen3-Embedding-0.6B model, which supports flexible vector dimensions via Matryoshka Representation Learning (MRL) (Kusupati et al., 2022). Table 10 presents the results on SWE-Bench-Lite after finetuning with MRL.

Dimension (D)	**Acc@5** (Before → After)	**Acc@10** (Before → After)
1024	52.55 → **66.79**	62.77 → **75.18**
512	50.00 → 65.69	59.85 → 74.09
256	44.16 → 59.12	52.92 → 69.71
128	39.05 → 56.93	46.72 → 66.79
64	33.21 → 50.00	38.32 → 60.22

Table 10: Controlled ablation on embedding dimension using Qwen3-Embedding-0.6B with Matryoshka Representation Learning. SWELOC provides larger relative gains at lower dimensions.

Performance drops significantly as embedding size decreases, identifying dimension as a critical factor. Interestingly, finetuning on SWELOC yields larger relative gains at lower dimensions (e.g., +21.9 points Acc@10 for $D = 64$ vs. +12.5 points for $D = 1024$), highlighting the dataset's utility even for compressed representations.

E MULTILINGUAL GENERALIZATION

Although SWELOC is constructed primarily from Python repositories, we hypothesize that SWERANK generalizes effectively to other languages because the underlying base models (CODERANKEMBED and GTE-QWEN2) are pretrained on massive multilingual corpora.

To empirically validate this, we evaluate the models on SWE-Bench Multilingual (Yang et al., 2025), which includes 234 tasks across 9 languages (C, C++, Java, JavaScript, TypeScript, Rust, PHP, Ruby, and Go). We compare the performance against Gemini-Embedding (Lee et al., 2025), a state-of-the-art proprietary general-purpose retriever.

Method	Model	Acc@5	Acc@10
Retriever	CodeRankEmbed (137M)	26.50	35.04
	SweRankEmbed-Small (137M)	33.33	44.02
	GTE-Qwen2-7B-Instruct (7B)	34.19	42.31
	SweRankEmbed-Large (7B)	**39.74**	**50.85**
	Gemini-Embedding	36.75	47.44
Reranker	SweRankEmbed-Small (Base)	33.33	44.02
	+ CodeRankLLM	42.74	51.28
	+ SweRankLLM-Small	**49.15**	**56.84**

Table 11: Function Localization Performance on SWE-Bench Multilingual. SWERANK generalizes effectively to non-Python languages.

The results, summarized in Table 11, support our hypothesis. SWERANKEMBED-LARGE (50.85% Acc@10) outperforms the proprietary Gemini-Embedding (47.44%), despite being finetuned on Python data. Furthermore, finetuning on SWELOC provided significant gains over the base models for both the retriever and the reranker. This demonstrates that the "issue-to-code" relevance signal learned from SWELOC is not language-specific and transfers effectively across different languages.

F EFFICIENCY COMPARISON WITH AGENTIC BASELINE

F.1 INFERENCE LATENCY ANALYSIS

To complement the cost analysis, we evaluate the inference latency of our approach compared to LocAgent. The average latency is measured over 50 instances on SWE-Bench-Lite. For the SWERANK framework, the retrieval embeddings can be pre-computed and indexed, making the online retrieval cost negligible; therefore, the primary latency bottleneck stems purely from the reranking step.

Approach	Model	Latency (s)
SweRank	SweRankLLM (7B)	**12.5**
SweRank	GPT-4o	30.2
LocAgent	GPT-4o	85.3

Table 12: Inference Latency comparison.

Table 12 summarizes the results. When deploying the SWERANKLLM-SMALL model locally on a single 80GB A100 GPU, the system achieves an average latency of just 12.5 seconds per instance. This is approximately **7× faster** than the LocAgent baseline, making SWERANK far more viable for real-time developer assistance scenarios where rapid feedback is critical. Even when controlling for the underlying model by using GPT-4o for both approaches, SWERANK remains nearly **3× faster**. This efficiency gain primarily comes from SWERANK resolving the issue in a single-turn ranking pass, whereas agentic baselines like LocAgent relying on multi-turn loops, involving iterative thought generation, tool execution, and context reading, which naturally accumulates significant latency.

F.2 Token Efficiency and Footprint

Beyond latency, the computational load of a system is heavily influenced by its token usage. Agent-based approaches often suffer from "context bloat," as they must maintain a running history of all past thoughts, observations, and tool outputs throughout the interaction loop. We analyzed the average token footprint (Average Input Prompt & Output Tokens) required for each github issue.

Approach	Prompt Tokens	Output Tokens
SweRank	78,409	741
LocAgent	234,197	1,884

Table 13: Token footprint comparison.

As shown in Table 13, SweRank operates with a $\sim$**3**$\times$ **lower** input token footprint compared to the agentic baseline. By formulating localization as a ranking problem rather than a sequential decision-making process, SweRank eliminates the need for extensive history management. Furthermore, the reduction in output tokens is even more pronounced. Since output tokens are significantly more expensive and slower to generate than input tokens, this reduction directly translates to the lower latency observed in §F.1 and substantially reduced inference costs. This confirms that SweRank provides a more sustainable and scalable alternative to agentic loops for issue localization.

G Qualitative Examples

Figure 10 presents qualitative examples from SWE-Bench-Lite where SweRank correctly localizes the function to edit while LocAgent is unable to. In both instances, LocAgent incorrectly identifies functions that likely correspond to where the problem manifests rather than where it originates.

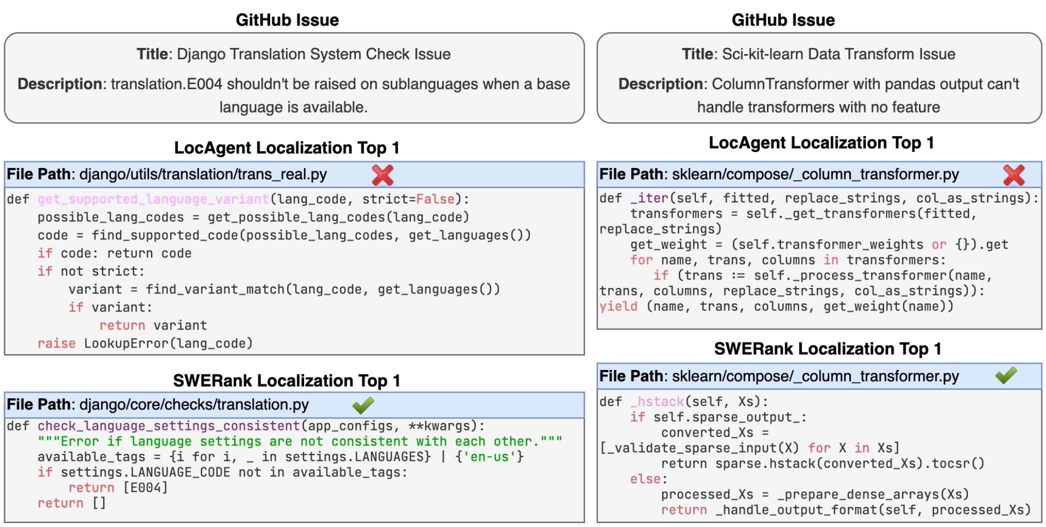


Figure 10: Examples from SWE-Bench-Lite where LocAgent mislocalizes the function, while our SWERank framework does function localization correctly

H Diversity of Issue Topics in SweLoc

To provide more insight into the variety and complexity of issue topics in SweLoc, we analyze the distribution of topics for 10k randomly sampled instances. We use Nomic Atlas[2], a popular unstructured text visualization tool, that employs a cluster-based keyword identification algorithm and leverages a language model to generate topics. Figure 11 shows the frequency of top-15 topics.

[2] `https://atlas.nomic.ai/`

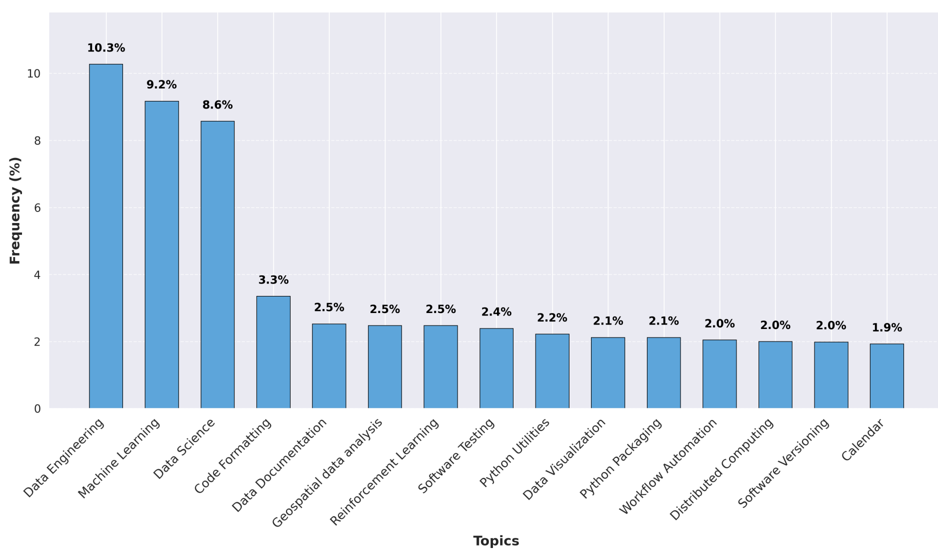


Figure 11: Top-15 issue topics and their frequencies from a randomly sampled subset of SweLoc.