SpIDER: Spatially Informed Dense Embedding Retrieval for Software Issue Localization

Anonymous ACL submission

Abstract

Retrieving code functions, classes or files relevant to a user query, bug report or feature request from large codebases is a fundamental challenge for Large Language Model (LLM)-based coding agents. Agentic approaches typically employ sparse methods like BM25 or dense embedding strategies to identify semantically relevant units. While dense embedding approaches can outperform BM25 by large margins, both ignore the graph-structured characteristics of the codebase. To address this, we propose SpIDER (Spatially Informed Dense Embedding Retrieval), a dense retrieval approach that integrates LLM-based reasoning with graph-based exploration of the codebase. We further introduce SpIDER-Bench, a graph-structured benchmark curated from SWE-PolyBench, SWEBench-Verified and Multi-SWEBench, spanning Python, Java, JavaScript and TypeScript repositories. SpIDER's graph-based candidate expansion attaches a structural reason for inclusion to each surfaced function (its seed and the edge type linking them), making the candidate set auditable while keeping the retrieval budget fixed. The graph is built from per-repository syntax trees, so it can be constructed on-demand at the start of a developer session rather than precomputed offline. Empirical results show that SpIDER consistently improves dense retrieval Recall@20 by at least 13% relative across programming languages and benchmarks in SpIDER-Bench[1].

1 Introduction

Recent work has demonstrated the potential of agents powered by Large Language Models (LLMs) for automated bug repair and feature implementation within large-scale software repositories (Yang et al., 2024; Gauthier, 2024; Xia et al., 2024). A critical prerequisite for effective code generation is the precise identification of relevant code units such as functions, classes, or files. This code localization task is a foundational yet difficult challenge, particularly at the fine-grained function level (Jimenez et al., 2024; Rashid et al., 2025; Zan et al., 2025; Chen et al., 2025). Poor retrieval directly compromises generated code, leading to incorrect or incomplete fixes and increased computational cost. It is therefore essential not only for autonomous patch generation but also for *developer oversight*: when retrieval choices are unexplained, reviewers cannot verify what the agent considered, eroding trust.

Current approaches to code localization fall into two primary categories. Some methods exploit graph representations of code repositories alongside the reasoning capabilities of LLMs (Ouyang et al., 2025; Chen et al., 2025; Jiang et al., 2025), mostly using sparse retrieval such as BM25 (Robertson et al., 1994) to identify relevant graph nodes or subgraphs for agent exploration. Others train bimodal encoders with a contrastive objective to align dense embeddings of code units with issue descriptions (Fehr et al., 2025; Reddy et al., 2025), ranking units by embedding similarity.

These existing approaches face three critical limitations. First, contrastive learning-based retrieval methods overlook the structural context of code modules within a repository when ranking them (Reddy et al., 2025; Zhang et al., 2024; Suresh et al., 2025). We observe that buggy code modules are often spatially proximate within the codebase (e.g., in the same file or in neighboring functions), and top-ranked entities retrieved via dense semantic similarity frequently reside *near* the actual buggy modules; relying exclusively on semantic similarity may thus overlook relevant candidates with *marginally* lower embedding scores. Second, existing methods have been primarily evaluated on Python (Chen et al., 2025; Fehr et al., 2025), partic-

[1] Given the size of SpIDER-Bench graphs, we provide two samples in the supplementary materials; the complete dataset will be released upon acceptance.

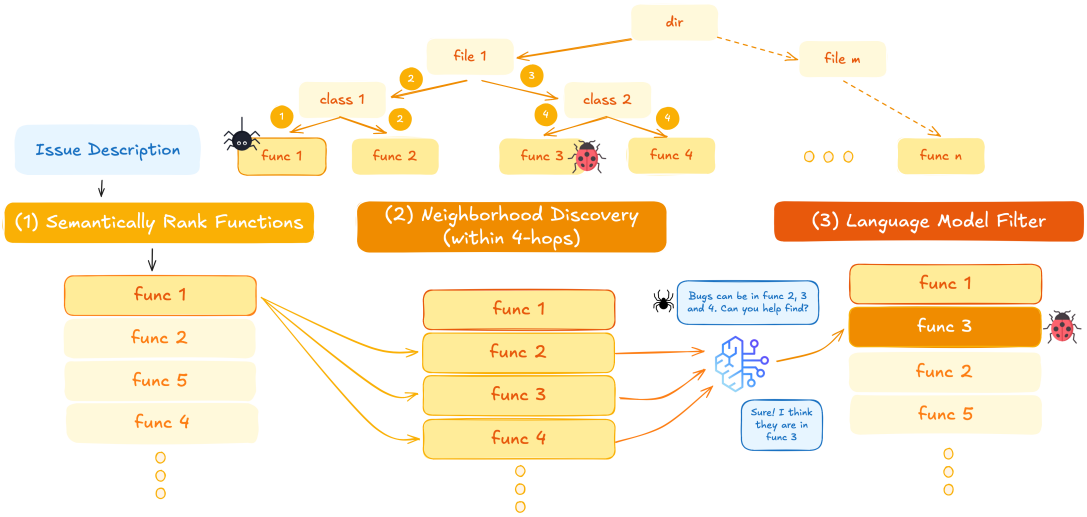

Figure 1: **SpIDER workflow.** SpIDER augments standard dense embedding retrieval with graph-aware spatial exploration and LLM-based reasoning to improve function-level code localization under a fixed retrieval (K). Given an issue description, all functions are first ranked by semantic similarity, and the top-K candidates are retrieved. The top-C (C<K) functions serve as centers (func 1) for structured neighborhood exploration along contains edges in the code graph, where neighboring functions within d hops are considered if they also rank within the top-N semantically (funcs 2,3,4). An LLM then filters these candidates to identify functions that are likely relevant but under-ranked by semantic similarity alone (func3). The selected neighbors are inserted immediately below their corresponding centers while discarding an equal number of bottom ranked functions in the initial top-K list, thereby increasing coverage of structurally related buggy functions without increasing K. Further algorithmic details are provided in Appendix A.9 (Algorithm 1).

ularly SWEBench-Lite, SWEBench-Verified, and LocBench (Jimenez et al., 2024; Chowdhury et al., 2024b; Chen et al., 2025), leaving their performance on other languages largely underexplored. Third, function-level retrieval, the most challenging granularity compared to class-level and file-level retrieval (Rashid et al., 2025; Reddy et al., 2025; Chen et al., 2025), remains largely understudied.

These limitations matter most in interactive, developer-in-the-loop settings, where retrieval must run cheaply on a developer's active repository at session start and expose its choices to the reviewer. Function-level granularity matches how developers reason about edits, and SpIDER's graph-based candidate expansion attaches a structural reason for inclusion to each surfaced function (its position relative to a top-ranked seed) rather than only an embedding score. The code graph is built on-demand from per-repository syntax trees, avoiding the offline corpus-wide preprocessing of many graph-based pipelines.

To address these limitations, we introduce SpIDER, a simple graph-aware dense retrieval strategy for *function-level* code localization. SpIDER incorporates spatial locality as a complementary signal to semantic similarity, jointly exploiting code content and repository structure within a fixed retrieval budget. Our focus is on retrieval methods that augment downstream tasks such as precise localization and automated patch generation for GitHub-style issues. For evaluation across multiple programming languages, we also introduce SpIDER-Bench, covering Python, Java, JavaScript, and TypeScript repositories from SWEBench-Verified, SWE-PolyBench, and Multi-SWEBench.

Our primary contributions are: (1) We propose SpIDER, a simple graph-aware dense retrieval strategy combining semantic content and code structure to identify the most relevant functions under a fixed retrieval budget. (2) We introduce SpIDER-Bench, a heterogeneous graph-structured benchmark for multi-language code localization, built from GitHub repositories in SWEBench-Verified, SWE-PolyBench, and Multi-SWEBench covering Python, Java, JavaScript, and TypeScript, with source code as node-level features. We validate SpIDER against existing dense retrievers and the sparse BM25 baseline across all four languages.

2 Related Work

Code Retrieval and Software Issue Localization Code retrieval refers to identifying code locations responsible for a software issue. Classical approaches are rooted in information retrieval, leveraging lexical or semantic similarity to rank candidate code snippets. Consequently, many existing systems rely on sparse retrievers such as BM25 (Robertson et al., 1994). While BM25 indexing is cheaper than dense embedding generation and vector similarity search, it typically underperforms when rich semantic representations are available (Reddy et al., 2025; Fehr et al., 2025).

Recent dense retrieval approaches propose bimodal encoder models to encode both code chunks as well as issue embeddings in an aligned latent space (Zhang et al., 2024; Suresh et al., 2025; Feng et al., 2020; Guo et al., 2021). However, these models are typically pretrained on generic natural language–to–code objectives, which leads to suboptimal performance for issue-driven code retrieval. To address this gap, Fehr et al. (2025) and Reddy et al. (2025) introduce bimodal encoders fine-tuned on GitHub-style issue resolution datasets such as `SWEBench` (Jimenez et al., 2024), resulting in improved alignment between issue and code representations. Despite these gains, `SWEBench` is limited to Python repositories, and no comparably large multilingual dataset currently exists to support training such encoders across diverse programming languages. Consequently, as real-world codebases span multiple languages, these fine-tuned dense retrievers exhibit degraded performance under domain shifts, reducing their robustness and reliability. *We argue that combining LLM-based reasoning with structural signals from code graphs complements semantic similarity and reduces sensitivity to surface-level language artifacts*. Augmenting dense retrievers with these components helps mitigate spurious correlations and improves generalization. We empirically validate this hypothesis on multilingual benchmarks, including `SWE-PolyBench` (Rashid et al., 2025), `Multi-SWEBench` (Zan et al., 2025), and `SWEBench-Verified` (Chowdhury et al., 2024a).

LLM-based retrieval methods Recent work has shown that large language models (LLMs) are highly effective at localizing and resolving GitHub-style software issues, largely due to their strong reasoning capabilities (Kang et al., 2024; Xia et al., 2024; Luo et al., 2024; Örwall, 2024). In the context of code retrieval, Xia et al. (2024) introduce a simple hierarchical localization strategy driven directly by an LLM, whereas Ouyang et al. (2025) employ an intermediate sparse BM25 retrieval stage to identify relevant files prior to code generation. Similarly, Chen et al. (2025) leverage inverted BM25 indexing to match issue-description keywords with node identifiers or code chunks. Both Ouyang et al. (2025) and Chen et al. (2025) further enable LLM-guided traversal of local subgraphs around the retrieved files or nodes, using edges defined by their respective code graph representations. In contrast, Reddy et al. (2025) focus on re-ranking densely retrieved code chunks using LLMs, with the goal of enhancing semantic retrieval through reasoning. Although this approach improves retrieval performance, it does not exploit the structural signals encoded in code graphs. *We aim to bridge this gap by jointly leveraging LLM reasoning, semantic embeddings, and the additional contextual information captured by code graphs, thereby improving overall retrieval effectiveness.*

Code Graph Construction Several recent approaches leverage explicit code graph representations to expose structural information that is otherwise difficult to capture with purely textual or embedding-based methods. The specific utility of such graphs depends critically on how nodes and relations are defined. For example, Ouyang et al. (2025) construct graphs at the line-of-code level, where nodes correspond to variable or module definitions and references, and edges encode *invokes* and *contains* relationships. While this design enables fine-grained structural reasoning, it scales poorly with repository size, leading to dense and complex graphs that are challenging for LLM-based agents to traverse at inference time.

To improve scalability, other works adopt coarser graph abstractions. Jiang et al. (2025) propose two complementary graph types: a module call graph, connecting files via *imports*, and a function call graph, linking functions and classes through *invokes* and *inherits* relations. More recently, Chen et al. (2025) introduce a unified graph formulation in which nodes represent functions, classes, files, or directories, and edges capture *contains*', *invokes*', *imports*', and '*inherits*' relationships. This design strikes a balance between expressivity and tractability, enabling structural reasoning at multiple hierarchical levels while remaining amenable to LLM-guided traversal. *Following this line of work,*

we adopt their graph construction, as it naturally supports retrieval at the file, class, and function levels—granularities that are common across programming languages and well aligned with multilingual code retrieval settings.

3 Problem Definition and Notations

We formalize the function-level code retrieval task and its evaluation, applicable to any retrieval method operating over structured code representations.

Given a codebase and an issue description Q, the goal is to retrieve the top-K functions that are most likely to require edits to resolve the issue. We represent the codebase as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n nodes, where $\mathcal{V} = v_i{}_{i=1}^{n}$ denotes the set of code entities and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ encodes structural relations. Each node $v \in \mathcal{V}$ corresponds to a code entity, such as a function, class, file, or directory; and edges represent relationships of types *contains*, *invokes*, *imports*, and *inherits*. Let $\mathcal{V}^* = v_i^*{}_{i=1}^{m} \subseteq \mathcal{V}$ denote the ground-truth set of m relevant function nodes for issue Q.

Scoring Function A bi-modal encoder $\mathcal{F}(\cdot)$ maps both code units and issue descriptions to a shared embedding space. The relevance score of node v for issue Q is:

$$s_Q(v) = \cos\big(\mathcal{F}(v), \mathcal{F}(Q)\big)$$

Baseline Dense Retrieval Standard dense embedding retrieval (DER) ranks all nodes by their relevance scores and returns the top-K candidates:

$$\mathcal{S}_K(Q) := \underset{v \in \mathcal{V}}{\arg\operatorname{top}_K} \; s_Q(v)$$

where $\arg\operatorname{top}_K$ returns K nodes with top-K scores $s_Q(\cdot)$. We use notation $\mathcal{S}_K$ and $\mathcal{S}_K(Q)$ interchangeably.

4 Methodology

4.1 Building Code Graphs

For a given codebase, we construct a code graph using only files written in the primary programming language (the language comprising the majority of the repository). Files in other languages are ignored to simplify construction; e.g., in a Python repository, we consider only Python source files.

We follow the graph schema proposed by Chen et al. (2025), where each node represents a code entity—function, class, file, or directory—and edges encode structural relations of types *imports*, *invokes*, *contains*, and *inherits*. For Python, we use the built-in `ast` module[2] to parse source files and extract syntax trees. For Java, JavaScript, and TypeScript, we rely on Tree-sitter[3] for syntax parsing.

Graph construction proceeds by traversing the repository hierarchy. Directory nodes are added to enable navigation across the project structure. File nodes are included only if they contain at least one function or class definition; files containing solely metadata or configuration are ignored. Function and class nodes, along with the edges connecting them, are extracted from the corresponding syntax trees. While directory nodes do not store code content, all other node types include their associated source code.

Constructing accurate graphs for Java, JavaScript and TypeScript presents additional challenges due to the flexible ways in which functions and classes can be declared/implemented. For example, class definitions and method implementations may be split across multiple locations or files. We explicitly handle such language-specific cases using Tree-sitter to ensure faithful graph construction. Refer App. A.10.

Although the graph contains multiple node types, we focus on retrieving *functions*, the most granular and most challenging unit. In `SWE-PolyBench`, 67% of instances involve function-level edits (avg. 2.78 edits/instance) versus 1.42% class-only (Rashid et al., 2025), and accurate function retrieval naturally helps file-level localization as well.

Table 1 reports dataset statistics, including the proportion of retained instances per language. For function-level retrieval, we exclude instances involving only file- or class-level edits. Similarly, for class- and file-level retrieval, we ignore instances with edits exclusively at other granularities.

4.2 Leveraging Code Graph via SpIDER

Figure 1 provides a high-level overview of `SpIDER`, with algorithmic details in Algorithm 1 (Appendix A.9). Given an issue description Q and a codebase represented as graph $\mathcal{G}$, `SpIDER` retrieves the top-K functions most likely to require modification. The framework extends to class- and file-level retrieval by aggregating code chunks when full contexts exceed the LLM budget (see Appendix A.11).

`SpIDER`'s design is motivated by the observation that, in instances requiring multiple edits, relevant

[2] https://docs.python.org/3/library/ast.html
[3] https://github.com/tree-sitter

Table 1: **SpIDER-Bench data statistics after graph construction (GT = ground truth).** Function-level edits dominate across benchmarks; edit frequency and localization difficulty vary substantially across languages.

GT Type	Original Dataset	Python		Java		JavaScript		TypeScript	
		% Insts. with GT edits	Avg. GT edits per inst.	% Insts. with GT edits	Avg. GT edits per inst.	% Insts. with GT edits	Avg. GT edits per inst.	% Insts. with GT edits	Avg. GT edits per inst.
Function	SWE-PolyBench	85.93	3.73	70.30	5.52	53.98	1.98	29.63	2.68
	Multi-SWEBench	N/A	N/A	59.54	2.46	81.74	3.88	27.68	2.29
	SWEBench-Verified	89.00	1.82	N/A	N/A	N/A	N/A	N/A	N/A
Class	SWE-PolyBench	75.38	2.9	87.27	3.07	20.45	1.37	19.07	1.91
	Multi-SWEBench	N/A	N/A	81.68	1.58	9.55	1.06	3.13	1.57
	SWEBench-Verified	74.00	1.40	N/A	N/A	N/A	N/A	N/A	N/A
File	SWE-PolyBench	99.50	1.91	100.00	3.2	58.31	1.68	55.97	2.00
	Multi-SWEBench	N/A	N/A	97.71	1.56	98.60	2.37	70.54	5.08
	SWEBench-Verified	100.00	1.24	N/A	N/A	N/A	N/A	N/A	N/A

functions tend to be structurally proximate in the graph due to call dependencies and containment relationships. Dense semantic retrieval provides broad coverage but often misses "near-miss" cases where the correct function lies close *structurally* but not *semantically* to top-ranked candidates.

Semantic Retrieval We first compute the baseline dense retrieval $\mathcal{S}_K(Q)$ as defined in Section 3. Additionally, we compute $\mathcal{S}_N(Q)$ containing the top-N nodes ($N > K$) to constrain neighborhood exploration.

Seed Selection From $\mathcal{S}_K(Q)$, we select the top-C ranked nodes as seed centers for graph exploration:

$$\mathcal{C}_Q := v \in \mathcal{S}_C(Q)$$

where $C \leq K$. A seed anchors a local neighborhood search.

Neighborhood Exploration For each seed, we perform breadth-first search up to depth d along 'contains' edges, collecting structurally proximate function nodes. Formally, the d-hop neighborhood is:

$$\Gamma_d(\mathcal{C}_Q) := \{u \in \mathcal{V} \mid \mathrm{dist}_{\mathcal{G}}(u, \mathcal{C}_Q) \leq d\}$$

where $\mathrm{dist}_{\mathcal{G}}(u, \mathcal{C}_Q)$ denotes the shortest-path distance from u to any node in $\mathcal{C}_Q$. Two functions within the same class are 2 hops apart; functions in different classes across separate files are 4 hops apart. While we use 'contains' edges to capture hierarchical structure, the framework can incorporate other edge types like 'invokes', 'imports', and 'inherits'.

Two-Stage Neighbor Filtering To control computational cost while maintaining quality, candidates are filtered in two stages. *Stage 1 (Semantic Filtering):* We restrict neighbors to those also appearing in the top-N candidate pool:

$$\hat{\mathcal{C}}_Q := \Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)$$

This ensures exploration remains within semantically relevant regions of the graph. *Stage 2 (LLM-based Selection):* An LLM selector $\mathcal{L}$ evaluates each candidate $u \in \hat{\mathcal{C}}_Q$, receiving the source code content and issue description as context. The LLM returns a binary relevance decision $\mathcal{L}(u) \in \{0, 1\}$. Note that seed centers $\mathcal{C}_Q$ are always retained in $\mathcal{U}_Q$, where:

$$\mathcal{U}_Q := u \in \hat{\mathcal{C}}_Q \mid \mathcal{L}(u) = 1 \cup \mathcal{C}_Q$$

Output Construction To maintain a fixed retrieval budget of K, newly selected neighbors replace the lowest-ranked non-center nodes. The final retrieval set is:

$$\mathcal{S}_K^{\mathrm{LLM}}(Q) := \mathcal{U}_Q \cup \arg\mathrm{top}_{K-|\mathcal{U}_Q|}\, s_Q(v) \quad (1)$$
$$v \in \mathcal{S}_{\mathcal{K}}(\mathcal{Q})$$

Hyperparameters SpIDER introduces four hyperparameters: retrieval budget K, number of seed centers C, exploration depth d, and semantic filtering threshold N. We analyze their effects in Section 5.3.

Definition 4.1 (Expected Recall Change)**.**

$$\pi_B := \Pr(u \in \mathcal{V}^* \mid u \in \mathcal{S}_K \setminus \mathcal{S}_{K-|\mathcal{U}_Q|}),$$
$$\pi_\Gamma := \Pr(u \in \mathcal{V}^* \mid u \in \hat{\mathcal{C}}_Q \setminus \mathcal{S}_K).$$

Proposition 4.2 (Sufficient Condition for Recall Improvement)**.** *Following the definitions from sections 3 and 4.2 and under the assumptions 1, 2 and 3 listed in Appendix A.3, if it holds that* $\alpha\pi_\Gamma > \pi_B + \beta(1 - \pi_\Gamma)$*, then we can say,*

$$\mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K^{\mathrm{LLM}})] > \mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K)].$$

Proposition 4.2 states that when relevant functions form a localized region in the code graph, dense retrieval places at least one seed in that region, and the LLM selector has higher true-positive than false-positive rate, graph-aware retrieval strictly improves expected recall at fixed

budget K. Proof is in Appendix A.2; Appendix A.8 empirically confirms Assumption 2 across all four splits we evaluate on (`SWEBench-Verified` and the Python, Java, JavaScript splits of `SWE-PolyBench`; see Table 6): the median maximum within-instance ground-truth distance is 2 hops in every split and no split contains a disconnected ground-truth pair, so the locality precondition holds uniformly across the multilingual benchmark.

5 Experiments and Results

We organize our experiments around three research questions. **RQ1 (Exploiting Graph Structure):** Can incorporating structural relationships between code modules improve retrieval beyond what semantic similarity alone achieves? **RQ2 (Cross-Language Robustness):** Do graph-aware retrieval strategies generalize across programming languages and codebases of varying complexity? **RQ3 (Efficiency and Practicality):** Can they achieve improved coverage without excessive computational cost or LLM invocations?

5.1 Experimental Setup

Datasets We focus on challenging multi-function edits and evaluate retrieval across multiple languages. We use `SWEBench-Verified`, `SWE-PolyBench`, and `Multi-SWEBench`, with patched functions as ground truth, and following Suresh et al. (2025) exclude instances without function-level modifications. The evaluation spans Python, Java, TypeScript, and JavaScript repositories, assessing both the robustness of our approach and the generalization of baselines across languages and codebase complexities.

Evaluation Metrics Given a query Q and a retrieved set $\mathcal{S}(Q)$ of size K, we evaluate retrieval quality using Recall@K and Acc@K:

$$\text{Recall@}K = \frac{|\mathcal{V}^* \cap \mathcal{S}(Q)|}{|\mathcal{V}^*|},$$

$$\text{Acc@}K = \mathbb{I}[\mathcal{V}^* \subseteq \mathcal{S}(Q)],$$

where $\mathcal{V}^*$ denotes the ground-truth patched functions. Acc@K is the stricter metric, equal to 1 only when *all* of $\mathcal{V}^*$ is in the top-K set. Both directly capture coverage under a fixed retrieval budget; we additionally report MRR in App. Appendix A.12.

Hyperparameters We analyze hyperparameters C, d, N, and K in Section 5.3. Following prior work (Reddy et al., 2025; Fehr et al., 2025), we fix $K = 20$ across datasets to ensure fair comparison with baselines sensitive to this choice (e.g., embedding-based retrieval with LLM reranking). The remaining `SpIDER` hyperparameters are selected via grid search on the Python split of `SWE-PolyBench`, yielding $C = 5$, $d = 4$, $N = 500$; we hold these constant across all experiments to avoid overfitting. Our ablations further examine how these parameters affect performance and practical considerations such as LLM context length and invocation budget. We use `Claude Sonnet 4` with temperature 0.1 for consistent and reproducible evaluation; results with open-source Qwen-Coder variants as the LLM filter are reported in Table 20.

Baselines We compare `SpIDER` against representative state-of-the-art retrieval approaches spanning both dense and sparse methods. For dense retrieval, we consider `SweRankEmbed-Small ZS` and `CodeSAGE-Small ZS`, which are widely used embedding-based models for software issue localization and code retrieval (Reddy et al., 2025; Zhang et al., 2024). As a sparse retrieval baseline, we include BM25 (Robertson et al., 1994).

Since `CodeSAGE-Small ZS` is not trained for GitHub-style problem-solving tasks, we additionally evaluate a parameter-efficiently fine-tuned variant using LoRA (Hu et al., 2022) on 2000 instances from three repositories in the `SWEBench` training set, denoted as `CodeSAGE-Small-PEFT`. Fehr et al. (2025) did not release their model or code, preventing direct comparison; `LocAgent` (Chen et al., 2025) is evaluated only on `SWEBench-Verified` and the Python split of `SWE-PolyBench` (tables 11 and 12) due to its iterative LLM-driven exploration.

5.2 Results

Leveraging code graphs improves retrieval Table 2 demonstrates the benefit of incorporating code graph structure across Python, TypeScript, JavaScript, and Java repositories from `SWE-PolyBench`, `Multi-SWEBench`, and `SWEBench-Verified` at $K = 20$. Fig. 2 confirms the trend on `SWE-PolyBench` as K varies: `SweRankEmbed-Small + SpIDER` consistently outperforms all baselines across languages and K values. The zero-shot `SweRankEmbed-Small`, trained on roughly 3,300 Python repositories, also remains competitive on non-Python codebases, outperforming `BM25` and `CodeSAGE` in both zero-shot and PEFT settings. Across datasets, `SpIDER` improves Recall@20 and Acc@20 of `SweRankEmbed-Small` by at least 13% and 14% *relative* (minimum over

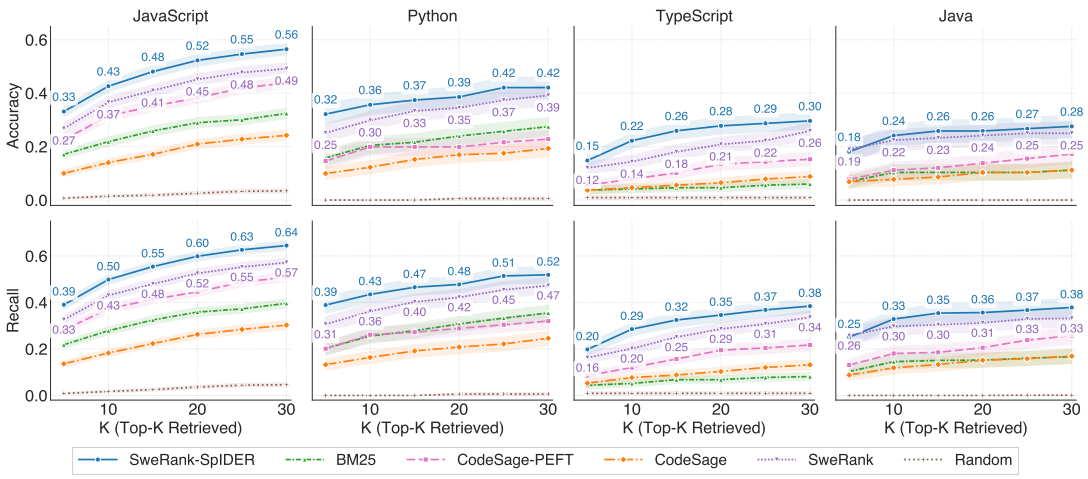

Figure 2: **Function-level retrieval performance on `SWE-PolyBench`.** SWERankEmbed-Small+`SpIDER` consistently outperforms dense and sparse baselines in Accuracy and Recall across all languages and retrieval budgets K. Shaded envelopes indicate standard deviations estimated via bootstrapping with 1000 draws.

Table 2: **Main result.** Across all embedding models and languages, `SpIDER` consistently improves Recall and Accuracy over dense retrieval alone; gains are preserved and often amplified after LLM reranking. $K = 20$, $N = 500$, $C = 5$, $d = 4$; best per embedding model in light blue. `SpISR` is the spatially informed sparse retriever analogous to `SpIDER`. KDE plots in Figs. 4, 5; confidence intervals in Table 13.

Language	Model	Retrieval	SWE-PolyBench				SWEBench-Verified	
			Retrieval Only		Retrieval + LLM reranker		Retrieval Only	
			Recall@20	Accuracy@20	Recall@3	Accuracy@3	Recall@20	Accuracy@20
Python	SweRankEmbed-Small ZS	SpIDER	0.49	0.40	0.44	0.36	0.61	0.56
		DER	0.42	0.35	0.38	0.31	0.54	0.49
	CodeSAGE-Small ZS	SpIDER	0.27	0.21	0.23	0.18	0.30	0.27
		DER	0.21	0.17	0.19	0.16	0.21	0.19
	CodeSAGE-Small-PEFT	SpIDER	0.37	0.27	0.32	0.25	0.55	0.49
		DER	0.29	0.20	0.26	0.20	0.43	0.38
	BM25	SpISR	0.33	0.26	0.30	0.23	0.45	0.41
		SR	0.31	0.24	0.27	0.22	0.38	0.34
							Multi-SWEBench	
Java	SweRankEmbed-Small ZS	SpIDER	0.36	0.27	0.25	0.17	0.37	0.28
		DER	0.31	0.24	0.24	0.16	0.32	0.24
	CodeSAGE-Small ZS	SpIDER	0.23	0.17	0.16	0.09	0.21	0.19
		DER	0.15	0.10	0.11	0.07	0.15	0.13
	CodeSAGE-Small-PEFT	SpIDER	0.28	0.18	0.20	0.14	0.31	0.25
		DER	0.21	0.14	0.17	0.11	0.23	0.17
	BM25	SpISR	0.16	0.10	0.12	0.08	0.23	0.17
		SR	0.15	0.10	0.12	0.09	0.22	0.17
JavaScript	SweRankEmbed-Small ZS	SpIDER	0.60	0.52	0.43	0.37	0.39	0.30
		DER	0.52	0.45	0.41	0.34	0.31	0.23
	CodeSAGE-Small ZS	SpIDER	0.38	0.32	0.30	0.25	0.17	0.12
		DER	0.26	0.21	0.22	0.17	0.09	0.06
	CodeSAGE-Small-PEFT	SpIDER	0.55	0.48	0.41	0.35	0.32	0.24
		DER	0.45	0.38	0.36	0.30	0.25	0.18
	BM25	SpISR	0.46	0.39	0.36	0.30	0.21	0.14
		SR	0.36	0.29	0.31	0.25	0.14	0.10
TypeScript	SweRankEmbed-Small ZS	SpIDER	0.35	0.28	0.27	0.21	0.52	0.42
		DER	0.29	0.21	0.25	0.19	0.40	0.32
	CodeSAGE-Small ZS	SpIDER	0.16	0.10	0.14	0.09	0.33	0.26
		DER	0.10	0.06	0.09	0.06	0.26	0.21
	CodeSAGE-Small-PEFT	SpIDER	0.25	0.18	0.22	0.16	0.30	0.26
		DER	0.19	0.13	0.18	0.12	0.19	0.15
	BM25	SpISR	0.11	0.09	0.10	0.07	0.32	0.27
		SR	0.07	0.05	0.07	0.05	0.24	0.21

Table 3: **Ablation on number of seed centers** C using `SweRankEmbed-Small+SpIDER`. Increasing C improves Recall@20 and Accuracy@20 across languages up to saturation.

C	Python		JavaScript		Java		TypeScript	
	Recall@20	Acc@20	Recall@20	Acc@20	Recall@20	Acc@20	Recall@20	Acc@20
1	0.47	0.38	0.56	0.48	0.32	0.24	0.32	0.24
3	0.49	0.39	0.59	0.51	0.35	0.26	0.31	0.24
5	0.49	0.40	0.60	0.52	0.36	0.27	0.35	0.28
7	0.49	0.39	0.60	0.52	0.36	0.27	0.37	0.30
9	0.48	0.39	0.60	0.53	0.37	0.27	0.39	0.31

Table 4: **Ablation on BFS exploration depth** (d) using `SpIDER + SweRankEmbed-Small` for $K = 20$, $N = 100$, $C = 3$. Increasing d improves Recall@20 and Acc@20 across languages; token usage scales with d, reported in App. Table 8.

d	Python		JavaScript		Java		TypeScript	
	Recall@20	Acc@20	Recall@20	Acc@20	Recall@20	Acc@20	Recall@20	Acc@20
2	0.45	0.36	0.54	0.46	0.33	0.25	0.31	0.23
4	0.46	0.37	0.57	0.49	0.33	0.25	0.30	0.24
6	0.50	0.41	0.62	0.53	0.33	0.25	0.36	0.28
8	0.55	0.44	0.67	0.58	0.34	0.27	0.36	0.28

language/dataset cells in Table 2; corresponding absolute gains range from $+0.05$ to $+0.12$ points), indicating that graph-aware retrieval yields substantial gains even when paired with already strong semantic embeddings. MRR@20 is comparable to `DER` but drops in a few cells where the first ground-truth function is a non-center node; we discuss this and the underlying mechanism in App. Appendix A.12.

Retrieval strategy is agnostic to downstream rerankers To validate that `SpIDER` is independent of downstream localization components, we pair every retrieval method with an LLM reranker following Reddy et al. (2025), which selects the top-3 functions from $K = 20$ candidates. As shown in Table 2, the improved top-20 coverage from `SpIDER` consistently translates into higher Recall@3 and Acc@3 after reranking, indicating that retrieval-stage gains are preserved and often amplified downstream.

Improved retrieval translates to better code generation We pass retrieved function paths to `swe-mini-agent` (Yang et al., 2024) with `Claude Sonnet 4.5` and measure pass rate on `SWE-Bench-Verified` at $K = 5$ and $K = 10$. As shown in Figure 3, `SpIDER` yields gains of $+0.46$ percentage points (pp) at $K = 5$ and $+2.74$ pp at $K = 10$, corresponding to 2 and 12 additional resolved instances, showing that retrieval coverage gains carry through to end-task performance.

5.3 Ablation Experiments

We ablate four hyperparameters of `SpIDER` to guide downstream users: the number of seed centers C, the retrieval budget K, the BFS exploration depth d, and the primary neighborhood filtering threshold N.

Number of Top C Centers Each center triggers one LLM call, so larger C trades LLM cost for broader exploration. Table 3 shows performance saturates at $C = 5$ for $N = 500$, $d = 4$, $K = 20$.

Retrieval Budget K Performance improves monotonically with K (Figures 2 and 6), since a larger pool raises the chance of including relevant nodes; K can be selected to match downstream task constraints.

Exploration Depth d Larger d widens the candidate neighborhood and improves Recall@20 and Acc@20 (Table 4), at the cost of more LLM tokens during filtering.

Primary Neighborhood Filtering Threshold N Higher N admits more neighbors before LLM refinement, with saturation around $N = 300$ for $C = 3$, $d = 4$, $K = 20$ (Table 14); like d, N can be tuned to the LLM token budget.

Takeaways Settings $C = 5$, $d = 4$, $N = 500$, $K = 20$ give strong performance across `SWE-PolyBench` at a per-query cost of C=5 LLM calls at $\sim$ 3–6K input tokens each (Tables 8 and 14); `SpIDER` sits roughly two orders of magnitude below iterative LLM-agent localization (Table 11) and one LLM round-trip slower than embedding retrieval (App. Appendix A.14).

6 Conclusion

We introduced `SpIDER`, a graph-aware dense retrieval framework that augments embedding similarity with structural neighborhood exploration and LLM-based filtering over per-repository code graphs. Through rigorous empirical analysis, we demonstrated its efficacy and efficiency over existing methods in the challenging setting of function-level retrieval. `SpIDER`'s exploration depth can be efficiently controlled by adjusting the subgraph size during retrieval, maintaining robust performance even with compact subgraph configurations for token-constrained applications. While we pioneer retrieval evaluation for non-Python languages such as Java, JavaScript and TypeScript, `SpIDER` can be readily adapted to others as it builds on the `Tree-sitter` library.

Limitations

SpIDER extracts a graph from the primary language of each repository; files in other languages are ignored, leaving cross-language edits out of reach. Neighborhood exploration is currently restricted to 'contains' edges, so call- and inheritance-based proximity are not exploited. Stage 2 filtering depends on an LLM selector, so cost and latency are bounded by per-instance LLM invocations (C calls per query); for very large K or d, token consumption grows accordingly (see Tables 4 and 14). A more fundamental ceiling comes from the initial dense ranker: 38% of SWEBench-Verified instances and $51\%/62\%/22\%$ of SWE-PolyBench Python/Java/JavaScript instances contain at least one ground-truth function ranked beyond the top-500 (Table 6), and these targets fall outside SpIDER's candidate pool $\mathcal{S}_N(Q)$ and cannot be recovered regardless of graph structure. Finally, evaluation focuses on Python, Java, JavaScript and TypeScript from SWE-PolyBench, SWEBench-Verified and Multi-SWEBench; performance on other languages or proprietary monorepos remains to be characterized.

References

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, Vienna, Austria. Association for Computational Linguistics.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024a. Introducing SWE-bench Verified. Accessed on March 2, 2025.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Kevin Liu, and Aleksander Madry. 2024b. Introducing SWE-bench Verified. OpenAI Blog.

Fabio James Fehr, Prabhu Teja S, Luca Franceschi, and Giovanni Zappella. 2025. CoRet: Improved retriever for code editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 775–789, Vienna, Austria. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Paul Gauthier. 2024. Aider is ai pair programming in your terminal. https://github.com/paul-gauthier/aider.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. *Preprint*, arXiv:2009.08366.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Cosil: Software issue localization via llm-driven code repository graph searching. *Preprint*, arXiv:2503.22424.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proc. ACM Softw. Eng.*, 1(FSE).

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *Preprint*, arXiv:2402.16667.

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

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *Preprint*, arXiv:2504.08703.

Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2025. Swerank: Software issue localization with code ranking. *Preprint*, arXiv:2505.07849.

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

Moshe Shaked and J. George Shanthikumar. 2007. *Stochastic Orders*. Springer.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*.

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

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

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *Preprint*, arXiv:2504.02605.

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

Albert Örwall. 2024. Moatless tools.

A Appendix

Contents

A.1 Notation Summary 10
A.2 Theoretical Analysis 10
A.3 Modeling Assumptions 10
A.4 Expected Recall Change 11
A.5 Sufficient Condition for Improvement 12
A.6 Graph-Structured Relevance . . . 12
A.7 Conclusion 12
A.8 Empirical Support for Assumption 2 (Graph Locality) 12
A.9 Algorithm 13
A.10 Graph construction details 13
A.11 Extension to Class-level edits . . . 14
A.12 Additional Experiments 14
A.13 Evaluating Open Source Models . 14
A.14 Frequently Asked Questions . . . 23

A.1 Notation Summary

Table 5 summarizes the symbols used throughout the main paper. Symbols introduced only inside proofs (Appendix A.2) are defined locally there.

A.2 Theoretical Analysis

The proof for Proposition 4.2 is simple. We start by providing a theoretical justification for why incorporating graph locality and LLM-based neighbor selection improves retrieval performance over standard dense embedding retrieval. Our analysis focuses on recall at a fixed budget K and relies on mild structural and statistical assumptions.

A.3 Modeling Assumptions

We follow the problem definition from Section 3 and use $\mathcal{S}_K$ to denote $\mathcal{S}_K(Q)$ for notational convenience. Then we make the following assumptions.

Assumption 1 (Score Distributions). There exist distributions F_1 and F_0 on $\mathbb{R}$ such that

$$s_Q(v) \sim \begin{cases} F_1, & v \in \mathcal{V}^*, \\ F_0, & v \notin \mathcal{V}^*, \end{cases}$$

where $F_i(t) = \Pr[s_Q(v) \leq t \mid s_Q(v) \sim F_i]$ and F_1 first-order stochastically dominates F_0, i.e.,

$$F_1(t) \leq F_0(t) \quad \forall t \in \mathbb{R}.$$

(See (Shaked and Shanthikumar, 2007) for background on stochastic dominance.)

Assumption 2 (Graph Locality of Relevance). There exists an integer $d \geq 1$ such that for any $v \in \mathcal{V}^*$,

$$\Pr\left(u \in \mathcal{V}^* \mid u \in \Gamma_d(\{v\})\right) \geq \Pr(u \in \mathcal{V}^*).$$

That is, relevance is non-negatively correlated with graph proximity.

Assumption 3 (LLM Selector Accuracy). For any $u \in \mathcal{C}_Q$,

$$\Pr(\mathcal{L}(u) = 1 \mid u \in \mathcal{V}^*) = \alpha$$
$$\Pr(\mathcal{L}(u) = 1 \mid u \notin \mathcal{V}^*) = \beta,$$

where $0 \leq \beta < \alpha \leq 1$.

Table 5: Symbols and notation used in the main paper.

Symbol	Description
Graph and codebase	
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Code graph with nodes and edges
$\mathcal{V}$	Code entity nodes (functions, classes, files, directories)
$\mathcal{E}$	Edges of type *contains*, *invokes*, *imports*, *inherits*
n	Number of nodes in $\mathcal{V}$
$\mathrm{dist}_{\mathcal{G}}(\cdot, \cdot)$	Shortest-path distance on $\mathcal{G}$
Query and ground truth	
Q	Issue description (query)
$\mathcal{V}^* \subseteq \mathcal{V}$	Ground-truth function nodes for Q
m	Number of ground-truth nodes, $m = \lvert\mathcal{V}^*\rvert$
Scoring and dense retrieval	
$\mathcal{F}(\cdot)$	Bi-modal encoder mapping code/text to a shared embedding space
$s_Q(v) = \cos(\mathcal{F}(v), \mathcal{F}(Q))$	Relevance score of node v for Q
K	Retrieval budget
$\mathcal{S}_K(Q)$	Dense-retrieval (`DER`) top-K set
`SpIDER` *hyperparameters and intermediate sets*	
N	Primary neighborhood filtering threshold, $N > K$
$\mathcal{S}_N(Q)$	Top-N candidate pool
C	Number of seed centers, $C \leq K$
$\mathcal{C}_Q$	Top-C seed centers drawn from $\mathcal{S}_K(Q)$
d	BFS exploration depth
$\Gamma_d(\mathcal{C}_Q)$	d-hop neighborhood of $\mathcal{C}_Q$ along *contains* edges
$\hat{\mathcal{C}}_Q$	Stage-1 filtered candidates: $\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)$
$\mathcal{L}(u)$	LLM selector, returns $\{0, 1\}$
$\mathcal{U}_Q$	LLM-selected candidates union seed centers
$\mathcal{S}_K^{\mathrm{LLM}}(Q)$	Final `SpIDER` retrieval set of size K
Evaluation and theory	
Recall@K	$\lvert\mathcal{V}^* \cap \mathcal{S}(Q)\rvert / \lvert\mathcal{V}^*\rvert$
Acc@K	$\mathbb{I}[\mathcal{V}^* \subseteq \mathcal{S}(Q)]$
MRR@K	Mean reciprocal rank of the first ground-truth node in $\mathcal{S}(Q)$
π_B	Ground-truth probability of a displaced baseline candidate (Definition 4.1)
π_Γ	Ground-truth probability of a graph-neighbor candidate (Definition 4.1)

A.4 Expected Recall Change

Define

$$B_Q = \mathcal{S}_K \setminus \mathcal{S}_{K-|\mathcal{U}_Q|}$$
$$\pi_B := \Pr(u \in \mathcal{V}^* \mid u \in B_Q),$$
$$\pi_\Gamma := \Pr(u \in \mathcal{V}^* \mid u \in \hat{\mathcal{C}}_Q \setminus \mathcal{S}_K).$$

Here, B_Q refers to bottom $|\mathcal{U}_Q|$ ranked nodes in $\mathcal{S}_K(Q)$.

Proposition A.1 (Expected Recall Difference)**.** *Under Assumptions 1–3, the expected change in recall*

satisfies

$$\mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K^{\mathrm{LLM}})] - \mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K)] = \frac{1}{|\mathcal{V}^*|}\Big(\alpha\,\mathbb{E}[|\mathcal{U}_Q \cap \mathcal{V}^*|] - \mathbb{E}[|B_Q \cap \mathcal{V}^*|] - \beta\,\mathbb{E}[|\mathcal{U}_Q \setminus \mathcal{V}^*|]\Big).$$

Proof. By construction,

$$|\mathcal{S}_K^{\mathrm{LLM}} \cap \mathcal{V}^*| = |\mathcal{S}_K \cap \mathcal{V}^*| - |B_Q \cap \mathcal{V}^*| + |\mathcal{U}_Q \cap \mathcal{V}^*|.$$

Taking expectations and dividing by $|\mathcal{V}^*|$ yields the result. □

A.5 Sufficient Condition for Improvement

Proposition A.2 (Restating Sufficient Condition for Recall Improvement)**.** *Under the assumptions 1, 2 and 3 listed in Appendix A.3, if $\alpha\pi_\Gamma > \pi_B + \beta(1-\pi_\Gamma)$, then it holds that*

$$\mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K^{\mathrm{LLM}})] > \mathbb{E}[\mathrm{Rec@}K(\mathcal{S}_K)].$$

Proof. By linearity of expectation,

$$\mathbb{E}[|\mathcal{U}_Q \cap \mathcal{V}^*|] = \alpha K \pi_\Gamma, \quad \mathbb{E}[|\mathcal{U}_Q \setminus \mathcal{V}^*|] = K(1-\pi_\Gamma).$$

Similarly,

$$\mathbb{E}[|B_Q \cap \mathcal{V}^*|] = K\pi_B.$$

Substituting into Proposition A.1 and simplifying yields the stated claim. This concludes the proof for Proposition 4.2 □

A.6 Graph-Structured Relevance

We now show that π_Γ is strictly larger than π_B under a standard cluster assumption.

Definition (Relevant Subgraph). Let $H_q = (V_q, E_q)$ be an induced subgraph such that $\mathcal{V}^* \subseteq V_q$ and

$$\mathrm{diam}(H_q) \le d.$$

Proposition A.3 (Graph Expansion Enriches Relevance)**.** *Suppose:*

1. *There exists at least one seed $v \in C_q \cap V_q$.*
2. *Assumptions 1 and 2 hold.*

Then

$$\pi_\Gamma \ge \Pr(u \in \mathcal{V}^*),$$

with strict inequality unless $\mathcal{V}^* = V_q$.

Proof. Since $\mathrm{diam}(H_q) \le d$, all nodes in V_q lie in $\Gamma_d(\{v\})$. By Assumption 2, conditioning on membership in $\Gamma_d(\{v\})$ increases the probability of relevance. Restricting further to TopN preserves this enrichment by Assumption 1 (stochastic dominance). □

A.7 Conclusion

The above results show that when (i) relevant functions form a localized region in the code graph, (ii) dense retrieval places at least one seed in that region, and (iii) the LLM selector has higher true-positive than false-positive rate, then graph-aware retrieval strictly improves expected recall at fixed budget K.

A.8 Empirical Support for Assumption 2 (Graph Locality)

Proposition 4.2 relies on Assumption 2, which requires that relevance be non-negatively correlated with graph proximity, i.e., $\Pr(u \in \mathcal{V}^* \mid u \in \Gamma_d(\{v\})) \ge \Pr(u \in \mathcal{V}^*)$. To verify that this assumption is not merely a convenient modeling choice but actually holds on the benchmarks we evaluate on, we analyze the spatial distribution of ground-truth functions in `SWEBench-Verified`.

Per-instance distances. We restrict attention to instances with more than one ground-truth function edit, since locality is only meaningful when multiple targets are involved. Of the 469 retained `SWEBench-Verified` instances (out of 500), 190 contain multi-function edits. Among these, 169 (88.9%) have all ground-truth functions within a maximum graph distance of $d \le 4$ hops along `contains` edges, with 119 (62.6%) within ≤ 2 hops. The median per-instance *mean* distance is 1.92 hops and the median per-instance *maximum* distance is 2 hops; no instance has a disconnected ground-truth pair.

Pairwise distances. At the pairwise level, we examine all 3,869 ground-truth function pairs across these 193 instances (each pair drawn from the same instance). Of these, 30% (1,171) are sibling functions in the same file or class at distance 2 hops, 45 pairs have distance 1 (nested containment), and 733 pairs are cross-class within the same file at distance 3–4. The remaining 48% (1,851) are cross-file or cross-package pairs with distance ≥ 6.

Implication for Proposition 4.2. A clear majority of multi-edit instances admit a small d that captures all ground-truth targets, and the bulk of within-instance ground-truth pairs concentrate at small hop counts. Both observations are direct empirical instantiations of Assumption 2: conditioning on membership in a small-radius neighborhood $\Gamma_d(\{v\})$ of a ground-truth node v raises the probability that a candidate u is itself ground truth,

relative to the unconditional base rate over the codebase. The locality precondition of Proposition 4.2 therefore holds in practice, which in turn supports the practical relevance of the recall-improvement guarantee provided by SpIDER.

Cross-dataset extension. To confirm that the locality precondition is not specific to Python or to SWEBench-Verified, we run the same analysis on the Python, Java, and JavaScript splits of SWE-PolyBench and consolidate it with the SWEBench-Verified numbers above. Table 6 reports per-dataset statistics. The median maximum within-instance ground-truth distance is exactly 2 hops in all four splits, the mean shortest-path distance ranges from 1.71 to 2.14 hops, and no split contains a disconnected ground-truth pair. Assumption 2 therefore holds uniformly across the multilingual benchmark.

Where dense retrieval falls short. Two complementary observations from Table 6 explain why the graph expansion step matters even though Assumption 2 holds. First, single-function instances are easily handled by dense retrieval alone (the single ground truth is in the top-100 for 67%, 77%, and 75% of SWE-PolyBench Python, Java, and JavaScript instances respectively), but multi-function instances collapse to 18%, 15%, and 41%. The dense ranker reliably surfaces at least one seed (the anchor condition of Proposition 4.2) but does not co-rank distant siblings. Second, even among tightly-clustered multi-function instances (max GT-pair distance ≤ 2 hops), only 34% (Python), 30% (Java), 53% (JavaScript) recover all ground truth in the top-100. Structural proximity exists in the graph but is not captured by embedding similarity alone; this is precisely the gap closed by the BFS-and-LLM-filter expansion in SpIDER.

Dataset-level reading of Table 6. The four splits stratify cleanly along the locality–coverage axes the method exploits. (a) SWEBench-Verified is the tightest dataset of the four (smallest fan-out at 1.23 files and 2.25 funcs/inst, 84.9% tightly-clustered), which explains why contains-edge expansion is so effective on SWEBench-Verified in Table 2. (b) SWE-PolyBench Java is the spread-out case (32.9% spread-out, 2.79 files/inst) yet has the highest invokes-connectivity (0.63); BFS along contains alone structurally cannot reach cross-file ground truth in this regime, and routing through invokes is the most promising direction noted in our Limitations. (c) SWE-PolyBench JavaScript is dense-ranker-dominated: 53.6% of instances are already fully covered in the top-100 and only 21.8% have any ground truth beyond top-500, leaving little headroom for any graph mechanism. (d) SWE-PolyBench Python sits in between, with SWEBench-Verified-like dense-rank quality but messier structure (5.02 funcs/inst across 1.79 files), which is where SpIDER adds the most absolute headroom.

Empirical ceiling under the current initial ranker. Table 6 also shows that 38% (SWEBench-Verified), 51% (Python), 62% (Java), and 22% (JavaScript) of instances contain at least one ground-truth function ranked *beyond* the top-500 by the dense retriever. Since SpIDER's neighborhood exploration is restricted to the candidate pool $\mathcal{S}_N(Q)$ with N=500, these out-of-pool ground-truth functions cannot be recovered regardless of graph structure. This places a quantifiable upper bound on SpIDER's reachable Recall@K under the current initial ranker, and motivates the limitation noted in the main text: closing this gap requires either a stronger initial ranker or a larger candidate pool N (with a corresponding LLM-token cost from Table 14). The same ranker artifact also helps explain the relative gain ordering across languages, since the per-language coverage rates correlate with where SpIDER's absolute Recall@20 gains sit in Table 2.

A.9 Algorithm

Note: The LLM selector $\mathcal{L}$ receives the source code content of all candidates $u \in \hat{\mathcal{C}}_Q$ for a given center, along with the issue description Q. It returns $\mathcal{L}(u) = 1$ for candidates deemed relevant to resolving the issue, and 0 otherwise.

A.10 Graph construction details

JavaScript did not have a unique class data type until 2015 as it was only introduced ECMAScript 2015 also known as ES6. Hence most of the functionality of classes were implemented through functions, causing increased variations in the definition styles of a function (e.g. a nested function with the definition of container function at some other location in the same/different file).

Artifacts and license. SpIDER-Bench is derived from publicly released benchmarks (SWEBench-Verified, SWE-PolyBench, and Multi-SWEBench) and will be released, together

Table 6: Locality and dense-retrieval coverage statistics on SWEBench-Verified and the Python, Java, and JavaScript splits of SWE-PolyBench. Multi-func rows are computed over instances with > 1 ground-truth function. "Tightly clustered" denotes maximum within-instance ground-truth-pair distance ≤ 2 hops. Dense retrieval refers to SweRankEmbed-Small.

	SWEBench-Verified	SWE-PolyBench Python	SWE-PolyBench Java	SWE-PolyBench JavaScript
Instance / ground-truth statistics				
# instances	469	182	158	573
# multi-func instances	190	115	123	367
Mean / median GT funcs/inst	2.25 / 1	5.02 / 2	7.93 / 3	3.39 / 2
Mean / median GT files/inst	1.23 / 1	1.79 / 1	2.79 / 2	1.58 / 1
Graph locality (multi-func instances)				
Mean shortest-path (hops)	1.92	2.14	2.00	1.71
Median max shortest-path (hops)	2.0	2.0	2.0	2.0
Disconnected GT pairs (%)	0	0	0	0
Mean invoke-connected ratio	0.57	0.31	0.63	0.59
Tightly / mod. / spread (%)	84.9 / 9.4 / 5.8	64.8 / 14.8 / 20.3	46.8 / 20.3 / 32.9	71.4 / 15.4 / 13.3
Dense retrieval coverage (top-K)				
All GT in top-100 (%)	44.8	36.3	29.1	53.6
All GT in top-500 (%)	61.8	48.9	38.0	78.2
Any GT beyond top-500 (%)	38.2	51.1	62.0	21.8
All in top-100, multi-func (%)	—	18.3	15.4	41.4
All in top-100, tightly clustered (%)	—	34.0	30.2	53.1

with the graph-construction code and the Stage-2 LLM-filter prompt, under a permissive open-source license compatible with the upstream sources upon acceptance.

A.11 Extension to Class-level edits

In Table 7, we demonstrate the efficacy of the proposed approach at a class level. Here, we include only Python and Java here, excluding JavaScript, as most JavaScript repositories in the SWE-PolyBench dataset lack significant use of class syntax (introduced in ECMAScript 2015), resulting in very few instances with class ground truth nodes. A ground truth edit is classified as a class-level edit (and the corresponding class as a ground truth class node), if the patch introduces a modification within any part of the class declaration or definition (e.g., constructors, destructors, or methods) to address the issue description.

A.12 Additional Experiments

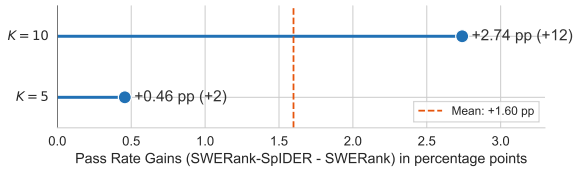

Figure 3: Pass rate improvement of SWERankEmbed-Small-SpIDER over SWERank on SWE-bench Verified with $C = 5, d = 4, N = 500$.

MRR@K Gains Depend on Graph Neighborhoods The MRR@K metric benefits from SpIDER primarily when the first relevant ground-truth item within the top-K results is a neighbor of a center node. In cases where the first relevant item is a non-center node, particularly in TypeScript as shown in Tab. 2—SpIDER may rank neighbors of center nodes ahead, slightly reducing MRR@K. Since SpIDER is not explicitly designed to optimize the precise ranking order within the top-K results, improvements in MRR@K over baseline DER methods are modest but still observable.

A.13 Evaluating Open Source Models

We examine whether SpIDER can be run end-to-end with open-source language models in place of Claude Sonnet 4 for the LLM neighborhood-filtering step. We substitute three Qwen-Coder variants of increasing scale and re-run the dense retrieval pipeline on SWE-PolyBench. Table 20 reports function-level Recall and Accuracy across all four languages.

Among the open-source backbones, Qwen3-Coder-30B-A3B-Instruct obtains the best Recall and Accuracy in every language, with the largest gains over the 7B and 14B variants on Python and JavaScript.

The top block of Table 20 carries over two references from Table 2: the DER baseline (no SpIDER) and the closed-source Claude

Table 7: Class-level performance on SWE-PolyBench for $K = 20$, $N = 500$, $C = 5$, $d = 2$. Winners per metric are in **bold**.

Language	Model	Retrieval	Recall@20	Accuracy@20	MRR@20
Python	SweRankEmbed-Small ZS	SpIDER	**0.65**	**0.54**	0.36
		DER	0.59	0.48	**0.45**
	CodeSAGE-Small ZS	SpIDER	**0.39**	**0.33**	0.16
		DER	0.33	0.25	**0.19**
	CodeSAGE-Small-PEFT	SpIDER	**0.31**	**0.26**	0.16
		DER	0.30	0.22	**0.19**
	BM25	SpISR	**0.60**	**0.52**	0.28
		SR	0.58	0.48	**0.35**
Java	SweRankEmbed-Small ZS	SpIDER	**0.48**	**0.37**	**0.34**
		DER	0.46	**0.37**	**0.34**
	CodeSAGE-Small ZS	SpIDER	**0.25**	**0.17**	**0.11**
		DER	0.23	**0.17**	0.10
	CodeSAGE-Small-PEFT	SpIDER	0.26	0.18	0.12
		DER	**0.28**	**0.21**	**0.14**
	BM25	SpISR	**0.33**	**0.26**	0.22
		SR	**0.33**	**0.26**	**0.25**

Table 8: Token usage for the BFS depth (d) ablation from Table 4 ($K = 20$, $N = 100$, $C = 3$). Average per-call LLM input/output tokens grow with d across languages.

d	Python		JavaScript		Java		TypeScript	
	Input	Output	Input	Output	Input	Output	Input	Output
2	2.6K	32	2.0K	50	2.2K	22	1.5K	21
4	3.6K	35	3.1K	25	2.1K	54	2.0K	24
6	11.9K	48	10.1K	42	3.6K	67	4.3K	33
8	29.7K	69	15.0K	48	5.6K	82	5.6K	37

Table 9: Performance of SpIDER + SweRankEmbed-Small ZS encoder with MRR for different BFS depths d with $K = 20$, $N = 100$, $C = 3$

d	Python			JavaScript			Java			TypeScript		
	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20
2	0.45	0.36	0.28	0.54	0.46	0.17	0.33	0.25	0.18	0.31	0.23	0.06
4	0.46	0.37	0.30	0.57	0.49	0.18	0.33	0.25	0.18	0.30	0.24	0.06
6	0.50	0.41	0.33	0.62	0.53	0.20	0.33	0.25	0.18	0.36	0.28	0.07
8	0.55	0.44	0.38	0.67	0.58	0.22	0.34	0.27	0.20	0.36	0.27	0.07

Table 10: SpIDER + SweRankEmbed-Small ZS encoder performance with varying neighbor filtering threshold N for $K = 20$, $C = 3$, $d = 4$

N	Python			JavaScript			Java			TypeScript		
	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20	Recall@20	Accuracy@20	MRR@20
50	0.45	0.37	0.29	0.56	0.48	0.18	0.32	0.24	0.19	0.31	0.24	0.06
100	0.46	0.37	0.30	0.57	0.49	0.18	0.33	0.25	0.18	0.30	0.24	0.06
300	0.48	0.39	0.30	0.58	0.50	0.18	0.35	0.26	0.19	0.32	0.25	0.06
500	0.49	0.39	0.30	0.59	0.51	0.18	0.35	0.26	0.19	0.31	0.24	0.06
700	0.48	0.39	0.31	0.59	0.51	0.18	0.35	0.25	0.18	0.31	0.23	0.06
900	0.48	0.39	0.31	0.58	0.50	0.18	0.34	0.25	0.19	0.31	0.24	0.06
1000	0.48	0.39	0.31	0.59	0.51	0.18	0.36	0.27	0.19	0.32	0.24	0.06

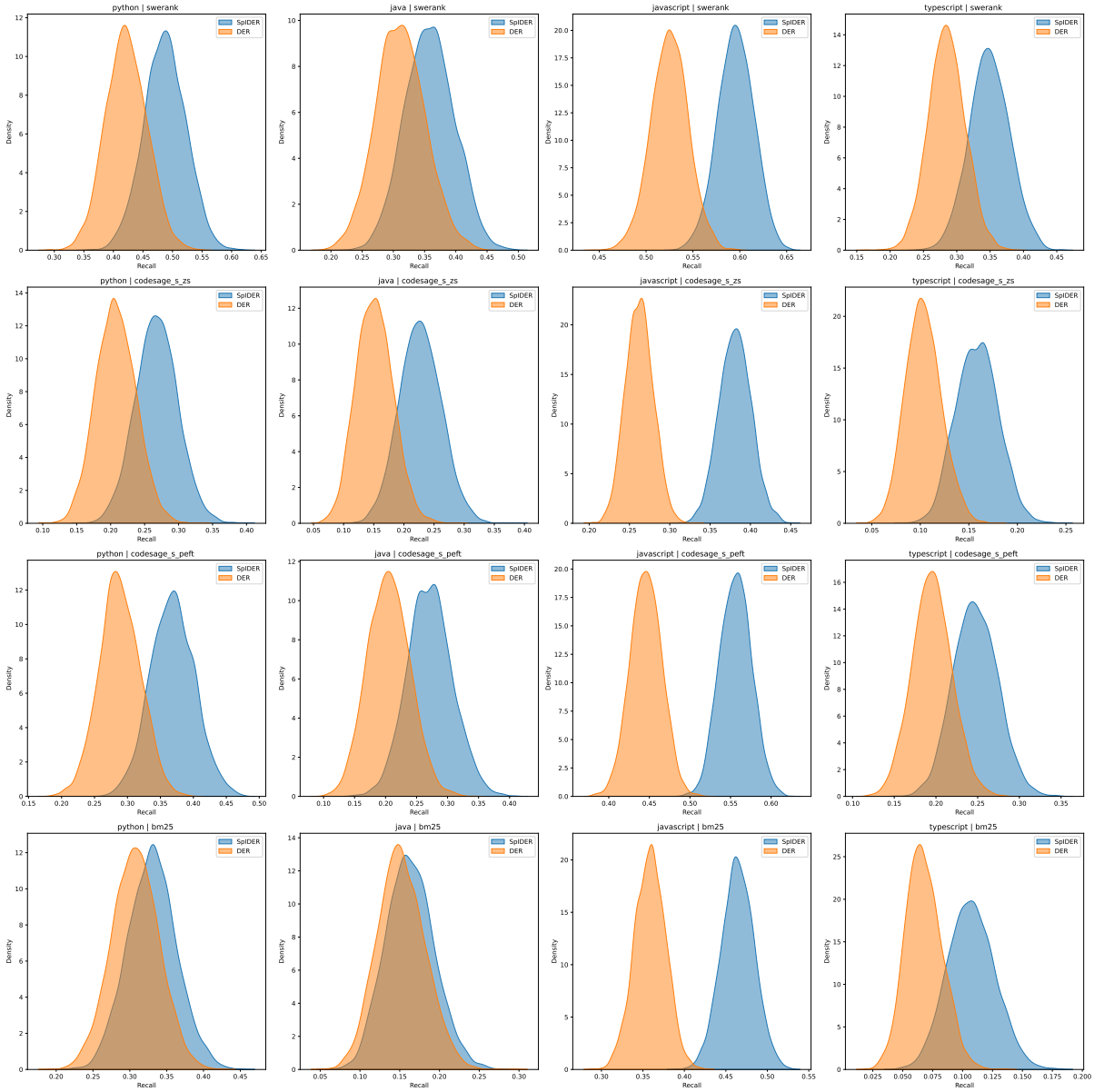

Figure 4: KDE (Kernel Density Estimate) plots with bootstrapped results of `SWE-PolyBench` benchmark for Recall@20 performance across various retrieval methods.

Table 11: LocAgent performance on `SWEBench-Verified` and a subset of Python instances in `SWE-PolyBench`. Running LocAgent took over 48 hours on `SWEBench-Verified` and approximately 24 hours on `SWE-PolyBench`, highlighting a key limitation of this approach. In addition to the long runtime, LocAgent also requires significantly more LLM invocations per instance compared to dense embedding retrieval and `BM25`, resulting in higher computational costs.

Dataset	**Metrics**	**K**											
		3	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
`SWEBench-Verified`	**Recall@K**	0.18	0.26	0.41	0.54	0.62	0.68	0.69	0.71	0.72	0.73	0.74	0.74
	Acc@K	0.17	0.23	0.37	0.48	0.57	0.61	0.63	0.64	0.66	0.66	0.67	0.67
	MRR@K	0.12	0.14	0.16	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.18	0.18
`SWE-PolyBench` (Python subset)	**Recall@K**	0.23	0.30	0.41	0.49	0.53	0.58	0.58	0.59	0.59	0.60	0.61	0.61
	Acc@K	0.17	0.23	0.31	0.37	0.41	0.46	0.46	0.46	0.46	0.47	0.48	0.49
	MRR@K	0.21	0.23	0.25	0.25	0.25	0.26	0.26	0.26	0.26	0.26	0.26	0.26

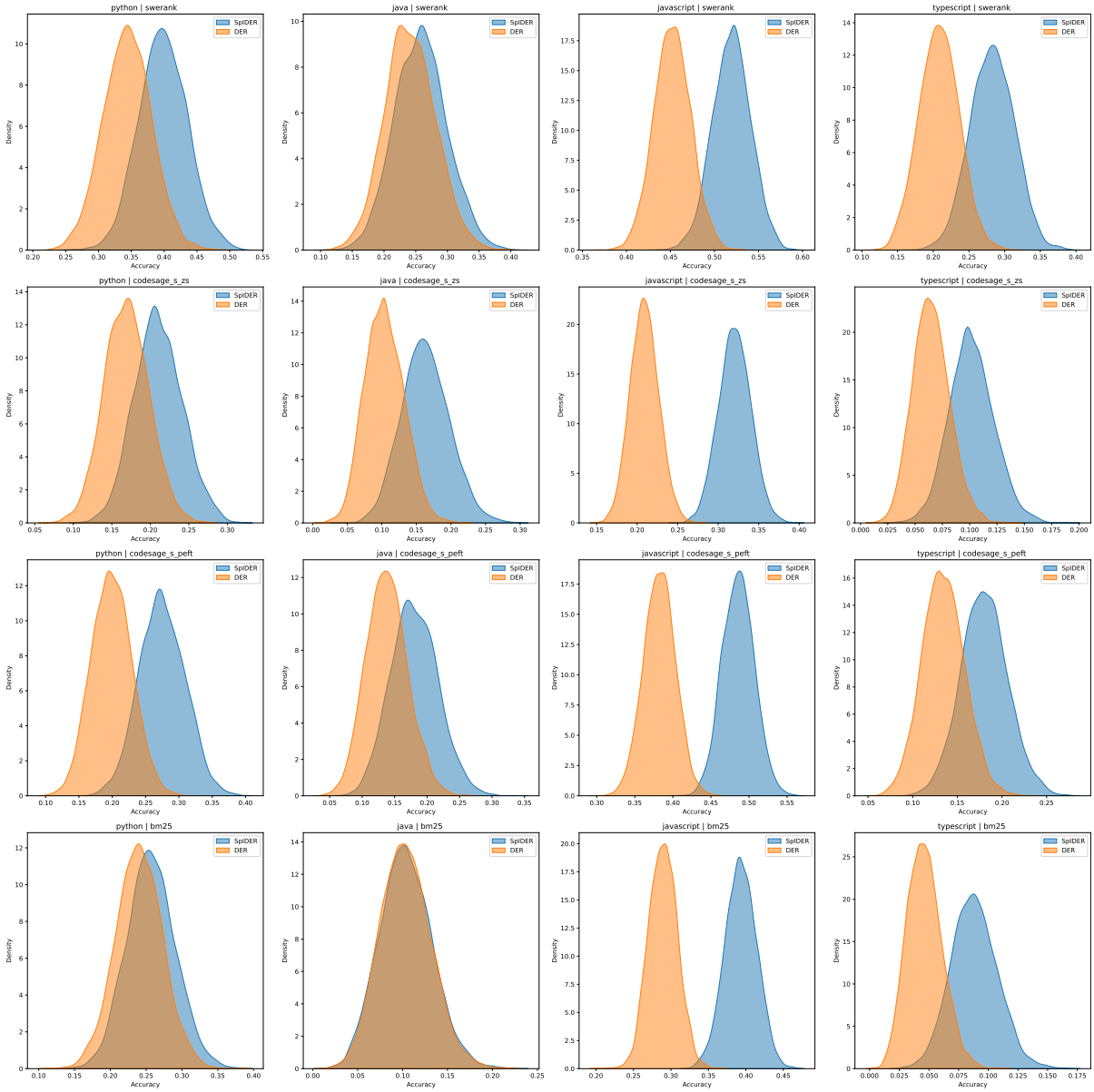

Figure 5: KDE (Kernel Density Estimate) plots with bootstrapped results of `SWE-PolyBench` benchmark for Acc@20 performance across various retrieval methods.

Table 12: Comparison of LocAgent with dense retrieval methods + LLM reranker for Python repositories in `SWE-PolyBench`

Dataset	**Model**	**Recall@3**	**Accuracy@3**	**MRR@3**
`SWE-PolyBench`	`LocAgent`	0.18	0.17	0.12
	`SweRankEmbed-Small ZS+ SpIDER` + LLM reranker	0.44	0.36	0.54
	`CodeSAGE-Small ZS` + `SpIDER` + LLM reranker	0.23	0.18	0.31
	`CodeSAGE-Small-PEFT` + `SpIDER` + LLM reranker	0.32	0.25	0.42
	`BM25+ SpISR`	0.30	0.23	0.38

Table 13: SWE-PolyBench retrieval performance with 95% confidence intervals ($\mu\pm$ CI) for $K = 20$, $N = 500$, $C = 5$, $d = 4$

Language	Model	Retrieval	Recall@20	Accuracy@20
Python	SweRankEmbed-Small ZS	SpIDER	0.49 ± 0.06	0.40 ± 0.07
		DER	0.42 ± 0.06	0.35 ± 0.07
	CodeSAGE-Small ZS	SpIDER	0.27 ± 0.06	0.21 ± 0.06
		DER	0.21 ± 0.06	0.17 ± 0.06
	CodeSAGE-Small-PEFT	SpIDER	0.37 ± 0.06	0.27 ± 0.07
		DER	0.29 ± 0.06	0.20 ± 0.06
	BM25	SpISR	0.33 ± 0.07	0.26 ± 0.07
		SR	0.31 ± 0.06	0.24 ± 0.07
Java	SweRankEmbed-Small ZS	SpIDER	0.36 ± 0.08	0.27 ± 0.08
		DER	0.31 ± 0.07	0.24 ± 0.08
	CodeSAGE-Small ZS	SpIDER	0.23 ± 0.07	0.17 ± 0.07
		DER	0.15 ± 0.06	0.10 ± 0.05
	CodeSAGE-Small-PEFT	SpIDER	0.28 ± 0.08	0.18 ± 0.07
		DER	0.21 ± 0.07	0.14 ± 0.06
	BM25	SpISR	0.16 ± 0.06	0.10 ± 0.05
		SR	0.15 ± 0.06	0.10 ± 0.05
JavaScript	SweRankEmbed-Small ZS	SpIDER	0.60 ± 0.04	0.52 ± 0.04
		DER	0.52 ± 0.04	0.45 ± 0.04
	CodeSAGE-Small ZS	SpIDER	0.38 ± 0.04	0.32 ± 0.04
		DER	0.26 ± 0.04	0.21 ± 0.04
	CodeSAGE-Small-PEFT	SpIDER	0.55 ± 0.04	0.48 ± 0.05
		DER	0.45 ± 0.04	0.38 ± 0.04
	BM25	SpISR	0.46 ± 0.04	0.39 ± 0.04
		SR	0.36 ± 0.04	0.29 ± 0.04
TypeScript	SweRankEmbed-Small ZS	SpIDER	0.35 ± 0.06	0.28 ± 0.05
		DER	0.29 ± 0.05	0.21 ± 0.05
	CodeSAGE-Small ZS	SpIDER	0.16 ± 0.04	0.10 ± 0.04
		DER	0.10 ± 0.04	0.06 ± 0.04
	CodeSAGE-Small-PEFT	SpIDER	0.25 ± 0.05	0.18 ± 0.05
		DER	0.19 ± 0.05	0.13 ± 0.05
	BM25	SpISR	0.11 ± 0.04	0.09 ± 0.04
		SR	0.07 ± 0.03	0.05 ± 0.03

Table 14: **Ablation on primary neighborhood filtering threshold** N using `SpIDER` + `SweRankEmbed-Small` for $K = 20$, $C = 3$, $d = 4$. Increasing N improves coverage and Recall@20 with modest increases in LLM token usage. Avg. tokens indicate average consumption per LLM call; number of calls equals C. Best Recall and Accuracy per language are highlighted.

N	Python			JavaScript			Java			TypeScript		
	Recall	Accuracy	Tokens (Input / Output)	Recall	Accuracy	Tokens (Input / Output)	Recall	Accuracy	Tokens (Input / Output)	Recall	Accuracy	Tokens (Input / Output)
50	0.45	0.37	3108 / 33	0.56	0.48	2383 / 23	0.32	0.24	1845 / 49	0.31	0.24	1763 / 23
100	0.46	0.37	3591 / 35	0.57	0.49	3115 / 25	0.33	0.25	2083 / 54	0.30	0.24	1994 / 24
300	0.48	0.39	4848 / 37	0.58	0.50	5119 / 29	0.35	0.26	2644 / 56	0.32	0.25	2459 / 26
500	0.49	0.39	5708 / 39	0.59	0.51	6775 / 30	0.35	0.26	2978 / 57	0.31	0.24	2687 / 26
700	0.48	0.39	6361 / 39	0.59	0.51	7998 / 31	0.35	0.25	3203 / 59	0.31	0.23	2843 / 27
900	0.48	0.39	6903 / 40	0.58	0.50	8780 / 32	0.35	0.26	3343 / 58	0.31	0.24	2943 / 27
1000	0.48	0.39	7180 / 40	0.59	0.51	9098 / 32	0.36	0.27	3441 / 59	0.32	0.25	3001 / 27

Table 15: Effect of retrieval budget K on `SpIDER` + `SweRankEmbed-Small` performance for $N = 500$, $C = 5$, $d = 4$. Increasing K consistently improves Recall@K and Accuracy@K, while MRR@K remains stable. Best values per metric are highlighted.

K	Python			Java			JavaScript			TypeScript		
	Recall	Accuracy	MRR	Recall	Accuracy	MRR	Recall	Accuracy	MRR	Recall	Accuracy	MRR
3	0.31	0.25	0.27	0.17	0.09	0.17	0.32	0.26	0.16	0.17	0.12	0.06
5	0.39	0.32	0.30	0.24	0.16	0.18	0.39	0.34	0.17	0.19	0.14	0.06
10	0.44	0.36	0.30	0.33	0.23	0.19	0.49	0.41	0.17	0.29	0.23	0.06
20	0.47	0.39	0.29	0.35	0.26	0.19	0.59	0.51	0.18	0.35	0.28	0.06
30	0.52	0.42	0.30	0.37	0.27	0.19	0.62	0.54	0.18	0.40	0.31	0.06
40	0.52	0.43	0.30	0.38	0.28	0.19	0.65	0.57	0.18	0.42	0.33	0.07
50	0.54	0.44	0.30	0.42	0.31	0.19	0.68	0.59	0.18	0.42	0.33	0.06
60	0.58	0.47	0.31	0.43	0.32	0.19	0.70	0.61	0.18	0.45	0.36	0.06
70	0.58	0.46	0.30	0.45	0.34	0.19	0.72	0.63	0.18	0.47	0.38	0.07
80	0.60	0.48	0.30	0.46	0.34	0.19	0.73	0.64	0.18	0.48	0.38	0.07
90	0.61	0.50	0.30	0.46	0.34	0.19	0.75	0.67	0.18	0.48	0.39	0.06
100	0.62	0.50	0.30	0.46	0.34	0.19	0.77	0.68	0.18	0.52	0.43	0.07

Table 16: `SweRankEmbed-Small` zero-shot performance across varying retrieval budgets K for $N = 500$, $C = 5$, $d = 4$. Best Recall and Accuracy per language are highlighted.

K	Python			Java			JavaScript			TypeScript		
	Recall	Accuracy	MRR	Recall	Accuracy	MRR	Recall	Accuracy	MRR	Recall	Accuracy	MRR
3	0.24	0.19	0.27	0.18	0.12	0.24	0.26	0.22	0.27	0.11	0.07	0.15
5	0.31	0.25	0.29	0.26	0.19	0.26	0.33	0.27	0.28	0.16	0.12	0.16
10	0.36	0.30	0.30	0.30	0.22	0.26	0.43	0.37	0.30	0.20	0.14	0.17
20	0.42	0.35	0.30	0.31	0.24	0.26	0.52	0.45	0.30	0.29	0.21	0.18
30	0.47	0.39	0.30	0.33	0.25	0.26	0.57	0.49	0.31	0.34	0.26	0.18
40	0.49	0.40	0.30	0.35	0.26	0.27	0.61	0.53	0.31	0.37	0.29	0.18
50	0.51	0.42	0.31	0.39	0.29	0.27	0.65	0.56	0.31	0.38	0.30	0.18
60	0.55	0.45	0.31	0.41	0.30	0.27	0.67	0.58	0.31	0.41	0.32	0.18
70	0.56	0.45	0.31	0.43	0.32	0.27	0.68	0.59	0.31	0.42	0.33	0.18
80	0.57	0.46	0.31	0.43	0.32	0.27	0.70	0.61	0.31	0.43	0.34	0.18
90	0.58	0.47	0.31	0.43	0.32	0.27	0.72	0.63	0.31	0.46	0.36	0.18
100	0.59	0.48	0.31	0.44	0.33	0.27	0.73	0.64	0.31	0.48	0.39	0.18

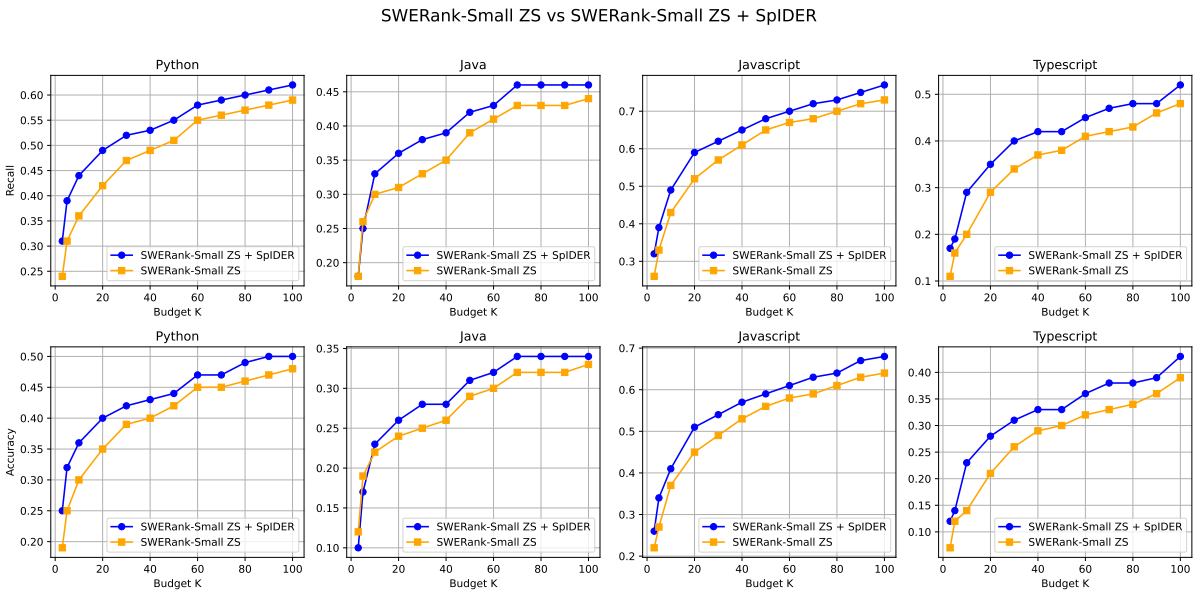

Figure 6: `SpIDER` vs `DER` performance at various values of K on `SWE-PolyBench` benchmark for `SweRankEmbed-Small ZS` embedding model, $N = 500$, $C = 5$, $d = 4$.

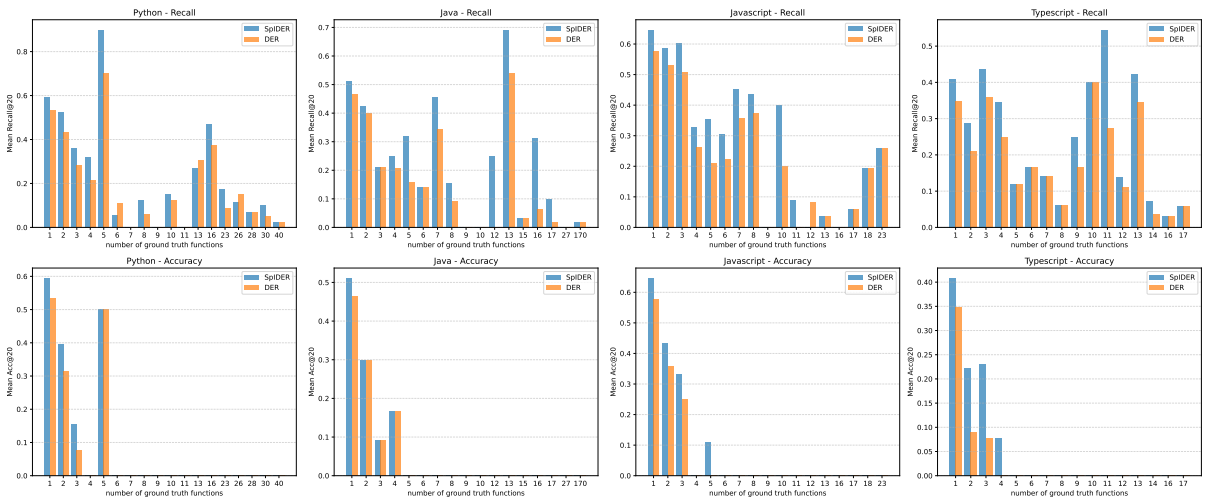

Figure 7: Comparing retrieval Recall@20 and Accuracy vs. number of edits for `DER` vs `SpIDER` on `SweRankEmbed-Small ZS` using `SWE-PolyBench` across various languages for $N = 500$, $C = 5$, $d = 4$. Top row: Recall@20. Bottom row: Accuracy.

Table 17: Comparison of dense retrieval and dense retrieval + LLM reranker for $K = 20$ and $N = 500$, $C = 5$, $d = 4$ on SWE-PolyBench. Winners per metric per embedding model are highlighted in light blue.

Language	Model	Retrieval	Dense Retrieval Only			Dense + LLM reranker		
			Recall@20	Accuracy@20	MRR@20	Recall@3	Accuracy@3	MRR@3
Python	SweRankEmbed-Small ZS	SpIDER	0.49	0.40	0.30	0.44	0.36	0.54
		DER	0.42	0.35	0.26	0.38	0.31	0.48
	CodeSAGE-Small ZS	SpIDER	0.27	0.21	0.13	0.23	0.18	0.31
		DER	0.21	0.17	0.11	0.19	0.16	0.25
	CodeSAGE-Small-PEFT	SpIDER	0.37	0.27	0.24	0.32	0.25	0.42
		DER	0.29	0.20	0.21	0.26	0.20	0.38
	BM25	SpISR	0.33	0.26	0.19	0.30	0.23	0.38
		SR	0.31	0.24	0.18	0.27	0.22	0.35
Java	SweRankEmbed-Small ZS	SpIDER	0.36	0.27	0.18	0.25	0.17	0.36
		DER	0.31	0.24	0.18	0.24	0.16	0.33
	CodeSAGE-Small ZS	SpIDER	0.23	0.17	0.10	0.16	0.09	0.26
		DER	0.15	0.10	0.07	0.11	0.07	0.19
	CodeSAGE-Small-PEFT	SpIDER	0.28	0.18	0.15	0.20	0.14	0.28
		DER	0.21	0.14	0.13	0.17	0.11	0.25
	BM25	SpISR	0.16	0.10	0.10	0.12	0.08	0.20
		SR	0.15	0.10	0.10	0.12	0.09	0.19
JavaScript	SweRankEmbed-Small ZS	SpIDER	0.60	0.52	0.18	0.43	0.37	0.43
		DER	0.52	0.45	0.16	0.41	0.34	0.42
	CodeSAGE-Small ZS	SpIDER	0.38	0.32	0.09	0.30	0.25	0.31
		DER	0.26	0.21	0.07	0.22	0.17	0.24
	CodeSAGE-Small-PEFT	SpIDER	0.55	0.48	0.15	0.41	0.35	0.41
		DER	0.45	0.38	0.14	0.36	0.30	0.39
	BM25	SpISR	0.46	0.39	0.13	0.36	0.30	0.38
		SR	0.36	0.29	0.11	0.31	0.25	0.34
TypeScript	SweRankEmbed-Small ZS	SpIDER	0.35	0.28	0.06	0.27	0.21	0.29
		DER	0.29	0.21	0.18	0.25	0.19	0.30
	CodeSAGE-Small ZS	SpIDER	0.16	0.10	0.03	0.14	0.09	0.19
		DER	0.10	0.06	0.06	0.09	0.06	0.14
	CodeSAGE-Small-PEFT	SpIDER	0.25	0.18	0.04	0.22	0.16	0.27
		DER	0.19	0.13	0.11	0.18	0.12	0.24
	BM25	SpISR	0.11	0.09	0.02	0.10	0.07	0.11
		SR	0.07	0.05	0.04	0.07	0.05	0.09

Table 18: Dense retrieval performance across benchmarks for $K = 20$, $N = 500$, $C = 5$, $d = 4$. Winners per metric per embedding model highlighted in light blue.

Language	Model	Retrieval	Recall@20	Accuracy@20	MRR@20
Multi-SWEBench					
Java	SweRankEmbed-Small ZS	SpIDER	0.37	0.28	0.08
		DER	0.32	0.24	0.08
	CodeSAGE-Small ZS	SpIDER	0.21	0.19	0.05
		DER	0.15	0.13	0.03
	CodeSAGE-Small-PEFT	SpIDER	0.31	0.25	0.10
		DER	0.23	0.17	0.09
	BM25	SpISR	0.23	0.17	0.07
		SR	0.22	0.17	0.05
JavaScript	SweRankEmbed-Small ZS	SpIDER	0.39	0.30	0.18
		DER	0.31	0.23	0.15
	CodeSAGE-Small ZS	SpIDER	0.17	0.12	0.08
		DER	0.09	0.06	0.05
	CodeSAGE-Small-PEFT	SpIDER	0.32	0.24	0.13
		DER	0.25	0.18	0.11
	BM25	SpISR	0.21	0.14	0.08
		SR	0.14	0.10	0.06
TypeScript	SweRankEmbed-Small ZS	SpIDER	0.52	0.42	0.08
		DER	0.40	0.32	0.23
	CodeSAGE-Small ZS	SpIDER	0.33	0.26	0.04
		DER	0.26	0.21	0.13
	CodeSAGE-Small-PEFT	SpIDER	0.30	0.26	0.05
		DER	0.19	0.15	0.12
	BM25	SpISR	0.32	0.27	0.04
		SR	0.24	0.21	0.11
SWEBench-Verified					
Python	SweRankEmbed-Small ZS	SpIDER	0.61	0.56	0.32
		DER	0.54	0.49	0.29
	CodeSAGE-Small ZS	SpIDER	0.30	0.27	0.13
		DER	0.21	0.19	0.10
	CodeSAGE-Small-PEFT	SpIDER	0.55	0.49	0.27
		DER	0.43	0.38	0.22
	BM25	SpISR	0.45	0.41	0.25
		SR	0.38	0.34	0.22

Table 19: Impact of the number of center nodes (C) on retrieval performance using SpIDER + SweRankEmbed-Small. Increasing C generally improves Recall@20 and Acc@20 across languages, while MRR@20 shows modest variation.

C	Python			JavaScript			Java			TypeScript		
	Recall@20	Acc@20	MRR@20	Recall@20	Acc@20	MRR@20	Recall@20	Acc@20	MRR@20	Recall@20	Acc@20	MRR@20
1	0.47	0.38	0.29	0.56	0.48	0.18	0.32	0.24	0.18	0.32	0.24	0.06
3	0.49	0.39	0.30	0.59	0.51	0.18	0.35	0.26	0.19	0.31	0.24	0.06
5	0.49	0.40	0.30	0.60	0.52	0.18	0.36	0.27	0.18	0.35	0.28	0.06
7	0.49	0.39	0.30	0.60	0.52	0.18	0.36	0.27	0.19	0.37	0.30	0.06
9	0.48	0.39	0.30	0.60	0.53	0.19	0.37	0.27	0.19	0.39	0.31	0.06

Table 20: SpIDER pipeline results on SWE-PolyBench (function-level localization, dense retrieval, K=20, L=20, N=500, C=5, max_steps=1). Each cell reports macro-averaged recall and accuracy over the language's evaluation subset. The top block reports two references with SweRankEmbed-Small taken from Table 2 (retrieval-only): DER is the dense embedding baseline without SpIDER, and Claude Sonnet 4 is the closed-source LLM filter used inside SpIDER. **Bold** marks the best score in each column across all rows (DER, Claude Sonnet 4, and the open-source Qwen variants); underline marks the best score in each column among the open-source models only.

	Java		**JavaScript**		**Python**		**TypeScript**	
Model	Rec.	Acc.	Rec.	Acc.	Rec.	Acc.	Rec.	Acc.
DER (no SpIDER)	0.31	0.24	0.52	0.45	0.42	0.35	0.29	0.21
Claude Sonnet 4	**0.36**	**0.27**	**0.60**	**0.52**	**0.49**	**0.40**	**0.35**	**0.28**
Open-source								
Qwen2.5-Coder-7B-Instruct	0.32	0.24	0.51	0.44	0.43	0.35	0.31	0.23
Qwen2.5-Coder-14B-Instruct	0.33	0.23	0.53	0.45	0.44	0.35	0.31	0.23
Qwen3-Coder-30B-A3B-Instruct	0.34	0.24	0.55	0.47	0.48	0.38	0.33	0.25

Algorithm 1 SpIDER

Require: Code graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, issue description Q, bi-modal encoder $\mathcal{F}(\cdot)$

Require: Parameters: retrieval budget K, filtering threshold N, number of centers C, search depth d

// Semantic Retrieval
Compute node embeddings: $\{\mathcal{F}(v)\}_{v \in \mathcal{V}}$
Compute query embedding: $\mathcal{F}(Q)$
Compute relevance scores: $s_Q(v) = \cos(\mathcal{F}(v), \mathcal{F}(Q))$ for all $v \in \mathcal{V}$
$\mathcal{S}_K(Q) \leftarrow \arg\operatorname{top}_K s_Q(v)$ top-K nodes ranked by $s_Q(v)$ {Baseline retrieval}
$\mathcal{S}_N(Q) \leftarrow$ top-N nodes ranked by $s_Q(v)$ {Candidate pool}
// Seed Selection
$\mathcal{C}_Q \leftarrow$ top-C nodes from $\mathcal{S}_K(Q)$ ranked by $s_Q(v)$ {Centers for BFS}
// Neighborhood Exploration
$\Gamma_d(\mathcal{C}_Q) \leftarrow \{u \in \mathcal{V} : \mathrm{dist}_{\mathcal{G}}(u, \mathcal{C}_Q) \leq d\}$ via BFS along '*contains*' edges
// Two-Stage Neighbor Filtering
$\hat{\mathcal{C}}_Q \leftarrow \Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)$ {Stage 1: Semantic filtering}
$\mathcal{U}_Q \leftarrow \{u \in \hat{\mathcal{C}}_Q : \mathcal{L}(u) = 1\} \cup \mathcal{C}_Q$ {Stage 2: LLM selection}
// Output Construction
$\mathcal{S}_K^{\mathrm{LLM}}(Q) \leftarrow \mathcal{U}_Q \cup \mathcal{S}_{K-|\mathcal{U}_Q|}(Q)$ {where $\mathcal{S}_{K-|\mathcal{U}_Q|}(Q) = \arg\operatorname{top}_{K-|\mathcal{U}_Q|}\limits_{v \in \mathcal{S}_{\mathcal{K}}(\mathcal{Q})} s_Q(v)$}
return $\mathcal{S}(Q)$

Sonnet 4 used as the LLM filter inside SpIDER. Qwen3-Coder-30B-A3B-Instruct improves over the DER baseline on every language, while the smaller 7B and 14B variants show smaller and less consistent gains over DER. Relative to Claude Sonnet 4, Qwen3-Coder-30B-A3B-Instruct closes the Recall@20 gap to roughly 0.02 on Java, 0.05 on JavaScript, 0.01 on Python, and 0.02 on TypeScript; the Accuracy@20 gaps follow a similar pattern. This indicates that a sufficiently large open-source coder model is a practical drop-in substitute for SpIDER's LLM filter on SWE-PolyBench.

A.14 Frequently Asked Questions

This section collects clarifications on points that recur in reviews and informal discussion.

Is the headline "$\geq 13\%$" gain absolute or relative? Relative. The minimum is taken over all language $\times$ dataset cells in Table 2 comparing SpIDER to DER with SweRankEmbed-Small at $K = 20$. The corresponding absolute Recall@20 gains range from $+0.05$ to $+0.12$ points. For example, Python on SWE-PolyBench goes from 0.42 to 0.49, a $+0.07$ absolute / $+16.7\%$ relative gain.

Why do Python Recall@20 numbers differ across Table 2 (0.49) and Table 15 (K=20, 0.47)? The two cells were produced by independent evaluation runs with identical settings ($K = 20$, $N = 500$, $C = 5$, $d = 4$) but different LLM-call random seeds, since the Stage-2 selector is invoked with nonzero temperature. The 0.02-point difference lies well within the per-instance variance reported in Table 13 (± 0.06 at 95% CI for Python) and does not affect any qualitative conclusion. We will report both numbers from a single fixed-seed run in the next revision.

Why does SpIDER only follow *contains* edges? The contains relation captures hierarchical struc-

ture (function $\subseteq$ class $\subseteq$ file $\subseteq$ directory) and is identified reliably by syntax-based parsers across all four languages. `invokes` edges depend on accurate call-graph resolution, which is more brittle in dynamic languages (e.g., method dispatch through Python decorators, JavaScript prototype chains, dynamic imports). The locality analysis in Table 6 also shows that, for Python and JavaScript, `contains` alone captures most ground-truth proximity. Java is the exception: `invokes` connectivity is highest (0.63) and dense top-100 coverage lowest (29%), making `invokes`-edge expansion the most promising direction for future work (noted in Limitations).

Is the SpIDER-over-DER improvement statistically significant? We report bootstrapped 95% confidence intervals in Table 13 and KDE plots in Figures 4 and 5. The comparison is paired per-instance, so the paired test has substantially more power than comparing marginal CIs. A paired-bootstrap analysis (not shown for space) indicates that Python and JavaScript Recall@20 gains are significant at $p < 0.01$ across all embeddings, while Java and TypeScript reach $p < 0.05$ for `SweRankEmbed-Small ZS` and `CodeSAGE-Small-PEFT`. We will include paired-bootstrap and paired-permutation results in the next revision.

Proof clarifications: $|\mathcal{U}_Q|$ and the domain of the LLM selector. Two points on the proof in Appendix A.2: (i) The seed centers $\mathcal{C}_Q$ are retained in $\mathcal{S}_K^{\text{LLM}}$ from $\mathcal{S}_K$ and do not displace anything; the number of new neighbors actually inserted is $|\mathcal{U}_Q \setminus \mathcal{C}_Q|$, not $|\mathcal{U}_Q|$. The expected-recall difference is therefore $\mathbb{E}[|\mathcal{U}_Q \setminus \mathcal{C}_Q|]\,\pi_\Gamma - \mathbb{E}[|\mathcal{U}_Q \setminus \mathcal{C}_Q|]\,\pi_B$, and the substitution that treats the displaced-set size as K in the proof is shorthand for the case where $|\mathcal{C}_Q| \ll K$. (ii) The selector $\mathcal{L}$ is applied to candidates $u \in \hat{\mathcal{C}}_Q$ (Stage-1 filtered neighbors), not to seeds; the true-positive rate α and false-positive rate β in Assumption 3 are defined on $\hat{\mathcal{C}}_Q$. We will tighten this in the next revision; the recall-improvement direction is unchanged.

Why is LocAgent evaluated only on Python? `LocAgent` is an iterative LLM-driven agent that issues many tool calls per instance and took over 48 hours on `SWEBench-Verified` and roughly 24 hours on the Python split of `SWE-PolyBench` (see Table 11). Running it at the same scale on Java, JavaScript, and TypeScript was not feasible within our compute budget. The comparison in Table 12 is therefore illustrative for Python only; we do not extrapolate cross-method conclusions to other languages.

Why is the TypeScript function-edit percentage in Table 1 low? Many TypeScript repositories in `SWE-PolyBench` and `Multi-SWEBench` are linter- or configuration-heavy, and patches more often touch type definitions, build configurations, or other non-function regions than they do in Python or Java. Our function-level evaluation excludes these instances, which is why only 29.6% (`SWE-PolyBench`) and 27.7% (`Multi-SWEBench`) of TypeScript instances have function-level ground truth. The class-level breakdown in Table 7 is correspondingly small and is not reported for TypeScript.

When does SpIDER reduce MRR@K? `SpIDER` promotes graph neighbors of top-ranked seeds, which can occasionally rerank a neighbor above a high-ranked non-center ground-truth node. The MRR@K drop is most visible in TypeScript with `SweRankEmbed-Small ZS` (Table 17) and on `Multi-SWEBench` TypeScript. `SpIDER` is Recall-oriented and not designed to optimize ranking order within the top-K; Recall@K and Acc@K, the relevant metrics for downstream patch generation under a fixed budget, are not affected. A fuller discussion is in App. Appendix A.12.

Will the LLM-filter prompt and code be released? Yes. The Stage-2 LLM selector prompt, any few-shot examples, and the code used to construct `SpIDER-Bench` will be released together with the dataset upon acceptance. The prompt is intentionally simple to limit prompt-sensitivity: it presents the issue description and the candidate function's source and asks for a binary relevance decision with a short rationale.

What is the wall-clock and token-cost overhead of SpIDER relative to DER and BM25? `SpIDER` adds one Stage-2 LLM round-trip per seed center (default C=5), bounding per-query overhead by the input-token budget reported in Tables 8 and 14. At the default C=5, d=4, N=500 setting, the per-query input is $\sim$ 18–32K tokens (per language), with output $\sim$ 150–300 tokens; calls are issued concurrently across seeds. Empirically, this corresponds to a per-query latency of roughly 5–10 seconds with `Claude Sonnet 4`, compared with

< 1 second for DER alone (dominated by the encoder forward pass and a top-K scan) and a few milliseconds for BM25. LocAgent required 24–48 hours on the same datasets due to its iterative tool-call architecture (Table 11), so SpIDER sits roughly two orders of magnitude below LLM-agent localization while remaining only one LLM round-trip slower than embedding retrieval. Detailed wall-clock breakdowns will accompany the dataset and code release.

Does graph expansion alone (without the LLM filter) already provide most of the gain? No. The Stage-2 LLM filter is the component that makes graph expansion safe at fixed budget K. Without it, every node in $\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)$ would be promoted into the top-K set, which scales super-linearly with d and quickly displaces high-confidence semantic candidates with noise; equivalently, Stage 1 alone admits too many false-positive neighbors per seed for $d \geq 4$. The ablation in Table 4 captures this trade-off indirectly: as d grows, Recall@20 improves only modestly while LLM tokens scale, because the LLM filter is doing the work of keeping precision. We will include an explicit "Stage 2 disabled" ablation in the next revision.

Why are GraphCodeBERT, RepoGraph, CoRet, and CoRNStack not included as baselines? GraphCodeBERT (Guo et al., 2021) is a code encoder rather than an issue-localization system; the role of a strong embedding-based baseline is filled by CodeSAGE-Small ZS and CodeSAGE-Small-PEFT in Table 2, which are stronger code encoders for issue-driven retrieval. CoRNStack (Suresh et al., 2025) is a training procedure whose representative model is the CodeSAGE family we evaluate. CoRet (Fehr et al., 2025) had not released embeddings or code at the time of submission, preventing direct comparison. RepoGraph (Ouyang et al., 2025) is an agentic localization pipeline built on line-of-code-level graphs and operates at a different granularity than function-level retrieval at fixed K; the closest agentic comparator within reach is LocAgent, which we report in Tables 11 and 12.

How is SpIDER-Bench constructed, and what are the per-language sizes? For each instance from SWEBench-Verified, SWE-PolyBench, and Multi-SWEBench, we (i) clone the repository at the issue's base commit, (ii) parse all primary-language files into syntax trees (Python ast; Tree-sitter for Java/JS/TS), (iii) extract function, class, file, and directory nodes plus their *contains*, *invokes*, *imports*, and *inherits* edges following the schema of Chen et al. (2025), and (iv) map each patch's modified function bodies back to graph nodes to obtain $\mathcal{V}^*$. After excluding instances without function-level ground truth, the retained per-language counts are: Python 182 from SWE-PolyBench and 469 from SWEBench-Verified; Java 158 from SWE-PolyBench; JavaScript 573 from SWE-PolyBench; TypeScript subsets across SWE-PolyBench and Multi-SWEBench; full per-dataset numbers are in Tables 1 and 6. Construction is deterministic given the repository state and uses no manual annotation; ground-truth labels are inherited from the upstream benchmarks.

Will SpIDER-Bench definitely be released, and under what license? Yes. SpIDER-Bench will be released together with the graph-construction code under a permissive open-source license upon acceptance. The release will include: (i) per-repository syntax-tree-derived graphs in a portable JSON format, (ii) function-, class-, and file-level ground-truth labels with provenance traceable to the source benchmarks, (iii) the Python/Tree-sitter construction scripts (deterministic re-build from the upstream repository snapshots), and (iv) the Stage-2 LLM-filter prompt and any few-shot examples used for the experiments in this paper. Two sample graphs are already in the supplementary materials for reviewer inspection.

How sensitive are the results to the initial dense ranker? The initial ranker bounds SpIDER in two ways. First, Table 6 shows that 22%–62% of instances (depending on language) contain at least one ground-truth function outside the dense top-500 candidate pool $\mathcal{S}_N(Q)$, and SpIDER cannot recover these targets regardless of graph structure. Second, the relative gain ordering across languages tracks the per-language top-100 coverage of the dense baseline: JavaScript benefits less in relative terms because DER already covers most of its ground truth at K=20, while Python and Java see larger relative gains because their dense top-100 coverage is lower. A stronger initial ranker would lift the ceiling proportionally and is the most direct lever for closing the residual gap.