

SpIDER: Spatially Informed Dense Embedding Retrieval for Software Issue Localization

Shravan Chaudhari^{1,2}

Rahul Thomas Jacob³

Jiajun Cao³

Shihab Rashid³

Mononito Goswami³

Christian Bock³

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-Bench⁴, 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 across every language and benchmark in SpIDER-Bench: by at least 13% relative (+0.05 to +0.12 absolute) along containment edges, rising to at least

27% relative (+0.11 to +0.19 absolute) once call edges are explored.

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

¹Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

²Work done while interning at AWS AI Labs

³AWS AI Labs, Seattle, WA, USA

⁴Dataset is available at <https://huggingface.co/datasets/AmazonScience/SpIDER-Bench>, and code at <https://github.com/amazon-science/SpIDER>.

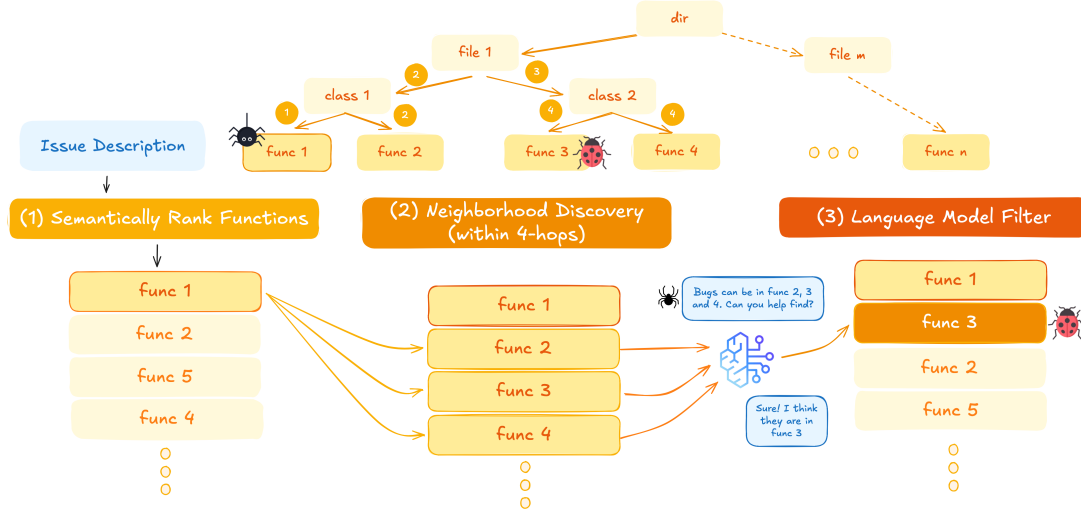


Figure 1: **SpIDER workflow.** SpIDER augments standard dense embedding retrieval with graph-aware spatial exploration and LLM-based reasoning to improve function-level code localization under a fixed retrieval (K). Given an issue description, all functions are first ranked by semantic similarity, and 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 Section A.4 (Algorithm 1).

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), particularly SWEBench-Lite, SWEBench-Verified, and LocBench (Jimenez et al., 2024; 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 bi-modal 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 corre-

lations 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 (Jimenez et al., 2024).

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 sub-graphs 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 multi-lingual 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:

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

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

$$\mathcal{S}_K(Q) := \arg \top_K s_Q(v)_{v \in \mathcal{V}}$$

where $\arg \top_K$ 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` module⁵ to parse source files and extract syntax trees. For Java, JavaScript, and TypeScript, we rely on `Tree-sitter`⁶ 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.6.

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.

⁵<https://docs.python.org/3/library/ast.html>

⁶<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

4.2 Leveraging Code Graph via SpIDER

Figure 1 provides a high-level overview of SpIDER, with algorithmic details in Algorithm 1 (Section A.4). 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 Section A.7).

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

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

where $\text{dist}_{\mathcal{G}}(u, \mathcal{C}_Q)$ 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 full prompt is in Section A.5. 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. Only candidates the baseline missed consume budget, so let $\mathcal{P} := (\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)) \setminus \mathcal{S}_K$ be the new proposals and $\mathcal{R} \subseteq \mathcal{S}_K$ the protected items, namely the seed centers and any accepted candidate already in $\mathcal{S}_K$. Writing $\mathcal{A} \subseteq \mathcal{P}$ for the set admitted after Stage-2 selection, deduplication and capacity truncation, $J = |\mathcal{A}| \leq K - |\mathcal{R}|$, and $\mathcal{D}_J$ for the J lowest-ranked members of $\mathcal{S}_K \setminus \mathcal{R}$, the final retrieval set is

$$\mathcal{S}_K^{\text{LLM}}(Q) := (\mathcal{S}_K \setminus \mathcal{D}_J) \cup \mathcal{A}. \quad (1)$$

Seed centers already lie in $\mathcal{S}_K$ and displace nothing.

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.

Let $\eta := \mathbb{E}|\mathcal{A} \cap \mathcal{V}^*|/\mathbb{E}|\mathcal{A}|$ denote the *admitted precision* and $\tau := \mathbb{E}|\mathcal{D}_J \cap \mathcal{V}^*|/\mathbb{E}|\mathcal{D}_J|$ the *frontier relevance*, both defined whenever $\mathbb{E}|\mathcal{A}| > 0$.

Proposition 4.1 (Exact exchange). *Write $\Delta\text{Rec}@K := \text{Rec}@K(\mathcal{S}_K^{\text{LLM}}) - \text{Rec}@K(\mathcal{S}_K)$. For every realization of Q , the labels and the policy,*

$$\Delta\text{Rec}@K = \frac{|\mathcal{A} \cap \mathcal{V}^*| - |\mathcal{D}_J \cap \mathcal{V}^*|}{m}, \quad (2)$$

and hence, at fixed m and with no further assumptions,

$$\mathbb{E}[\Delta\text{Rec}@K] > 0 \iff \eta > \tau. \quad (3)$$

SpIDER is therefore a strict exchange. A promotion helps only if the admitted function is more likely to be edited than the item it displaces, so the comparison is against the *eviction frontier* τ , the relevance of a rank- K dense hit, not the repository base rate. Graph proximity can clear that base rate and still fall below τ . Since Equation (3) is an equivalence, neither locality nor a selector with $\alpha > \beta$ implies improvement on its own. The selector still governs the outcome, because η is determined by α and β composed with the top- N gate (Section A.2.3), but it must move η past τ . Section A.2 proves these statements, treats $\text{Acc}@K$, and models how candidate quality decays with distance, yielding conditions for a finite useful radius and an LLM-pivotal regime; Section A.9 reports edge types where it fails.

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 $\text{Recall}@K$ and $\text{Acc}@K$:

$$\begin{aligned} \text{Recall}@K &= \frac{|\mathcal{V}^* \cap \mathcal{S}(Q)|}{|\mathcal{V}^*|}, \\ \text{Acc}@K &= \mathbb{I}[\mathcal{V}^* \subseteq \mathcal{S}(Q)], \end{aligned}$$

where $\mathcal{V}^*$ denotes the ground-truth patched functions. $\text{Acc}@K$ 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. A.8.

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, and App. A.11 verifies from the per-language sweeps that this configuration is at or near each language’s optimum. 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 23.

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 addition-

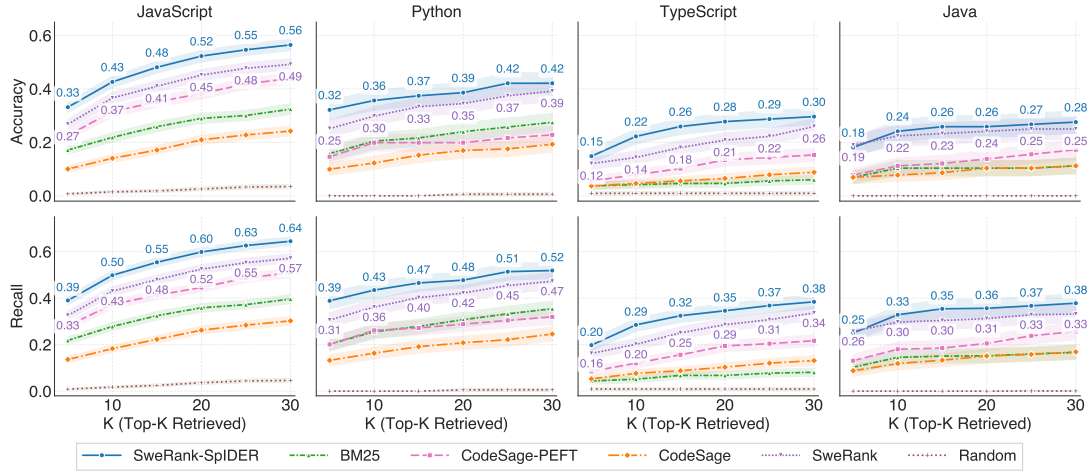


Figure 2: **Function-level retrieval performance on SWE-PolyBench.** SWERankEmbed-Small+SpIDER consistently outperforms dense and sparse baselines in Accuracy and Recall across all languages and retrieval budgets K . Shaded envelopes indicate standard deviations estimated via bootstrapping with 1000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

ally 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. This adaptation budget is small, so CodeSAGE-Small-PEFT remains a comparatively weak dense baseline; the load-bearing dense comparison throughout is SweRankEmbed-Small ZS, which is trained on roughly 3300 repositories. 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. Beyond these, App. A.10 compares SpIDER against a lexical grep baseline and the OpenHands agent (Wang et al., 2025) at matched K , and App. A.14 explains why other graph-based localizers are excluded.

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 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. A.8.

Which edge types carry the structural signal

Table 3 ablates the edge type used for neighborhood exploration on SWE-PolyBench at $K = 20$, $N = 500$, $C = 5$. Expansion along ‘invokes’ edges outperforms ‘contains’-only expansion in every language, and combining the two is best overall, raising the minimum relative Recall@20 gain over DER across the four languages from 13% to 27% (Python 0.42 $\rightarrow$ 0.60, Java 0.31 $\rightarrow$ 0.42, JavaScript 0.52 $\rightarrow$ 0.66, TypeScript 0.29 $\rightarrow$ 0.48). This ordering follows the locality analysis in Table 6: Java combines the highest invoke-connectivity (0.63) with the lowest dense top-100 coverage (29%), and is correspondingly the split where ‘contains’-only exploration cannot structurally reach cross-file ground truth. ‘imports’ and ‘inherits’, by contrast, add nothing over DER outside Python, since no *inherits* edge in our schema is incident to a function node and *imports* edges touch function nodes only in Python. The gains are not free: routing through ‘invokes’ raises per-instance token consumption by roughly an order of magnitude. We therefore retain ‘con-

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 the edge type used for neighborhood exploration**, on SWE-PolyBench with SweRankEmbed-Small1 at $K = 20$, $N = 500$, $C = 5$ (single fixed-seed run). “Tok” is the average total (input + output) LLM tokens per instance; DER issues no LLM call, hence 0. Best Recall@20 and Acc@20 per language in light blue. For the denser *invokes* graph we cap exploration at the 100 highest-ranked neighbors per query before Stage-2 filtering, to bound the token budget. *inherits* expansion is degenerate at function granularity: no *inherits* edge in our schema is incident to a function node in any of the four languages, so it can surface no function beyond DER. We report the Python run as empirical confirmation, since it returns exactly the DER set, and mark the remaining languages N/A, since the configuration is inapplicable rather than unmeasured.

Exploration edges / depth	Python			Java			JavaScript			TypeScript		
	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok
DER (no exploration)	0.42	0.35	0	0.31	0.24	0	0.52	0.45	0	0.29	0.21	0
<i>contains</i> / $d=4$	0.49	0.40	12.9K	0.36	0.27	7.8K	0.60	0.52	14.9K	0.35	0.28	8.3K
<i>invokes</i> / $d=2$	0.56	0.50	109.9K	0.42	0.30	87.4K	0.65	0.58	55.5K	0.47	0.41	92.7K
<i>imports</i> / $d=2$	0.44	0.38	7.9K	0.31	0.24	4.6K	0.52	0.45	5.2K	0.29	0.21	4.9K
<i>inherits</i> / $d=2$	0.42	0.35	6.8K	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
<i>contains</i> / $d=4$ + <i>invokes</i> / $d=2$	0.60	0.52	110.1K	0.42	0.29	90.1K	0.66	0.58	60.8K	0.48	0.42	98.6K

tains’ as the default configuration reported in Table 2, and treat ‘invokes’ as the setting of choice when the token budget allows. Gains from structural edges are significant under paired per-instance tests. Paired bootstrap ($B=10,000$) and two-sided paired permutation agree in every cell, giving $p < 0.01$ on Recall@20 in all four languages, with Java Acc@20 under ‘contains’ alone the single exception, resolved once ‘invokes’ edges are added. Meanwhile ‘imports’ and ‘inherits’ are uniformly non-significant (App. Table 20). App. A.12 works through cases where each edge type recovers

ground truth that dense retrieval misses, and where the Stage-2 filter errs.

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 ampli-

Table 4: **Hands-tied downstream code generation.** mini-swe-agent is restricted to the top- K retrieved functions with no open grep, and run once with DER and once with SpIDER at the same budget on a SWEbench-Verified subset of SpIDER-Bench. Resolve rate is the fraction of instances whose generated patch passes; best per K in light blue. Tokens are the total per instance for DER and for the better SpIDER variant at that K .

K	Resolve rate $\uparrow$			Tokens/inst. $\downarrow$	
	DER	<i>contains</i> /4	<i>invokes</i> /2	DER	best SpIDER
5	0.584	0.623	0.611	102.6K	62.2K
10	0.611	0.681	0.658	96.4K	81.5K
15	0.626	0.638	0.646	122.3K	97.6K
20	0.650	0.673	0.677	185.3K	87.6K

fied downstream.

Improved retrieval translates to better code generation We test whether retrieval gains survive to the end task under a *hands-tied* protocol: mini-swe-agent (Yang et al., 2024) is restricted to the top- K retrieved functions with no open grep, so the retrieved set is the only thing that varies. Running it once with DER and once with SpIDER at matched budget (Table 4), the better SpIDER variant resolves more instances than DER at every K , with the largest margin at $K=10$ (+7.0 pp, 18 additional resolved instances). The margin compresses at larger K , where a wider budget lets DER include more of the ground truth on its own. The effect is strongest on efficiency: SpIDER with *contains* expansion at $K=10$ resolves more than DER at $K=20$ (0.681 vs. 0.650) while using 44% of the tokens (81.5K vs. 185.3K), because it reaches comparable coverage at a smaller budget. App. A.8 reports the complementary setting in which the agent keeps full tool access.

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. App. Table 19 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 (App. Table 9), at the cost of more LLM tokens during filtering. Input tokens grow $8.2\times$ from $d = 4$ to $d = 8$ on Python (App. Table 8), so the deployed $d = 4$ is a budget choice rather than a measured optimum, which Section A.2.6 makes precise: pricing inference per candidate moves the useful radius inward.

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.

Stage-2 Filtering Disabling the selector promotes every Stage-1 neighbor and abandons the fixed budget, so its recall bounds the *reach* of graph expansion rather than anything achievable at K . SpIDER captures 79–89% of that reach while returning only K functions (App. Table 21), so the selector’s admission rate α , not the graph, is what limits η . Removing the filter would not help, since at fixed K the sign is set by $\eta > \tau$.

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 ~ 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. 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 configurable over edge types, but the configuration reported throughout Table 2 uses ‘contains’ edges alone; the edge-type comparison in Table 3, which finds ‘invokes’ expansion stronger in every language, was run on SWE-PolyBench only, so the SWEBench-Verified and Multi-SWEBench cells do not reflect it. Inheritance-based proximity is not exploited at all, since no ‘inherits’ edge is incident to a function node under our schema. Stage 2 filtering depends on an LLM selector, so cost and latency are bounded by per-instance LLM invocations (C calls per query), and the selector rather than the graph is what now bounds achievable recall within the candidate pool: SpIDER reaches 79–89% of the unfiltered neighborhood’s recall (Table 21), so the remainder is lost to filtering rather than to unreachable structure; App. A.12 characterizes the two failure modes, an over-restrictive selector that discards ground truth reached from a tangential seed and an over-permissive one that lets neighbor flooding evict it. For very large K or d , token consumption grows accordingly (see Tables 9 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.
- William Evans, Claire Kenyon, Yuval Peres, and Leonard J. Schulman. 2000. **Broadcasting on trees and the Ising model**. *The Annals of Applied Probability*, 10(2).
- 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/Aider-AI/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. **Graphcode{bert}: Pre-training code representations with data flow**. In *International Conference on Learning Representations*.
- 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. **Issue localization via llm-driven iterative code 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**. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages

- 436–464, Miami, Florida, USA. Association for Computational Linguistics.
- Russell Lyons. 1989. [The Ising model and percolation on trees and tree-like graphs](#). *Communications in Mathematical Physics*, 125:337–353.
- Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. 2025. [Tool-integrated reinforcement learning for repo deep search](#). *Preprint*, arXiv:2508.03012.
- Sean MacAvaney, Nicola Tonellotto, and Craig Macdonald. 2022. [Adaptive re-ranking with a corpus graph](#). In *CIKM*, pages 1491–1500.
- 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. 1977. [The probability ranking principle in IR](#). *Journal of Documentation*, 33(4):294–304.
- Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. 1994. [Okapi at trec-3](#). In *Text Retrieval Conference*.
- M. Shaked and J.G. Shanthikumar. 2007. *Stochastic Orders*. Springer Series in Statistics. Springer New York.
- 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*.
- Steven K. Thompson. 1990. [Adaptive cluster sampling](#). *Journal of the American Statistical Association*, 85(412):1050–1059.
- 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, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. [Openhands: An open platform for AI software developers as generalist agents](#). 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](#). *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*.
- Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Weikang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, Jiyan He, Shuxin Zheng, and Yunfang Wu. 2026. [One tool is enough: Reinforcement learning of LLM agents for repository-level code navigation](#). In *Forty-third International Conference on Machine Learning*.
- Albert Örwall. 2024. [Moatless tools](#).

A Appendix

Contents

A.1 Notation Summary	12
A.2 Theoretical Analysis	12
A.3 Empirical Support for Graph Locality	19
A.4 Algorithm	20
A.5 Stage-2 LLM Filter Prompt	20
A.6 Graph construction details	21
A.7 Extension to Class-level edits	21
A.8 Additional Experiments	22
A.9 Edge-Type Ablation: Statistical Significance	24
A.10 Comparison with Lexical and Agentic Retrieval	32
A.11 Per-Language Hyperparameter Sensitivity	33
A.12 Qualitative Analysis	33
A.13 Evaluating Open Source Models	34
A.14 Frequently Asked Questions	34

A.1 Notation Summary

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

A.2 Theoretical Analysis

A.2.1 Setup and exact accounting

Fix an issue Q with edited set $\mathcal{V}^*$, $m = |\mathcal{V}^*| \geq 1$. Recall $\mathcal{P} = (\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)) \setminus \mathcal{S}_K$, the protected baseline set $\mathcal{R} \subseteq \mathcal{S}_K$ (seed centers and accepted candidates already in $\mathcal{S}_K$), the admitted set $\mathcal{A} \subseteq \mathcal{P}$ with $J = |\mathcal{A}| \leq K - |\mathcal{R}|$, and the displaced set $\mathcal{D}_J$ consisting of the J lowest-ranked members of $\mathcal{S}_K \setminus \mathcal{R}$, so that $\mathcal{S}_K^{\text{LLM}} = (\mathcal{S}_K \setminus \mathcal{D}_J) \cup \mathcal{A}$ has size K .

Three modelling points deserve emphasis, because the earlier version of this analysis did not enforce them. First, the seed centers already belong to $\mathcal{S}_K$; treating $\mathcal{U}_Q$ as new insertions double-counts them, and the budget-consuming set is $\mathcal{U}_Q \setminus \mathcal{S}_K$. Second, J is a *random* quantity determined by the query, and cannot be replaced by K . Third, $\mathcal{U}_Q$ is already post-selection, so its counts must not be multiplied by α and β a second time. Restricting attention to $\mathcal{A}$ and $\mathcal{D}_J$ removes all three issues.

Proof of Proposition 4.1. For the first claim, note that by construction $\mathcal{A} \cap \mathcal{S}_K = \emptyset$ and $\mathcal{D}_J \subseteq \mathcal{S}_K$, so $|\mathcal{S}_K^{\text{LLM}} \cap \mathcal{V}^*| = |\mathcal{S}_K \cap \mathcal{V}^*| - |\mathcal{D}_J \cap \mathcal{V}^*| + |\mathcal{A} \cap \mathcal{V}^*|$. Dividing by m gives Equation (2). For the second, take expectations in Equation (2) at fixed m and use $|\mathcal{D}_J| = |\mathcal{A}| = J$, which gives $\mathbb{E}[\Delta \text{Rec}@K] = (\mathbb{E}|\mathcal{A} \cap \mathcal{V}^*| - \mathbb{E}|\mathcal{D}_J \cap \mathcal{V}^*|)/m = (\eta - \tau)\mathbb{E}|\mathcal{A}|/m$. This is positive if and only if $\eta > \tau$. $\square$

The proof requires neither label independence across functions nor any distributional assumption, since only linearity of expectation is used.

The same accounting gives the exact condition for full recovery. Writing $\mathcal{M} := \mathcal{V}^* \setminus \mathcal{S}_K$ for the targets missed by dense retrieval,

$$\text{Acc}@K(\mathcal{S}_K^{\text{LLM}}) = \mathbb{1}\{\mathcal{M} \subseteq \mathcal{A} \text{ and } \mathcal{D}_J \cap \mathcal{V}^* = \emptyset\}, \quad (4)$$

since the output contains all of $\mathcal{V}^*$ exactly when every baseline miss is admitted and no baseline target is displaced. $\text{Acc}@K$ is therefore not an average-precision quantity, and Equation (4) remains the exact requirement.

Nothing above assumes anything about the score distribution. The depth analysis in the rest of this

section does, so we restate the paper’s score assumption here; it is used in Remark A.4 and Proposition A.13, not in Proposition 4.1.

Assumption A.1 (Score distributions). There exist distributions F_1, F_0 on $\mathbb{R}$ with $s_Q(v) \sim F_1$ for $v \in \mathcal{V}^*$ and $s_Q(v) \sim F_0$ otherwise, such that F_1 first-order stochastically dominates F_0 , i.e. $F_1(t) \leq F_0(t)$ for all t (Shaked and Shanthikumar, 2007).

Roadmap for the rest of this section. Proposition 4.1 settles *whether* an exchange helps, but it is silent on the two questions that decide how SpIDER should be configured: when $\eta > \tau$ actually holds, and how far from a seed it is still worth exploring. Both need a model of how candidate quality varies with graph distance, and the remainder of this section builds one and then draws its consequences.

Section A.2.2 derives the model from a latent issue-bearing context that propagates along graph edges. It yields a shell-prevalence curve $p_h = \pi + \Delta\lambda^h$, and relates the seeding condition it requires to the paper’s existing score assumption, while being explicit that the two are not equivalent without a further condition on the ranking. Section A.2.3 decomposes the admission rates into the top- N gate, the Stage-2 selector, and capacity truncation, which is what connects α and β to η .

Section A.2.4 proves Theorem A.11, the destination of the analysis. It states that a shell is profitable exactly when its prevalence exceeds a threshold θ determined by τ and the admission rates, that deeper shells can expose strictly more edited functions even as per-candidate precision falls, and that the profitable shells form a prefix, so that whenever the threshold sits strictly above the background prevalence the useful exploration radius d^* is finite, and is language-dependent rather than universal. When it does not, every shell remains profitable and exploration is bounded by token budget instead.

The three remaining parts qualify that result. Section A.2.5 shows the prefix property survives when the rates vary with depth, so only the closed form for d^* depends on holding them constant. Section A.2.6 identifies the regime in which the Stage-2 filter is pivotal rather than redundant, and prices its token cost. Section A.2.7 states what would have to be measured to test the model on held-out repositories and what would falsify it, and Section A.2.8 bounds where the branching approximation applies. We stress throughout that this half of the analysis is a falsifiable hypothesis about our data, not some-

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

Symbol	Description
<i>Graph and codebase</i>	
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Code graph with nodes and edges
$\mathcal{V}$	Code entity nodes (functions, classes, files, directories)
$\mathcal{E}$	Edges of type <i>contains</i> , <i>invokes</i> , <i>imports</i> , <i>inherits</i>
n	Number of nodes in $\mathcal{V}$
$\text{dist}_{\mathcal{G}}(\cdot, \cdot)$	Shortest-path distance on $\mathcal{G}$
<i>Query and ground truth</i>	
Q	Issue description (query)
$\mathcal{V}^* \subseteq \mathcal{V}$	Ground-truth function nodes for Q
m	Number of ground-truth nodes, $m = \mathcal{V}^* $
<i>Scoring and dense retrieval</i>	
$\mathcal{F}(\cdot)$	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
<i>SpIDER hyperparameters and intermediate sets</i>	
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 <i>contains</i> 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
$\mathcal{S}_K^{\text{LLM}}(Q)$	Final SpIDER retrieval set of size K
<i>Evaluation and theory</i>	
$\text{Recall}@K$	$ \mathcal{V}^* \cap \mathcal{S}(Q) / \mathcal{V}^* $
$\text{Acc}@K$	$\mathbb{I}[\mathcal{V}^* \subseteq \mathcal{S}(Q)]$
$\text{MRR}@K$	Mean reciprocal rank of the first ground-truth node in $\mathcal{S}(Q)$
$\mathcal{P}$	New graph proposals, $(\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)) \setminus \mathcal{S}_K$
$\mathcal{R}$	Protected baseline items: seeds, plus accepted candidates already in $\mathcal{S}_K$
$\mathcal{A}$	Admitted set, the candidates that consume retrieval budget
$\mathcal{D}_J$	The $J = \mathcal{A} $ displaced baseline items
η	Admitted precision, $\mathbb{E} \mathcal{A} \cap \mathcal{V}^* /\mathbb{E} \mathcal{A} $ (Proposition 4.1)
τ	Frontier relevance of a displaced item, $\mathbb{E} \mathcal{D}_J \cap \mathcal{V}^* /\mathbb{E} \mathcal{D}_J $ (Proposition 4.1)
$h(v), n_h$	Distance from v to its nearest seed, and the number of functions at distance h
p_h	Target prevalence at distance h , $\pi + \Delta\lambda^h$ (Assumption A.8)
π, Δ, λ	Background rate, seeding gain, and per-hop persistence
a, c	Admission rates for edited and non-edited candidates (Assumption A.9)
b	Branching factor of the shell counts, $n_h = kb^{h-1}$
θ, d^*	Profitability threshold and useful exploration radius (Equation (17))

thing our experiments establish. Across issues with different m , per-issue gains must be weighted by $1/m$ before aggregation, which is exactly macro-averaged Recall@ K .

Remark A.2 (Why the base rate is the wrong comparison). The locality analysis of Section A.3 establishes the *necessary* precondition for SpIDER: edited functions are mutually reachable at small radius, so a seed landing on one of them can in

principle reach the others. Proposition 4.1 shows what must be added for sufficiency. Because the exchange is against a rank- K dense hit rather than a random function, the relevant comparison is $\eta > \tau$, and τ is far above the repository base rate; establishing $\pi_{\Gamma} \geq \Pr(u \in \mathcal{V}^*)$ therefore does not by itself imply improvement. Concretely, two edited functions among 100 siblings of one file are 2 hops apart, satisfying every locality statistic, yet their

neighborhood has prevalence $2/100$, which may lie below τ : reachability between targets does not count the non-edited functions in the same ball. The two analyses are complementary. Section A.3 shows the targets are reachable, and Proposition 4.1 states the precision they must clear once reached.

A.2.2 A latent-context model for Assumption A.8

Equating “the seed is edited” with “the seed is useful” is too restrictive: it cannot explain recovering a single edited function from a semantically related but *unedited* seed, which is the regime SpIDER targets. We instead posit a latent regional state.

Fix the implementation’s distance convention and a deterministic BFS parent for every reached node, yielding a rooted analysis tree whose geometry is independent of the labels. Let $Z_v \in \{0, 1\}$ indicate that v lies in an *issue-bearing structural context* (the right class, file, or local dependency region). Assume the tree-indexed conditional Markov property

$$\Pr(Z_{\text{child}}=1 \mid Z_{\text{parent}}=1) = \rho_1, \quad (5)$$

$$\Pr(Z_{\text{child}}=1 \mid Z_{\text{parent}}=0) = \rho_0, \quad (6)$$

with $0 \leq \rho_0 < \rho_1 \leq 1$ and $(\rho_0, \rho_1) \neq (0, 1)$. We further assume that the seed-selection event S is conditionally independent of a node’s context given its parent’s,

$$S \perp Z_{\text{child}} \mid Z_{\text{parent}}, \quad (7)$$

so that conditioning on S leaves the transition kernel in Equation (5) unchanged. Write

$$\lambda = \rho_1 - \rho_0 \in (0, 1), \quad \zeta = \frac{\rho_0}{1 - \rho_1 + \rho_0}, \quad (8)$$

the per-hop persistence of regional information and the stationary probability of the issue-bearing state. Edit labels are emissions,

$$\Pr(v \in \mathcal{V}^* \mid Z_v=1) = r_1, \quad (9)$$

$$\Pr(v \in \mathcal{V}^* \mid Z_v=0) = r_0, \quad r_1 > r_0, \quad (10)$$

and are conditionally independent of ancestry and of the seed-selection event S given Z_v . Let $q := \Pr(Z_{\text{seed}}=1 \mid S)$ be the context posterior induced by dense retrieval selecting the seed.

Lemma A.3 (Seeded spatial mixing). *Under Equations (5) and (9) with $q > \zeta$, every nonempty shell $h \geq 0$ satisfies*

$$p_h := \Pr(w \in \mathcal{V}^* \mid w \text{ uniform at distance } h, S) = \pi + \Delta \lambda^h$$

where $\pi = r_0 + (r_1 - r_0)\zeta$ and $\Delta = (r_1 - r_0)(q - \zeta) > 0$. Hence enrichment above the local background decays geometrically with graph distance.

Proof. Let $z_h = \Pr(Z=1 \mid h, S)$. By Equation (7) the transition probabilities are unchanged by conditioning on S , so averaging Equation (5) over the parent state gives the affine recurrence $z_{h+1} = \rho_0 + (\rho_1 - \rho_0)z_h$ with $z_0 = q$. Its fixed point is ζ from Equation (8), so $z_h - \zeta = \lambda(z_{h-1} - \zeta)$ and $z_h = \zeta + (q - \zeta)\lambda^h$. Applying Equation (9), $p_h = r_0 + (r_1 - r_0)z_h = \pi + \Delta\lambda^h$. $\square$

Equation (7) is the transition-side counterpart of the conditional independence already imposed on the emissions, and it is what lets a posterior formed at the seed be propagated with the unconditioned kernel. Substantively it says that dense retrieval’s decision to select this seed carries information about the surrounding region only through the seed’s own context state. It would fail if the ranker scored a node using signal that also predicts the context of nodes deeper in the tree without passing through the parent, for instance a repository-level or file-level feature shared by a whole subtree. As with the other conditions in this section, we regard it as a restriction on the analyzed regime rather than something the setup supplies.

Lemma A.3 yields exactly Assumption A.8. The observable content is the curve $p_h = \pi + \Delta\lambda^h$; the latent state is one mechanism that produces it, and the theory does not require Z_v to be a literal causal variable.

Remark A.4 (Interpreting the seeding condition). The requirement $q > \zeta$ has a direct operational reading. Since $\Pr(\text{seed} \in \mathcal{V}^* \mid S) = r_0 + (r_1 - r_0)q$ and the unconditional rate is $\pi = r_0 + (r_1 - r_0)\zeta$, we have

$$q > \zeta \iff \Pr(\text{seed} \in \mathcal{V}^* \mid S) > \Pr(v \in \mathcal{V}^*), \quad (12)$$

i.e. $q > \zeta$ holds exactly when a dense-selected seed is more likely to be edited than a uniformly random function. This is the property Assumption A.1 is intended to capture, but it does *not* follow from that assumption alone. First-order stochastic dominance constrains the two marginal score distributions, whereas a seed is the arg max of scores across nodes, so whether it is enriched depends on their joint behaviour. Under ties, or under strong dependence between target and non-target scores, a dominating marginal can still yield a top-ranked

item that is no more likely to be edited than a random function, and passing from the marginals to the selection event requires an additional exchangeability or conditional-independence condition. We therefore treat $q > \zeta$ the same way we treat the per-rank profile in Section A.2.5, as an empirical property rather than a theorem. It is also the cheapest of our quantities to check, since q is identified by seed precision, which is directly measurable from the runs we already report.

Remark A.5 (Unedited seeds). If one additionally observes that the seed is *not* edited, its context posterior drops to $q_0 = q(1 - r_1)/(q(1 - r_1) + (1 - q)(1 - r_0))$, and such a seed remains informative only when $q_0 > \zeta$: the dense evidence must outweigh the negative update from the seed’s own label. SpIDER does not observe that label at retrieval time, so its operative ex-ante parameter is q .

Proposition A.6 (Mixing over multiple seeds). *Let $\mathcal{C}_Q$ contain C seeds with context posteriors $q_1, \dots, q_C$, assign each reached function to its nearest seed, and let $w_{c,h}$ be the share of shell- h functions assigned to seed c , determined by tree geometry alone. Then*

$$\begin{aligned} p_h &= \pi + \bar{\Delta}_h \lambda^h, & \bar{\Delta}_h &= (r_1 - r_0)(\bar{q}_h - \zeta), \\ \bar{q}_h &= \sum_c w_{c,h} q_c. \end{aligned} \quad (13)$$

If in addition the shares are depth-invariant over the analyzed range, $w_{c,h} = w_c$ for all h , then $\bar{\Delta}_h \equiv \bar{\Delta}$ and the curve takes the single-seed form assumed in Assumption A.8.

Proof. By Lemma A.3, $p_h(q) = \pi + (r_1 - r_0)(q - \zeta)\lambda^h$ is affine in q . Averaging over the shell- h seed assignment therefore replaces q by $\bar{q}_h$, giving Equation (13). Depth-invariant shares make $\bar{q}_h$ and hence $\bar{\Delta}_h$ constant in h . $\square$

Remark A.7 (Depth-invariance is an extra condition). Depth-invariance of the shares does *not* follow from disjointness of the seed neighborhoods. Disjointness removes double counting, but the shells around different seeds can still grow at different rates, so a seed sitting in a large class may supply most of the functions at $h=2$ while a seed in a small one supplies more at $h=4$. In that case $\bar{q}_h$ drifts with depth and there is no single $\bar{\Delta}$. Two consequences follow. First, Assumption A.8 should be read, under $C > 1$, as holding exactly only when the shares are depth-invariant, and otherwise as

an approximation whose adequacy is part of what the protocol in Section A.2.7 tests, since $w_{c,h}$ is directly countable from the graph. Second, the prefix property in Theorem A.11 relies on p_h decreasing in h , which Equation (13) no longer gives automatically. A sufficient condition is that $\bar{\Delta}_h$ be nonincreasing, since the product of a nonincreasing positive sequence with the strictly decreasing λ^h is strictly decreasing. This is the multi-seed analogue of the monotonicity condition on θ_h in Section A.2.5, and we treat it the same way, as a checkable condition rather than a consequence.

Subject to that condition, Proposition A.6 is why the single-seed analysis carries over to the deployed $C=5$ configuration, with an effective seeding gain $\bar{\Delta}$. Assigning each node to its nearest seed also discards evidence when neighborhoods overlap, and the direction of that bias is known: a node reachable from a second, farther seed c' receives additional positive evidence whenever $q_{c'} > \zeta$, and discarding it understates $\bar{q}$, hence $\bar{\Delta}$, hence p_h . The approximation therefore biases d^* downward and under-promises reach, so a cutoff derived under it is conservative. Adding seeds raises $\bar{q}$ only if the new seeds are themselves well-placed, which is one mechanism for the saturation in C observed in Section 5.3.

A.2.3 Graph volume and admission rates

Let H_h be the distinct function nodes at distance h and $n_h = |H_h|$. Theorem A.11 uses these observed counts and does *not* require geometric growth; the regular-growth form $n_h = kb^{h-1}$ is invoked only to interpret target volume. In pure contains graphs, function-to-function distances are typically even (siblings are 2 hops apart through their container), so odd shells may be empty; the theorem is stated over *nonempty* shells and the last beneficial one is $\bar{d}^* = \max\{h \leq d^* : n_h > 0\}$.

Decompose the admission pipeline by ground-truth label, at depth h :

$$\begin{aligned} t &= \Pr(v \in \mathcal{S}_N \setminus \mathcal{S}_K \mid v \in \mathcal{V}^*), \\ f &= \Pr(v \in \mathcal{S}_N \setminus \mathcal{S}_K \mid v \notin \mathcal{V}^*), \\ \alpha &= \Pr(\mathcal{L}(v)=1 \mid v \in \mathcal{V}^*, \cdot), \\ \beta &= \Pr(\mathcal{L}(v)=1 \mid v \notin \mathcal{V}^*, \cdot), \\ \gamma_1 &= \Pr(v \in \mathcal{A} \mid \mathcal{L}(v)=1, v \in \mathcal{V}^*), \\ \gamma_0 &= \Pr(v \in \mathcal{A} \mid \mathcal{L}(v)=1, v \notin \mathcal{V}^*), \end{aligned} \quad (14)$$

so that the effective rates of Assumption A.9 are $a = t\alpha\gamma_1$ and $c = f\beta\gamma_0$, with γ_1, γ_0 absorbing priority, deduplication and capacity truncation.

Depth comparisons use a nested, shell-first policy: increasing d from $h - 1$ to h leaves earlier admissions unchanged and causes only additional baseline-prefix evictions. Once capacity binds the effective rates are zero, a saturated shell has zero incremental gain, and the depth statement no longer applies.

Let $h(v)$ be the distance from v to its nearest seed and n_h the number of distinct functions in shell h .

Assumption A.8 (Seeded spatial decay). The shell- h target prevalence is $p_h := \Pr(v \in \mathcal{V}^* \mid h(v) = h) = \pi + \Delta\lambda^h$, for a background rate π , a seeding gain $\Delta > 0$ and a per-hop persistence $\lambda \in (0, 1)$. Lemma A.3 derives this form.

Assumption A.9 (Admission and exchange rates). Over the analyzed depth range, a shell- h candidate is admitted with probability $a > 0$ if edited and $c > 0$ if not (composing top- N retention, the LLM decision, deduplication and capacity). Moreover the frontier relevance is constant *conditional on the number of evictions*,

$$\mathbb{E}[|\mathcal{D}_J \cap \mathcal{V}^*| \mid J] = \tau J, \quad \tau \in (0, 1). \quad (15)$$

Remark A.10 (Why Equation (15) is stronger than the definition of τ). In Proposition 4.1, τ is a ratio of marginal expectations, $\mathbb{E}[|\mathcal{D}_J \cap \mathcal{V}^*|]/\mathbb{E}[|\mathcal{D}_J|]$, and that identity holds with no assumption because the proposition aggregates over the whole exchange. Theorem A.11 instead charges an eviction cost to each shell separately, which requires the per-eviction relevance to be unaffected by how many candidates a query happens to admit. Without Equation (15) the two can differ: if J and the relevance of the displaced frontier are correlated, for instance because queries that admit many neighbors are also queries whose rank- K items are unusually relevant, then $\mathbb{E}[\tau_J J] \neq \tau \mathbb{E}[J]$ and the shell decomposition below fails. We therefore state the conditional form explicitly rather than deriving it from the definition. Like the constancy of a and c , it is an assumption about our operating range and not a consequence of the setup.

Theorem A.11 (Seeded spatial-completion law). *Under Assumptions A.8 and A.9, shell h contributes $n_h a p_h$ targets and $n_h c(1 - p_h)$ distractors at precision $\eta_h = a p_h / (a p_h + c(1 - p_h))$, and its expected contribution to the recall numerator is $G_h = n_h[(1 - \tau)a + \tau c]p_h - \tau c$. Then: (i) with $\theta := \tau c / ((1 - \tau)a + \tau c)$,*

$$G_h > 0 \iff \eta_h > \tau \iff p_h > \theta; \quad (16)$$

(ii) the shell's excess target mass is $n_h \Delta \lambda^h$, which under regular growth $n_h = k b^{h-1}$ equals $k \Delta \lambda (b \lambda)^{h-1}$, so if $b \lambda > 1$ deeper shells expose strictly more edited functions even though η_h strictly decreases; and (iii) beneficial shells form a prefix, so when $\pi < \theta < p_1$ the expected recall numerator is maximized at the useful radius

$$d^* = \lceil \log((\theta - \pi)/\Delta) / \log \lambda \rceil - 1. \quad (17)$$

A.2.4 Proof of Theorem A.11

Proof. (Counts and precision.) By linearity of expectation, and again without requiring independence within a shell, the expected admitted target and distractor counts from shell h are $T_h = n_h a p_h$ and $F_h = n_h c(1 - p_h)$, and dividing target admissions by all admissions gives $\eta_h = a p_h / (a p_h + c(1 - p_h))$.

(Threshold.) By Equation (15), the expected relevant loss accompanying these admissions is $\tau(T_h + F_h)$, so

$$\begin{aligned} G_h &= T_h - \tau(T_h + F_h) \\ &= n_h [(1 - \tau)a p_h - \tau c(1 - p_h)] \\ &= n_h [(1 - \tau)a + \tau c]p_h - \tau c. \end{aligned} \quad (18)$$

Since $(1 - \tau)a + \tau c > 0$, $G_h > 0 \iff p_h > \theta$ with θ as in Equation (16). Dividing $\eta_h > \tau$ through by $c(1 - p_h) > 0$ yields $(a/c)p_h/(1 - p_h) > \tau/(1 - \tau)$, which rearranges to the same inequality, giving the equivalence $G_h > 0 \iff \eta_h > \tau \iff p_h > \theta$.

(Reach vs. precision.) Substituting $p_h - \pi = \Delta \lambda^h$ gives excess mass $n_h \Delta \lambda^h$, and $n_h = k b^{h-1}$ gives $k \Delta \lambda (b \lambda)^{h-1}$, which increases in h iff $b \lambda > 1$. Meanwhile η_h is strictly increasing in p_h and p_h is strictly decreasing, so η_h strictly decreases regardless of b .

(Useful radius.) As $0 < \lambda < 1$ and $\Delta > 0$, p_h decreases strictly to π , so $\{h : p_h > \theta\}$ is a prefix. In the interior case $\pi < \theta < p_1$,

$$\begin{aligned} p_h > \theta &\iff \lambda^h > \frac{\theta - \pi}{\Delta} \\ &\iff h < \frac{\log((\theta - \pi)/\Delta)}{\log \lambda}, \end{aligned} \quad (19)$$

the inequality reversing because $\log \lambda < 0$. The largest integer strictly below the right-hand side is d^* of Equation (17), and $\theta < p_1$ forces $d^* \geq 1$. Shell increments are positive up to d^* and nonpositive after, so the cumulative expectation is maximized there; if the first excluded shell has $G_h = 0$ then $d^* + 1$ attains the same value but no larger. $\square$

Corollary A.12 (Accuracy is harder than recall). *If the baseline is already complete ($\mathcal{V}^* \subseteq \mathcal{S}_K$), then $\mathbb{E}[\text{Acc}@K(\mathcal{S}_K^{\text{LLM}})] - \mathbb{E}[\text{Acc}@K(\mathcal{S}_K)] = -\Pr(\mathcal{D}_J \cap \mathcal{V}^* \neq \emptyset) \leq 0$, with strict inequality whenever $\tau > 0$ and $J \geq 1$.*

Proof of Corollary A.12. If $\mathcal{V}^* \subseteq \mathcal{S}_K$ then $\mathcal{M} = \emptyset$, so by Equation (4) $\text{Acc}@K(\mathcal{S}_K^{\text{LLM}}) = \mathbb{1}\{\mathcal{D}_J \cap \mathcal{V}^* = \emptyset\}$ while $\text{Acc}@K(\mathcal{S}_K) = 1$. Taking expectations gives the claim; the inequality is strict when some eviction can be a target. $\square$

Corollary A.12 is a *stratified* prediction and we are careful not to overstate it. It says that on issues whose baseline already contained every edited function, any insertion can only lose $\text{Acc}@K$, through a single bad eviction. It does *not* predict that aggregate $\text{Acc}@K$ falls, and aggregate $\text{Acc}@K$ is not a valid test of it: gains on incomplete-baseline issues can dominate, and in our main results $\text{Acc}@K$ rises alongside $\text{Recall}@K$ in almost every cell. The corresponding test is to restrict to instances where the dense baseline already attains $\text{Acc}@K = 1$ and measure the eviction rate there; we have not run it, and report Corollary A.12 as a prediction rather than a confirmed finding.

A.2.5 Relaxing the constant-rate assumption

The closed form Equation (17) comes from holding a, c, τ fixed. If they vary with depth, define $\theta_h = \tau_h c_h / ((1 - \tau_h)a_h + \tau_h c_h)$.

Proposition A.13 (Robustness of the prefix property). *Write $\theta_h = \tau_h / ((1 - \tau_h)x_h + \tau_h)$ with $x_h = a_h/c_h$. Then θ_h is increasing in τ_h and decreasing in x_h . Consequently, if (τ_h) is nondecreasing and (x_h) is nonincreasing in h , then θ_h is nondecreasing and $p_h - \theta_h$ is strictly decreasing, so the beneficial shells still form a prefix. That prefix is finite if and only if $\theta_h > \pi$ for some h , equivalently $\liminf_h \theta_h > \pi$. If instead $\theta_h \leq \pi$ at every depth, then $p_h > \pi \geq \theta_h$ for all h and every shell remains beneficial, so no finite radius exists.*

Proof. Write $\theta_h = (1 + (1 - \tau_h)x_h/\tau_h)^{-1}$. The map $\tau \mapsto (1 - \tau)/\tau$ is decreasing on $(0, 1)$, so θ_h increases in τ_h ; it plainly decreases in x_h . Under the stated monotonicity θ_h is nondecreasing while p_h strictly decreases, so $p_h - \theta_h$ is strictly decreasing and changes sign at most once, from positive to negative, which gives the prefix property. For finiteness, recall $p_h \downarrow \pi$ strictly. If $\theta_H > \pi$ for some H then $\theta_h \geq \theta_H > \pi$ for all $h \geq H$ by monotonicity, while $p_h \rightarrow \pi$, so $p_h < \theta_h$ for all large h and the

prefix terminates. Conversely if $\theta_h \leq \pi$ for every h then $p_h > \pi \geq \theta_h$ at every depth and the prefix is the whole sequence. $\square$

Both hypotheses are the *natural* directions of violation, though we assume rather than derive them. Under the shell-first policy, admitting more candidates evicts progressively further up the dense ranking, so τ_h is nondecreasing whenever the baseline’s per-rank target probability is nonincreasing in rank. We note that this does *not* follow from Assumption A.1 alone: first-order stochastic dominance of the score distributions constrains the marginals, not the per-rank posterior, and an additional exchangeability condition is required to pass from one to the other. The per-rank profile is directly estimable from the existing dense rankings at no additional inference cost, and we regard it as an empirical check rather than a theorem. Likewise deeper candidates are less semantically similar to Q , so both the top- N gate and the LLM plausibly discriminate less well and $x_h = a_h/c_h$ degrades. Proposition A.13 therefore says the qualitative conclusion survives the simplification, and only the closed form Equation (17) depends on it. The finiteness clause is the varying-rate analogue of the trichotomy in Theorem A.11: just as a constant $\theta \leq \pi$ makes every shell profitable there, a sequence with $\theta_h \leq \pi$ throughout leaves the useful radius unbounded here, and exploration is then limited by token budget rather than by precision. Without monotone θ_h , a later shell can be useful after a harmful one and no single-cutoff theorem holds.

A.2.6 LLM pivotality and token cost

Corollary A.14 (When is the LLM pivotal?). *Compare a graph-only policy with rates (t, f) against the LLM-filtered policy with rates $(t\alpha, f\beta)$ at a common, count-matched frontier τ . If $\alpha > \beta > 0$ then $\theta_L < \theta_G$, and shell h is graph-sufficient if $p_h > \theta_G$, LLM-pivotal if $\theta_L < p_h \leq \theta_G$, and unprofitable for both if $p_h \leq \theta_L$.*

Proof of Corollary A.14. With nonbinding capacity the graph-only policy has effective rates $(a_G, c_G) = (t, f)$ and the filtered policy $(a_L, c_L) = (t\alpha, f\beta)$. Writing $\theta = \tau / ((1 - \tau)x + \tau)$ with $x = a/c$, the map $x \mapsto \theta$ is strictly decreasing for $x > 0$. The graph-only ratio is t/f and the filtered ratio is $(t/f)(\alpha/\beta) > t/f$ since $\alpha > \beta$, hence $\theta_L < \theta_G$. The three regimes follow by applying Equation (16) with each threshold. $\square$

Two caveats bound this result. It does not establish that the evaluated LLM *is* pivotal: that requires measuring α, β and running a count-matched graph-only ablation, since an unfiltered policy admits more candidates and would otherwise face a different frontier, which is why Corollary A.14 fixes a common τ . Nor does it make an LLM uniquely necessary: any calibrated query-candidate interaction model with the same likelihood ratio α/β plays the same role. Finally, if LLM inference is priced at κ utility per raw candidate, the numerator of θ becomes $\tau c + \kappa$, so a better likelihood ratio need not imply better total utility.

A.2.7 What to measure, and what would falsify the model

Remark A.15 (What the existing ablations can and cannot show). Theorem A.11 predicts a finite useful radius in its interior case. Our present ablations cannot confirm or refute one, and we state this explicitly rather than claim support. The depth and top- N sweeps (Tables 9 and 10) are single-seed runs reported to two decimals, with no confidence intervals; the differences that would bear on Equation (16), namely the flat Java depth column, the TypeScript plateau from $d=6$ to $d=8$, and the Python top- N maximum near $N=500$, are all of size 0.01. That is roughly one sixth of the 95% interval half-width we report for the main results and below the seed-to-seed drift we document for the Stage-2 selector, which runs at nonzero temperature. Pairing across settings would tighten the interval on a *difference*, but it does not remove that selector noise, since each run draws fresh LLM decisions. We therefore treat these sweeps as compatible with either a cutoff beyond the measured range or saturation within it, and as distinguishing neither.

One measurement in this family does clear the noise floor, because it is deterministic given the candidate set: inference *cost*. Input tokens per query in Table 8 grow $3.6\text{K} \rightarrow 29.7\text{K}$ (Python) from $d=4$ to $d=8$, an $8.2\times$ increase. Pricing inference at κ per raw candidate replaces the numerator of θ by $\tau c + \kappa$ (Section A.2.6), which moves the useful radius inward; this is what makes the deployed $d=4$ a budget choice rather than a measured performance optimum. We emphasise that these token counts do *not* estimate the shell sizes n_h . They are recorded after Stage-1 gating, and the depth ablation runs at $N=100$, so the evaluated set

$\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N$ is capped at 100 candidates irrespective of d ; the curve therefore reflects a bounded pool filling up, not raw shell volume. Consistently, the per-two-hop growth ratios are not constant and Java’s decreases from $d=2$ to $d=4$. Estimating n_h requires the before-gating counts specified in Section A.2.7.

The theory is useful only as a held-out curve prediction. Its observables are n_h (distinct shell functions before dense-rank gating), p_h (count-weighted target prevalence before top- N), t_h, f_h ($\mathcal{S}_N \setminus \mathcal{S}_K$ retention by label), α_h, β_h (raw LLM decisions by label), γ_{1h}, γ_{0h} (post-LLM survival), and τ_j (the ground-truth rate of the actual j -th eviction). Across variable-size shells, count weighting is essential: $n_h = \mathbb{E}N_h$ and $p_h = \mathbb{E}R_h/\mathbb{E}N_h$ for raw and target shell counts N_h, R_h ; an unweighted mean of per-query shell precisions does not satisfy $\mathbb{E}R_h = n_h p_h$. For macro Recall@ K across varying m , use $n_h^{\text{macro}} = \mathbb{E}[N_h/m]$ and $p_h^{\text{macro}} = \mathbb{E}[R_h/m]/\mathbb{E}[N_h/m]$.

The protocol is to fit π, Δ, λ on shallow shells and k, b on distinct non-backtracking shell counts, freeze them, and *predict* deeper-shell prevalence, the sign of G_h , and d^* on held-out repositories, rather than estimating a free prevalence at every depth. Repository-held-out estimation is required because functions within a repository, and candidates within one LLM batch, are not independent samples. The model is falsified, or must be replaced by a relation-specific one, if: (i) shell prevalence is not approximately geometric after controlling for dense rank, node type and degree; (ii) candidate quality has multiple peaks across depth, so beneficial shells are not a prefix; (iii) degree-preserving graph rewiring does not reduce seeded enrichment; (iv) a count-matched graph-only policy matches the LLM policy in the purported pivotal region; (v) actual relevant evictions do not track the frontier model; or (vi) the predicted cutoff does not transfer from shallow shells to held-out deeper shells and repositories.

These quantities are complementary to, not substitutes for, Table 6: target-to-target distances establish reachability, whereas Theorem A.11 additionally requires all-candidate shell prevalence and the realized exchange frontier (Remark A.2). Likewise the depth and top- N ablations in Section 5.3 are outcomes of the full pipeline, so while they are consistent with the model (Remark A.15) they do not identify which term of G_h changed.

A.2.8 Scope

The regular-branching model is deliberately small enough to make a falsifiable prediction. Real containment graphs mix directories, files, classes and functions, admit multiple paths under the chosen distance convention, and have overlapping seed neighborhoods (Proposition A.6). A multitype extension replaces $b\lambda$ by the spectral radius of a type-specific branching–persistence matrix; it is warranted only if the single-decay model fails and enough data exist to estimate the extra parameters. The branching approximation must also stop before graph collisions, finite-target depletion, or insertion capacity invalidate geometric shell growth.

We note that $b\lambda$ governs growth of the *expected number* of excess targets and is not the Kesten–Stigum root-reconstruction threshold $b\lambda^2$ for broadcasting on trees. The theorem concerns expected hit count, hence Recall@ K at fixed m ; it does not analyze MRR, because Equation (1) does not fully determine the final ordering.

The hidden-context model is related to broadcasting on trees, where a noisy state propagates through a branching structure (Evans et al., 2000), and the first-moment $b\lambda$ behaviour has precedent in Ising and percolation analysis on trees (Lyons, 1989). The use here is narrower: the relevant question is not reconstruction of the root state, but whether a decaying spatial signal can support a profitable fixed-budget exchange. The filtered-precision comparison instantiates posterior ranking under additive retrieval utility (Robertson, 1977), and triggering local exploration from an initially promising unit resembles adaptive cluster sampling (Thompson, 1990); adaptive corpus-graph re-ranking (MacAvaney et al., 2022) is the closest system analogue. The individual mathematical ingredients are standard; the contribution is their coupling to SpIDER’s spatial mixing, filtering, and fixed-budget eviction.

A.3 Empirical Support for Graph Locality

Proposition 4.1 needs no locality assumption, since it is an identity about the exchange. Graph locality enters one step later, through Assumption A.8, which posits that target prevalence decays with distance from a seed as $p_h = \pi + \Delta\lambda^h$ with $\Delta > 0$. That form implies $p_h > \pi$ at every finite h , which is exactly the requirement that relevance be non-negatively correlated with graph proximity. Locality is therefore necessary for the depth analysis to

have any content, and, as Remark A.2 notes, necessary but not sufficient for a profitable exchange. To check that it is not merely a convenient modeling choice, 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 $d \leq 4$ 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 4.1. 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 the decay form assumed in Assumption A.8: 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 precondition that the depth analysis needs therefore holds in practice. It does not by itself imply that expansion is profitable, which by Proposition 4.1 additionally requires $\eta > \tau$.

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. The locality that Assumption A.8 presumes 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 this locality 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, which is the seeding condition $q > \zeta$ of Remark A.4, but it 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 alone 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.4 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.5 Stage-2 LLM Filter Prompt

The Stage-2 selector is driven by the system and user prompts reproduced below. The prompt is deliberately simple, to limit prompt-sensitivity, and is applied independently to each of the C seed centers.

System prompt.

```
You are a code localization agent. You are examining the
graph neighborhood of a {node_type} to find related
{node_type}s that MUST be modified to fix a GitHub issue.
You will see:
- The center {node_type}
- Its graph neighbors (connected via {edge_phrase})
BE VERY RESTRICTIVE. Only select neighbors where you have
HIGH CONFIDENCE that the code MUST be modified to fix the
issue. When in doubt, do NOT select. It is better to
return an empty list than to include a neighbor that
probably does not need changes.
Do NOT include the center itself. Do NOT include neighbors
that are merely related, called by, or similar to code in
the issue - only include neighbors whose own code must
change.
Respond with ONLY a JSON object:
{"selected": ["node_id_1", "node_id_2", ...]}
If no neighbor clearly needs changes, respond with:
{"selected": []}
```

User prompt.

```
## GitHub Issue
{PROBLEM_STATEMENT}

## Center function (1/5)

### [center_i] {CENTER_NODE_ID}
```

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
<i>Instance / ground-truth statistics</i>				
# 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
<i>Graph locality (multi-func instances)</i>				
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
<i>Dense retrieval coverage (top-K)</i>				
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

```
{CENTER_CODE}

## Neighbors

### [nbr_i] {NEIGHBOR_i_ID}
{NEIGHBOR_i_CODE}

Select ONLY the functions that you are highly confident
MUST be modified to fix the issue.
```

Placeholders. `node_type` is the retrieval granularity (function for all function-level results in this paper) and `edge_phrase` names the edge types traversed for that run. `PROBLEM_STATEMENT` is the raw issue title and body. The center header is numbered $(i+1)/C$ over the C seed centers, shown above for the default $C=5$. `CENTER_CODE` and each `NEIGHBOR_i_CODE` carry the node’s source, truncated to 1500 characters with a trailing ... [truncated] marker when longer. Neighbors are enumerated with the graph node ids that the model echoes back in its JSON response. Doubled braces are Python format-string escapes; the model receives single braces. The template contains no in-context examples, so the selector is used zero-shot.

A.6 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). Ground-truth labels are inherited from those upstream sources, and construction is deterministic given the repository state, using no manual annotation. We release the dataset, the graph-construction code, and the Stage-2 LLM-filter prompt through the links on the first page. Because SpIDER-Bench embeds source code drawn from many upstream repositories, it carries no single license. Individual code segments remain under the licenses of the projects they were adapted from, which include MIT, Apache 2.0, BSD and GPL, and the release ships a notice file identifying the third-party sources. Users must comply with the license attached to each segment they use.

A.7 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

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

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

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

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.8 Additional Experiments

Downstream code generation with a tool-equipped agent. Figure 3 reports the complementary setting to Table 4, in which the protocol is *not* hands-tied: mini-swe-agent (Yang et al., 2024) with Claude Sonnet 4.5 receives the retrieved function paths but retains full grep and file-navigation tools, so it can partially compensate for a poor retrieval list on its own. In this setting SpIDER improves the pass rate on SWEbench-Verified by +0.46 percentage points (pp) at $K=5$ and +2.74 pp at $K=10$, corresponding to 2 and 12 additional resolved instances. Two caveats apply. First, these are single runs, and the $K=5$ margin of 2 instances is within run-to-run variation, so we do not read a positive effect into it. Second, and more fundamentally, an agent that can already locate code independently compresses the gap between any two retrieval lists, so this comparison measures only what SpIDER contributes *on top of* a competent search capability rather than the contribution of retrieval quality itself. Table 4 isolates the latter by removing the agent’s independent search, which is why we report it as the primary downstream result.

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

corresponding class as a ground truth class node),

Table 8: Token usage for the BFS depth (d) ablation from Table 9 ($K = 20$, $N = 100$, $C = 3$). 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

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

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	0.44	0.36	0.54
	+ LLM reranker			
	CodeSAGE-Small ZS + SpIDER	0.23	0.18	0.31
	+ LLM reranker			
	CodeSAGE-Small-PEFT + SpIDER	0.32	0.25	0.42
	+ LLM reranker			
	BM25+ SpISR	0.30	0.23	0.38

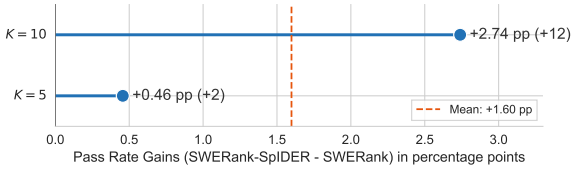


Figure 3: Pass rate improvement of SWERankEmbed-Small-SpIDER over SWERank on SWE-bench Verified with $C = 5, d = 4, N = 500$.

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.9 Edge-Type Ablation: Statistical Significance

Table 3 in the main text reports point estimates for each exploration edge type. Because SpIDER and DER are evaluated on identical instances and their per-instance scores are highly correlated, we assess significance with paired per-instance tests rather than by comparing the marginal confidence intervals of Table 13, which have substantially less power. For each language and edge configuration we form the per-instance differences Δ_i between the SpIDER and DER score on instance i , and report (i) a paired-bootstrap p -value, $\Pr(\bar{\Delta} \leq 0)$ over $B = 10,000$ resamples of $\{\Delta_i\}$, and (ii) a two-sided paired-permutation (sign-flip) test over 10,000 permutations. Table 20 reports both, for Recall@20 and Acc@20.

The two tests agree in every cell. All structural-edge gains on Recall@20 are signifi-

cant at $p < 0.01$ in all four languages, and at $p < 0.001$ for Java and TypeScript under *invokes* or *contains+invokes*. On Acc@20 the only non-significant structural cell is Java with *contains* alone, which becomes significant once *invokes* edges are added (permutation $p = 0.0224$); this is consistent with the locality analysis of Table 6, where Java combines the highest invoke-connectivity with the lowest dense top-100 coverage. *imports* and *inherits* are uniformly non-significant, and Proposition 4.1 predicts these two nulls for structurally different reasons. For *inherits*, no edge in our schema is incident to a function node in any language, so the explored shell contains no functions at all, $n_h = 0$, and no exchange is offered. The configuration returns exactly the DER set, which is what Table 3 reports. For *imports*, edges do reach function nodes but their prevalence sits at the background rate outside Python, so η falls below the frontier τ and the condition genuinely fails. A framework that could not separate these two cases would be vacuous, whereas here each predicts a distinct observable, an identical-to-DER result in one case and a non-significant difference in the other.

What the Stage-2 selector contributes. Disabling Stage 2 does not yield a comparable operating point: every node in $\Gamma_d(\mathcal{C}_Q) \cap \mathcal{S}_N(Q)$ is promoted, the retrieved set grows to hundreds of candidates, and the fixed budget K is abandoned. The resulting recall is therefore an *upper bound* on what any selection rule over the same neighborhood could achieve, and the useful question is how much of that ceiling SpIDER captures while still returning only K functions. Table 21 reports both. SpIDER recovers 79–89% of the unfiltered neighborhood’s reachable recall at $K = 20$, so Stage 2 discards the large majority of surfaced nodes while preserving

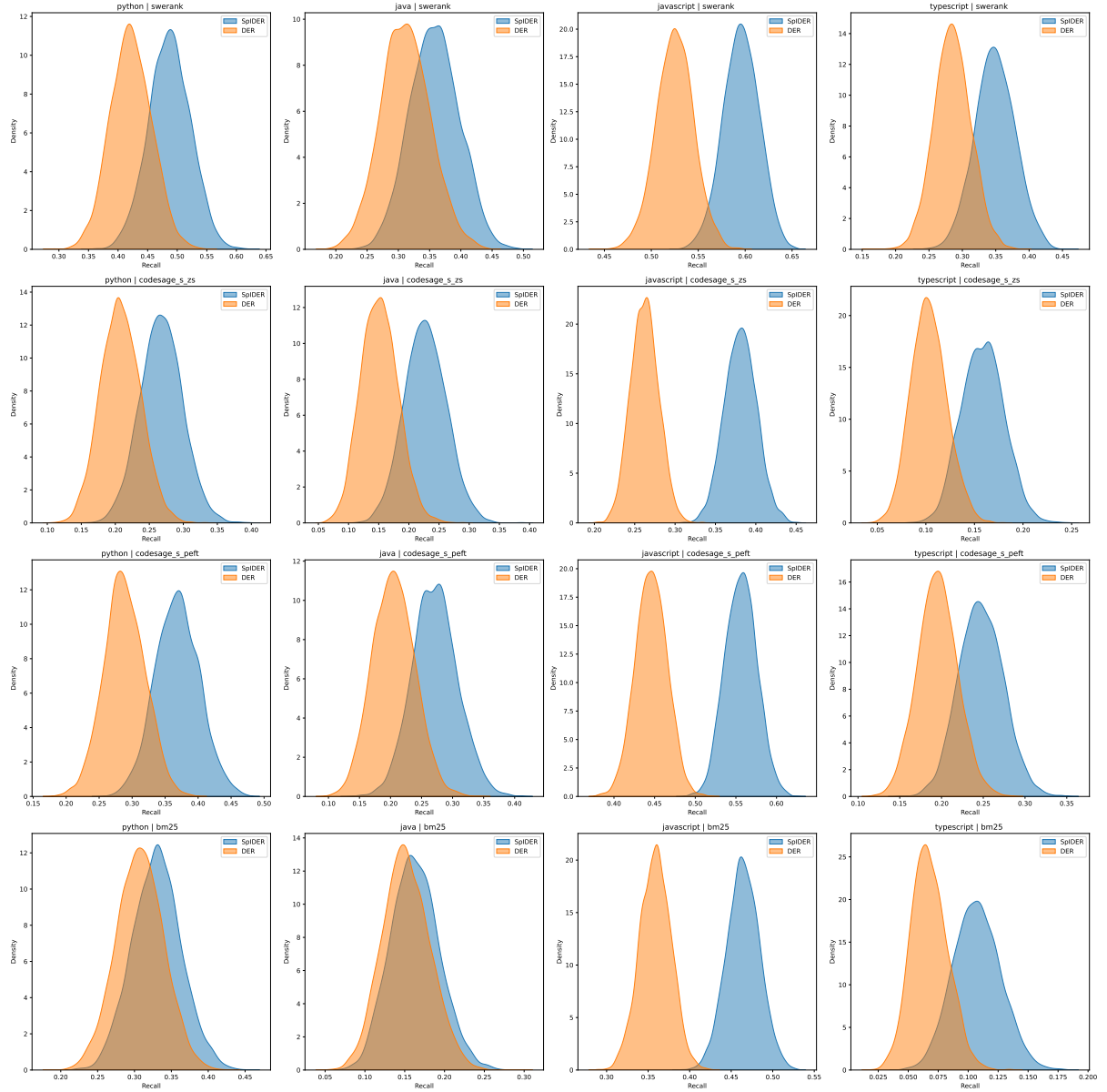


Figure 4: KDE (Kernel Density Estimate) plots with bootstrapped results of SWE-PolyBench benchmark for Recall@20 performance across various retrieval methods.

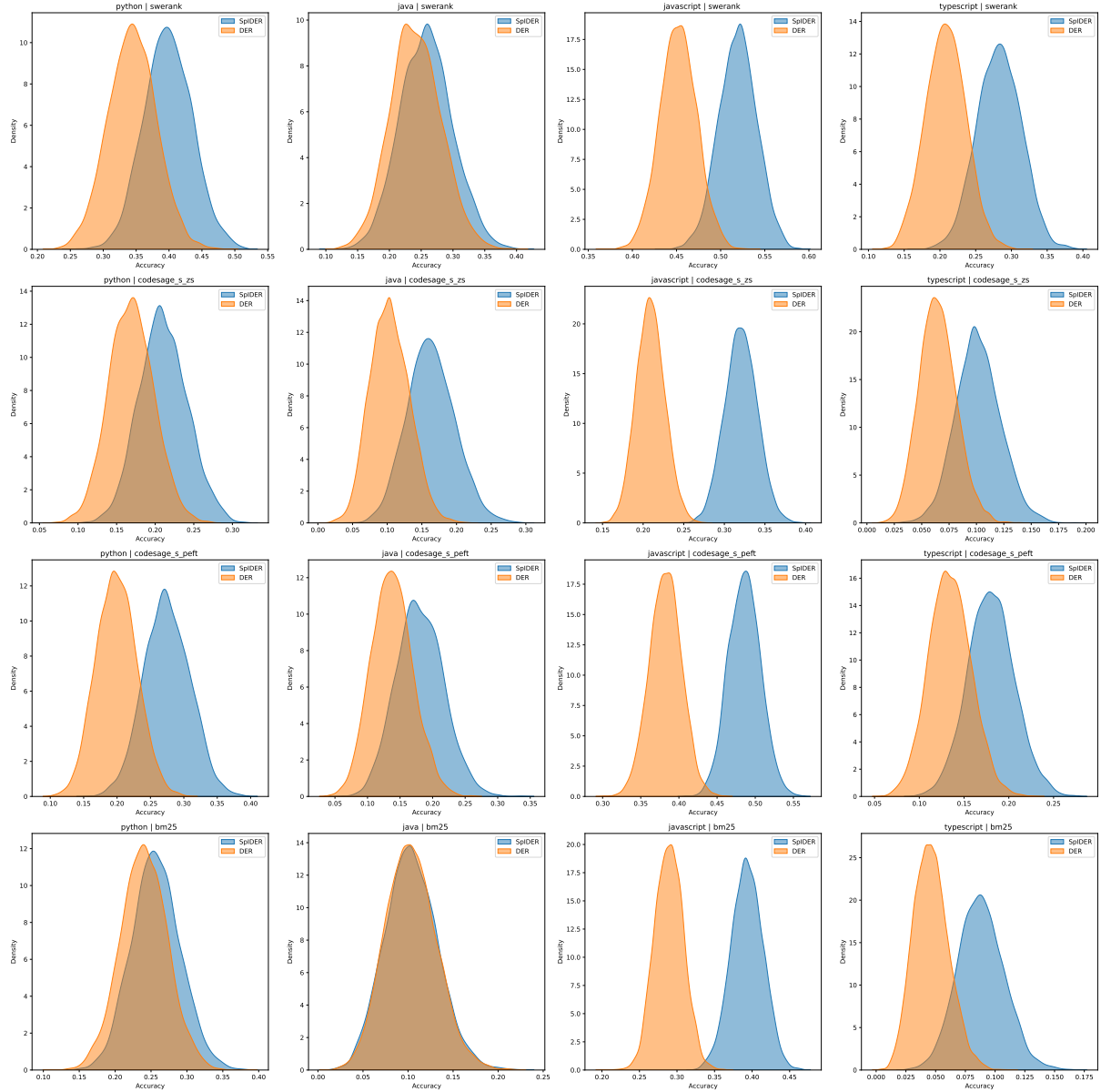


Figure 5: KDE (Kernel Density Estimate) plots with bootstrapped results of SWE-PolyBench benchmark for Acc@20 performance across various retrieval methods.

Table 13: SWE-PolyBench retrieval performance with 95% confidence intervals ($\mu \pm \text{CI}$) 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	0.49	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

SWERank-Small ZS vs SWERank-Small ZS + SpIDER

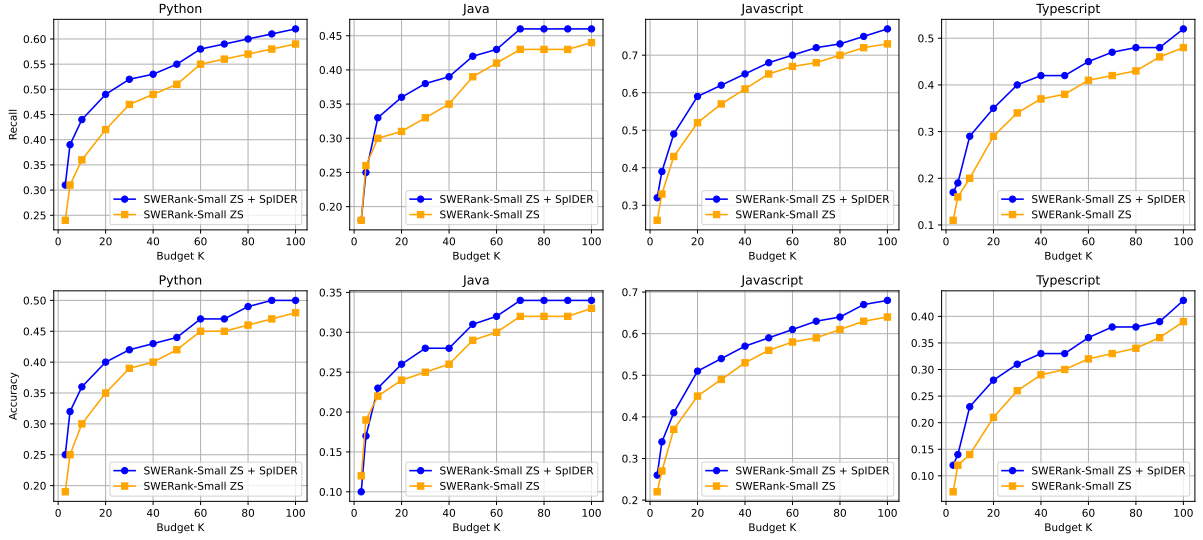


Figure 6: SpIDER vs DER performance at various values of K on SWE-PolyBench benchmark for SweRankEmbed-Small ZS embedding model, $N = 500$, $C = 5$, $d = 4$.

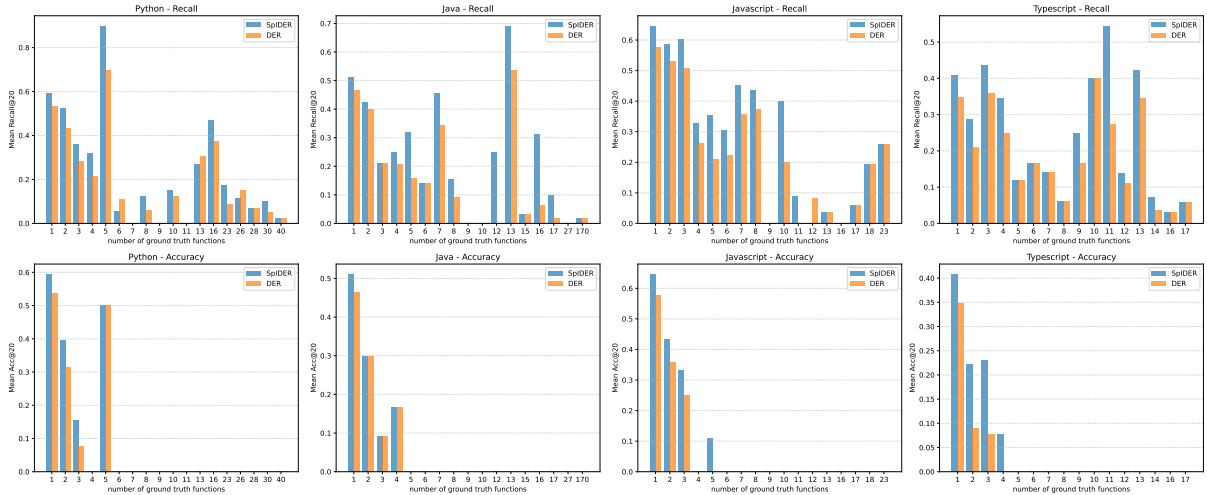


Figure 7: Comparing retrieval Recall@20 and Accuracy vs. number of edits for DER vs SpIDER on SweRankEmbed-Small ZS using SWE-PolyBench across various languages for $N = 500$, $C = 5$, $d = 4$. Top row: Recall@20. Bottom row: Accuracy.

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: **Paired significance of SpIDER over DER on SWE-PolyBench**, by exploration edge type, with SweRankEmbed-Small at $K = 20$, $N = 500$, $C = 5$. Δ is the mean per-instance difference; “bootP” is the paired-bootstrap $\Pr(\bar{\Delta} \leq 0)$ over $B = 10,000$ resamples; “perm” is the two-sided paired-permutation p -value over 10,000 sign-flips. $***p < 0.001$, $**p < 0.01$, $*p < 0.05$; unmarked entries are not significant. Δ is computed on unrounded per-instance scores and may therefore differ by up to 0.01 from the difference of the rounded means in Table 3. *inherits* is marked N/A outside Python because the configuration is degenerate at function granularity, for the reason given in Table 3.

Exploration edges / depth	Python			Java			JavaScript			TypeScript		
	Δ	bootP	perm	Δ	bootP	perm	Δ	bootP	perm	Δ	bootP	perm
<i>Recall@20</i>												
<i>contains / d=4</i>	+0.07	< 0.0001***	0.0008***	+0.05	0.0001***	0.0060**	+0.08	< 0.0001***	< 0.0001***	+0.05	< 0.0001***	0.0006***
<i>invokes / d=2</i>	+0.15	< 0.0001***	< 0.0001***	+0.11	< 0.0001***	< 0.0001***	+0.13	< 0.0001***	< 0.0001***	+0.17	< 0.0001***	< 0.0001***
<i>imports / d=2</i>	+0.01	0.1318	0.4979	+0.00	1.0000	1.0000	+0.00	1.0000	1.0000	+0.00	1.0000	1.0000
<i>inherits / d=2</i>	+0.00	1.0000	1.0000	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
<i>contains / d=4 + invokes / d=2</i>	+0.17	< 0.0001***	< 0.0001***	+0.11	< 0.0001***	< 0.0001***	+0.15	< 0.0001***	< 0.0001***	+0.17	< 0.0001***	< 0.0001***
<i>Acc@20</i>												
<i>contains / d=4</i>	+0.05	0.0026**	0.0304*	+0.03	0.3690	1.0000	+0.07	< 0.0001***	0.0001***	+0.07	0.0005***	0.0080**
<i>invokes / d=2</i>	+0.14	< 0.0001***	0.0001***	+0.07	0.0038**	0.0224*	+0.12	< 0.0001***	< 0.0001***	+0.17	< 0.0001***	< 0.0001***
<i>imports / d=2</i>	+0.01	0.3648	1.0000	+0.00	1.0000	1.0000	+0.00	1.0000	1.0000	+0.00	1.0000	1.0000
<i>inherits / d=2</i>	+0.00	1.0000	1.0000	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
<i>contains / d=4 + invokes / d=2</i>	+0.15	< 0.0001***	0.0003***	+0.06	0.0082*	0.0392*	+0.14	< 0.0001***	< 0.0001***	+0.17	< 0.0001***	< 0.0001***

nearly all of the recoverable ground truth; without it, the same recall is attainable only by giving up the fixed budget. The residual gap also quantifies the headroom available to a better Stage-2 selector.

Table 21: **Stage-2 selector against the unfiltered neighborhood ceiling** on SWE-PolyBench with SweRankEmbed-Small, $N = 500$, $C = 5$. The ceiling is Recall of the full Stage-1 neighborhood with Stage 2 disabled, which returns hundreds of candidates rather than K ; the SpIDER column is Recall@20 under *invokes* expansion from Table 3, and the last column is its share of the corresponding ceiling.

Language	Ceiling (Stage 2 disabled)		SpIDER Recall@20	% of ceiling
	<i>contains/4</i>	<i>invokes/2</i>		
Python	0.517	0.626	0.56	89
Java	0.376	0.533	0.42	79
JavaScript	0.667	0.730	0.65	89
TypeScript	0.400	0.593	0.47	79

A.10 Comparison with Lexical and Agentic Retrieval

Coding agents in practice often locate code with lexical grep search and iterative tool calls rather than with a dense retriever. We therefore compare SpIDER against both, on SWE-PolyBench at $K = 20$.

Baselines. The *lexical* baseline operates over the same function nodes as the dense retriever: an LLM extracts up to 25 search terms from the issue, every function in the code graph is ranked by keyword-hit count, and the top- K are returned. The *agentic* baseline is OpenHands (Wang et al., 2025), a general-purpose software agent that explores the repository with its own tool suite and is

not restricted to a fixed retrieval budget.

Findings. Table 22 supports three observations. First, SpIDER outperforms the lexical baseline on Python, Java and TypeScript. Second, JavaScript is an exception where keyword matching wins, and this is the outcome our own locality analysis predicts: Table 6 identifies JavaScript as the dense-ranker-dominated split, with 53.6% of instances already fully covered in the top-100 and only 21.8% having any ground truth beyond the top-500, leaving little headroom for a graph mechanism, and it is also the split where lexical matching is strongest. Third, OpenHands attains the highest recall in every language, but by iterative exploration costing 223K–600K tokens per instance, between two and forty times the cost of SpIDER depending on the edge configuration, and without the fixed-budget guarantee that makes the retrieved set directly consumable by a downstream generator under a known context cost. The comparisons are therefore complementary rather than competing: SpIDER is agnostic to its first stage and improves both dense and sparse BM25 retrievers (Table 2), so the same expansion could be applied on top of a lexical first stage in the regime where that stage is strongest.

Deployability. The two regimes also differ in how they can be served. SpIDER’s cost is spread over C independent LLM calls that are issued concurrently and carry no context between them, so each call sees a small, bounded context regardless of repository size; Table 23 shows an open 30B coder model filling the filter role to within 0.01–0.05 Recall@20 of Claude Sonnet 4, and the bounded per-call context is what makes serving

Table 22: **Lexical and agentic retrieval on SWE-PolyBench** at $K = 20$. SpIDER rows use SweRankEmbed-Small with $N = 500$, $C = 5$; DER is carried over from Table 2 for reference. “Tok” is total (input + output) LLM tokens per instance. The upper block operates under a *fixed* retrieval budget of K functions, and the best entry per language and metric within that block is in light blue. OpenHands is listed separately because it is an iterative agent that is not budget-constrained: it explores until it decides to stop, so its recall is not obtained at a comparable operating point.

Method	Python			Java			JavaScript			TypeScript		
	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok	Recall@20	Acc@20	Tok
<i>Fixed retrieval budget ($K = 20$)</i>												
DER	0.42	0.35	0	0.31	0.24	0	0.52	0.45	0	0.29	0.21	0
grep + LLM keywords	0.46	0.40	1.1K	0.29	0.22	0.7K	0.69	0.61	0.7K	0.28	0.24	0.7K
SpIDER (<i>contains</i> /4)	0.49	0.40	12.9K	0.36	0.27	7.8K	0.60	0.52	14.9K	0.35	0.28	8.3K
SpIDER (<i>invokes</i> /2)	0.56	0.50	109.9K	0.42	0.30	87.4K	0.65	0.58	55.5K	0.47	0.41	92.7K
<i>Iterative agent (no fixed budget)</i>												
OpenHands	0.84	0.77	273.2K	0.71	0.59	371.4K	0.93	0.89	223.4K	0.66	0.62	599.7K

or fine-tuning a smaller model for this step practical. An iterative agent’s context instead grows with its tool-call history, so its budget is neither fixed in advance nor parallelizable in the same way, and the same substitution is correspondingly harder. This matters for the developer-in-the-loop setting SpIDER targets, where retrieval runs on an active repository at session start under a cost ceiling the developer chooses.

A.11 Per-Language Hyperparameter Sensitivity

C , d and N were grid-searched on the Python split of SWE-PolyBench and then held fixed for every language and benchmark. The per-language sweeps already reported let us check directly whether that transfer costs anything. Table 19 shows Recall@20 saturating at $C = 5$ for Python, JavaScript and Java, with only TypeScript still improving marginally up to $C = 9$ (+0.04 Recall@20). Table 9 shows monotone improvement in d for all four languages, so no language prefers a smaller depth. Table 10 shows saturation at $N \approx 300$ –500 in all four.

The Python-tuned configuration ($C=5, d=4, N=500$) is therefore at or near the per-language optimum everywhere we measure, and the single deviation, TypeScript preferring a larger C , would *increase* SpIDER’s reported gains rather than reduce them. The cross-language conclusions in Table 2 are consequently not an artifact of tuning on Python, and the fixed configuration is, if anything, conservative for TypeScript.

A.12 Qualitative Analysis

Which issues benefit from graph expansion?

Those where the issue names or points at one function, but the code that must change

is a structural neighbor of it; dense retrieval ranks the named function highly and stops there. In huggingface/transformers-5749 (Python, *contains*), the issue observes that `BasicTokenizer._clean_text` appears never to be called. Dense retrieval ranked `_clean_text` at the top but never surfaced the ground truth `BasicTokenizer.tokenize`, the function that must be edited to call it; SpIDER recovered `tokenize` as a same-class *contains* neighbor of the `_clean_text` center. In apache/dubbo-3479 (Java, *invokes*), the issue concerns URL keys in simplify mode. Dense retrieval missed both ground-truth functions, `ServiceConfig.doExport` and `ServiceConfig.doExportUrlsFor1Protocol`, which live in a different file from any top-ranked function; SpIDER recovered both as *invokes* neighbors of the center `ZKTools.pathToKey`. These are cross-file targets that *contains* expansion cannot structurally reach. Quantitatively, *invokes* expansion changes the retrieved set on 23% of instances (macro-averaged over languages) against 10% for *contains*, and improves recall in the large majority of those cases, with losses of 1–2%.

When does the Stage-2 filter judge incorrectly?

In two opposite ways. *Over-restrictive*: the prompt (Section A.5) instructs the selector to keep only high-confidence candidates and to return an empty set when in doubt. When the ground truth is reached from a center that is itself only tangentially related to the issue, the selector sees no issue-relevance and discards it. In keras-team/keras-18975, the ground truth `CompileLoss.build` was surfaced as a neighbor of `Trainer.compile` but rejected by the filter, so the instance remained a miss. Across languages the filter retains only 50–70% of the ground-truth functions that graph expansion successfully surfaces,

which is the same conclusion Table 21 reaches from the opposite direction: *the selector, not the graph, is the current bottleneck. Over-permissive*: dense call graphs can surface hundreds of neighbors per center, and the selector then admits tangential callers and test functions which, under a fixed budget, can evict a correctly retrieved ground-truth function. In `prettier/prettier-14788` (JavaScript), dense retrieval already had the sole ground truth `handleError` inside the top- K and neighbor flooding displaced it. This is why we cap the *invokes* neighborhood at the 100 highest-ranked neighbors per query (Table 3).

Operating boundaries. SpIDER helps when the target is both (i) within d hops of a top- C center and (ii) inside the top- N pool. Targets that are simultaneously low-ranked and graph-distant remain unreachable. The best-fit cases are call-chain bugs and localized refactors (*invokes*) and same-file or same-class fixes (*contains*); the worst-fit cases are large multi-function refactors and fixes that are semantically distant from every highly ranked function. Taken together with the filter’s 50–70% retention, this points at issue-aware neighbor ranking before filtering, and per-center neighbor caps on dense *invokes* graphs, as the most promising directions for improving SpIDER.

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 23 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 23 carries over two references from Table 2: the DER baseline (no SpIDER) and the closed-source Claude 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 discussion of this work.

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 is *contains* the default exploration edge?

It does not: Table 3 evaluates all four edge types, and *invokes* expansion outperforms *contains*-only expansion in every language. *contains* is the *default* because it is the cheapest reliable signal. It captures hierarchical structure (function $\subseteq$ class $\subseteq$ file $\subseteq$ directory), is identified reliably by syntax-based parsers across all four languages, and costs roughly an order of magnitude fewer tokens per instance than *invokes*, whose accurate call-graph resolution is also more brittle in dynamic languages (e.g., method dispatch through Python decorators, JavaScript prototype chains, dynamic imports). Where the token budget permits, *invokes* or *contains+invokes* is the stronger choice, and the margin is largest exactly where Table 6 predicts: Java, with the highest invoke-connectivity (0.63) and the lowest dense top-100 coverage (29%).

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. Table 20 reports paired-bootstrap and two-sided paired-permutation results for every language and edge configuration; the two tests agree in every cell, and all structural-edge Recall@20 gains are significant at $p < 0.01$ in all four languages. Section A.9 discusses the one non-significant structural cell.

Why is LocAgent evaluated only on Python?

LocAgent is an iterative LLM-driven agent that

Table 23: SpIDER pipeline results on SWE-PolyBench (function-level localization, dense retrieval, $K=20$, $N=500$, $C=5$, $d=4$, with a single non-iterative Stage-2 call per seed center). 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.

Model	Java		JavaScript		Python		TypeScript	
	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
<i>Open-source</i>								
Qwen2.5-Coder-7B-Instruct	0.32	<u>0.24</u>	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	<u>0.34</u>	<u>0.24</u>	<u>0.55</u>	<u>0.47</u>	<u>0.48</u>	<u>0.38</u>	<u>0.33</u>	<u>0.25</u>

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. A.8.

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 ~ 18 – 32 K tokens (per language), with output ~ 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.

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 $d \geq 4$. Table 21 measures this directly: the unfiltered neighborhood’s recall is an upper bound attained only by abandoning the fixed budget, and SpIDER captures 79–89% of it while returning just K functions.

Why are GraphCodeBERT, RepoGraph, CoRet, CoRNStack, RepoDeepSearch and RepoNavigator 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) has released neither embeddings nor code, preventing direct comparison. RepoGraph (Ouyang et al., 2025) is an agentic localization pipeline built on line-of-code-level graphs, constructed with the Python ast module, so it operates at a different granularity than function-level retrieval at fixed K and is not directly usable on SpIDER-Bench or on existing non-Python benchmarks. RepoDeepSearch (Ma et al., 2025) and RepoNavigator (Zhang et al., 2026) are recent agentic localization systems, but, like LocAgent, both lean heavily on Python-specific tooling such as ast and Python-oriented parsers; running them on the Java, JavaScript and TypeScript splits of SpIDER-Bench would require substantial pipeline modification, which is itself part of the gap SpIDER-Bench is intended to expose. RepoNavigator additionally has no public implementation. The agentic comparators within reach are therefore LocAgent (Tables 11 and 12) and OpenHands (Table 22), which is language-agnostic.

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. Con-

struction is deterministic given the repository state and uses no manual annotation; ground-truth labels are inherited from the upstream benchmarks.

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.