CODERETRX: EFFICIENT AND SCALABLE FUNCTION RETRIEVAL WITH HIERARCHICAL CODE SEARCH

Anonymous authors
Paper under double-blind review

Code: `https://anonymous.4open.science/r/CodeRetrX-54C6`

ABSTRACT

Function-level code search, the task of locating the functions in a large codebase that implement a given behavior (e.g., password authentication), is fundamental to program comprehension and underpins vulnerability discovery, large-scale migration, and data compliance. Emerging LLM-based tools (e.g., AI coding assistants) explore repositories iteratively, spending tokens at every step to decide what to read next, yet still miss most of the relevant functions. We present CODERETRX, a framework for near-exhaustive, low-cost function retrieval at scale. It is built on *hierarchical code search*: fragment-level embeddings first locate query-relevant fragments inside every function, an LLM-guided gate then retains only the functions that contain such fragments, and full LLM evaluation is applied to those candidates alone. This design bounds token usage while retaining fragment-level semantics, enabling precise retrieval across large codebases without exhaustive traversal. On CODERETRBENCH, our new benchmark of 18 multi-language repositories, CODERETRX attains 88.16% recall, the highest among the compared methods, using only 26.17% of the tokens of exhaustive LLM traversal. Against embedding-then-LLM baselines, it raises recall from 61.43% to 88.16% (1.44×) at 78.83% of their cost. Its recall is more than 25× that of industrial coding assistants (Cursor, Claude Code) and more than 6× that of LocAgent, a state-of-the-art localization agent. Deployed as the retrieval stage of our vulnerability-discovery agent, CODERETRX has contributed to 23 security advisories (21 CVEs, 20 of High or Critical severity) and two paid bug bounties in Web3 infrastructure, one rated High.

1 INTRODUCTION

Large language models have made natural-language questions about code commonplace, and a growing number of software-engineering tools depend on retrieving the code relevant to such a question. For most developer-facing uses, a few correct results suffice. High-stakes engineering tasks, however, pose a harder question: *can every relevant function be found, and at a cost that stays affordable as repositories grow?* Security auditing must locate every function that matches a vulnerability pattern or touches a compromised dependency; post-quantum cryptography migration must find every call site that implements, wraps, or invokes a cryptographic algorithm; GDPR compliance must inventory every code path that collects, processes, or persists user data. Unlike a developer seeking a single example to imitate, an auditor, migration engineer, or compliance officer must find all of them essentially. Each missed function is an unexamined attack surface, a broken security guarantee, or a legal exposure. Modern repositories, however, span millions of lines and several languages, and the logic for a single concept is scattered across files and layers, so manual inspection is prohibitively expensive.

Existing approaches occupy two extremes (Figure 1). At one extreme, index-based methods embed the code once or match keywords; they are cheap but incomplete. On our benchmark, a symbol-embedding baseline built on OpenAI *text-embedding-3-large* (TextEmb3-L) recalls only 61.43% of the relevant functions at a top-30% cutoff, and the LLM-grep pipelines behind Cursor and Claude Code recall under 4%. At the other extreme, exhaustive LLM traversal reads every function and achieves near-complete recall, but a single query over the 955K-LOC `keycloak` repository costs about $100 at frontier-model prices, and cost grows linearly with repository size and query count. Section 3 examines why each family is confined to its extreme; the region of practical interest, near-exhaustive recall at substantially lower cost, remains unoccupied. We call this the *recall–cost*

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

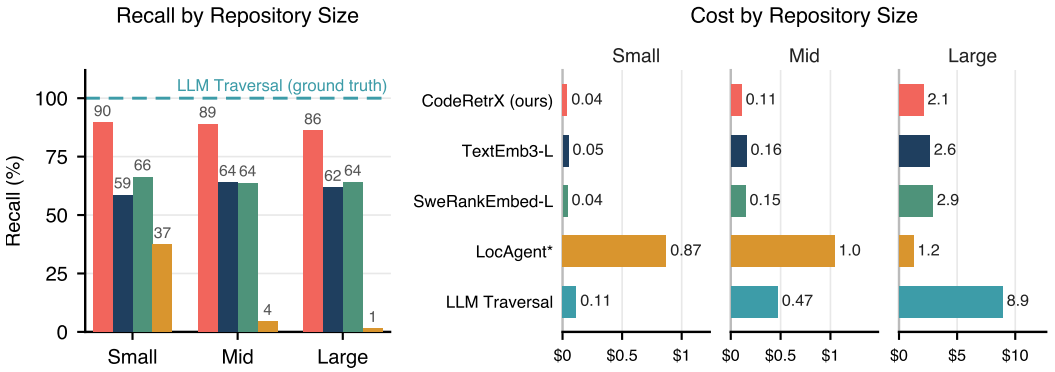

Figure 1: The recall–cost dilemma: recall and cost of existing retrieval methods and CODERETRX across three repository-size tiers, averaged per repository on our benchmark. **Left:** Recall by repository size: CODERETRX stays near the LLM-traversal ground truth at every scale. **Right:** Average per-repository cost: CODERETRX is the cheapest method in every tier (*LocAgent is evaluated on the Python repositories it supports).

dilemma: with existing approaches, higher recall requires proportionally more LLM computation, and lower cost necessarily sacrifices coverage.

Existing code-search and issue-localization benchmarks use snippet- or patch-based annotations that do not enumerate every function satisfying a repository-wide property. We therefore build CODERETRBENCH, a benchmark of 18 repositories spanning six language groups and three size tiers, with queries drawn from vulnerability localization, post-quantum cryptographic migration, and code refactoring (Section 4). For every query, it provides a reference set of relevant functions obtained by exhaustive LLM traversal, together with a protocol for recording the tokens each method spends, so that recall and cost can be compared directly.

To escape the dilemma, we present CODERETRX, a framework that decouples coverage from token expenditure through *hierarchical code search*, a coarse-to-fine procedure in three steps. **(i)** Static analysis parses the repository into function units and embeds the fragments of each unit, so that every function can be searched internally without an LLM reading it. **(ii)** Coarse recall retrieves the query-relevant fragments of each unit and applies low-cost LLM judgments to them, yielding an over-approximation of the relevant function set. **(iii)** Refined recall confines full LLM reasoning to the small set of candidates that pass the coarse stage. Because only the final step reads whole functions, recall remains near exhaustive while cost falls by roughly a factor of four. On our benchmark, CODERETRX reaches 88.16% recall, 26.73 points above the TextEmb3-L-30% baseline, at less than 80% of its cost. Cursor and Claude Code reach only 3–4% of CODERETRX's recall, and LocAgent (Chen et al., 2025) reaches 16.3% of it at comparable cost. As a case study, CODERETRX serves as the retrieval stage of a vulnerability-discovery workflow that has so far produced 23 published security advisories, 20 rated High or Critical, and two paid bug bounties (Section 6, Appendix A); neither extreme of the dilemma could sustain such a workflow.

This paper makes three contributions:

- We characterize the recall–cost dilemma of high-recall function retrieval and show why embedding search, LLM-grep, and exhaustive traversal each fail to escape it (Section 3).
- We build CODERETRBENCH, a benchmark of 18 repositories spanning six language groups and three task families, with traversal-based relevance labels and joint evaluation of recall and LLM cost (Section 4).
- We propose CODERETRX, a hierarchical framework that combines fragment-level retrieval with staged LLM verification (Section 5). It improves recall by 26.73 percentage points over the TextEmb3-L-30% baseline at lower cost (Section 6) and underpins a deployment that has produced 23 published security advisories (Appendix A).

2 BACKGROUND AND RELATED WORK

Embedding-based code retrieval. Embedding-based code retrieval maps natural-language queries and code units into a shared vector space and ranks code by vector similarity; existing approaches

fall into dense single-vector retrieval, late-interaction retrieval, and post-retrieval reranking. Dense single-vector retrievers, including code-pretrained models such as CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5 (Feng et al., 2020; Guo et al., 2020; 2022; Wang et al., 2021; Gu et al., 2018; Sachdev et al., 2018), are inexpensive at query time. Their recall, however, degrades on repository-scale tasks with heterogeneous naming and cross-file dependencies. Late-interaction models and rerankers (Khattab & Zaharia, 2020; Santhanam et al., 2022) improve the ranking of retrieved candidates but cannot recover units outside a fixed top-k (Thakur et al., 2021; Muennighoff et al., 2023; Enevoldsen et al., 2025; Zhang et al., 2023).

LLM keyword generation and regex pipelines. Agentic coding tools commonly locate code by having an LLM generate keywords or regular expressions and matching them lexically; we refer to this family as LLM-grep. Lexical matching is limited by vocabulary mismatch and ambiguity (Di Grazia & Pradel, 2023; Sun et al., 2024), and LLM-generated regexes are frequently incorrect or vulnerable to ReDoS (Siddiq et al., 2024a;b).

Code localization agents. LLM-based code localization agents for repository-level software engineering tasks fall into two paradigms. *Agentic localizers* such as LocAgent (Chen et al., 2025), OrcaLoca (Yu et al., 2025), CoSIL (Jiang et al., 2025), and FuseSearch (Xu et al., 2026) explore a repository iteratively through tool calls. *Retrieve-and-rerank methods* such as SweRank and SweRank+ (Reddy et al., 2026; 2025) first retrieve candidate code regions and then rerank them with an LLM. Even state-of-the-art localizers exhibit low repository-wide recall: their search is tuned to a few likely edit locations, and relevant functions elsewhere are neither visited nor shortlisted.

None of these benchmarks labels every relevant function in a repository; CODERETRBENCH does so via exhaustive LLM traversal for complex queries across languages (Section 4).

Appendix C discusses all three families in more detail.

3 MOTIVATION

3.1 PROBLEM FORMULATION

A repository is parsed into a set of *function units* $\mathcal{U} = \{u_1, \ldots, u_n\}$ consisting of functions, methods, and classes. A query q is a natural-language description of a property, and its target set $\mathcal{T}(q) \subseteq \mathcal{U}$ is the set of all units satisfying that property. A retrieval method returns a set $\mathcal{S}(q) \subseteq \mathcal{U}$ at a cost of $C(q)$ LLM tokens. The objective is to maximize coverage of the target set at a small fraction of the cost $C_{\text{trav}}(q)$ of LLM inspection of every unit:

$$\max_{\mathcal{S}} \frac{|\mathcal{S}(q) \cap \mathcal{T}(q)|}{|\mathcal{T}(q)|} \quad \text{subject to} \quad C(q) \ll C_{\text{trav}}(q). \tag{1}$$

$C(q)$ comprises all input and output tokens consumed in answering q, including candidate verification; one-time parsing and embedding costs are amortized across queries and excluded. Recall is the primary metric because a missed unit is never inspected downstream. We evaluate against a traversal-based approximation to $\mathcal{T}(q)$; Section 4 describes the reference labels and how metrics are computed.

3.2 THE RECALL–COST DILEMMA

Restricting which units an LLM inspects reduces cost but may exclude relevant units before their implementations are examined. Consider post-quantum migration in *anubis*, where the query targets all code that implements, references, or uses cryptographic functionality. The relevant logic is embedded in ordinary initialization and challenge-handling functions, so neither a function's name nor its whole-function similarity to the query reliably signals relevance. Appendix B lists the missed functions.

Embedding search and LLM-grep are inexpensive but incomplete. Two families of methods answer a query without LLM inspection of the repository, and both sacrifice recall as a result. (1) In *embedding search*, a single vector can obscure a small but decisive operation inside a larger function, and units ranked below the retrieval cutoff never reach LLM verification (Weller et al., 2026). In the motivating example, retrieving the top 30% of units still misses the `main` and `New` functions that configure cryptographic functionality (Listing 1). Across our benchmark, the TextEmb3-L-30% baseline attains only 61.43% recall at this cutoff (Section 6.2). Enlarging the candidate budget sends more implementations to the LLM and raises verification cost. (2) In *LLM-grep*, coverage depends on anticipating the vocabulary of every relevant implementation (Sun et al., 2024; Siddiq et al., 2024a).

Table 1: Task composition of CODERETRBENCH. Reference entries are distinct unit IDs per repository–family pair; a unit relevant to several families is counted once in each.

Task family	Topics	Repos. with positives	Reference entries
Vulnerabilities	9	18	3,175
Refactoring	4	17	35,506
Crypto/PQ migration	3	16	15,989

In *anubis*, generic cryptographic terms fail to retrieve functions such as `issueChallenge` and `process` (Listing 2). Functions that no pattern matches are never examined, and a handful of hits offers no evidence that the search is complete. The Cursor and Claude Code pipelines recover under 4% of the traversal-derived target set in our case study (Appendix A).

Exhaustive LLM traversal is complete but unaffordable. Traversal recovers the functions missed in the motivating example, but its cost grows with the size of the repository for every query. A single query over the 955K-LOC `keycloak` repository consumes more than 13 million input tokens. Repeated across repositories and query families, such inspection becomes the dominant expense.

The dilemma is therefore how to retain broad coverage while limiting the number of full implementations an LLM must read. To address this, we first introduce a dataset that measures both coverage and cost (Section 4), then describe how CODERETRX resolves the dilemma with a hierarchical retrieval workflow (Section 5).

4 THE CODERETRBENCH DATASET

CODERETRBENCH is a benchmark for repository-level retrieval built from natural-language queries and exhaustive LLM relevance judgments.

4.1 DATA SOURCES AND RETRIEVAL TASKS

CODERETRBENCH comprises 18 open-source repositories in six language groups (C/C++, Rust, Go, Java, Python, and JavaScript/TypeScript), one *Small*, *Mid*, and *Large* repository per group (Table 2).

Each query targets every function unit that satisfies its description, including wrappers and callers; queries fall into three task families (Table 1): **vulnerability-related retrieval** (e.g., externally influenced file operations, command execution, deserialization), **refactoring** (e.g., synchronous server calls, parsing, HTTP request handling), and **cryptographic migration** (general cryptography, quantum-vulnerable asymmetric algorithms, and symmetric algorithms with reduced security under quantum attacks).

4.2 DATA CURATION AND RELEVANCE ANNOTATION

We pin each repository to a commit hash and parse the snapshot into function units annotated with file paths and source spans; all compared methods operate on these same units and locations. For each query, *LLM Traversal* applies *gpt-oss-120b* to every unit; the resulting reference set $\widehat{\mathcal{T}}(q)$ approximates the ideal target set $\mathcal{T}(q)$, so recall measures agreement with these model labels.

Deduplicating unit IDs within each repository–family pair yields 54,670 reference entries; 51 of the 54 pairs have nonempty reference sets. Cost totals include all 54 pairs, empty reference sets included. Appendices E and G detail the artifacts, aggregation, and snapshot alignment.

4.3 EVALUATION METRICS

Recall and candidate precision. Recall is the fraction of the reference set covered by the final result $\mathcal{S}(q)$:

$$\mathrm{Recall}(q) = \frac{|\mathcal{S}(q) \cap \widehat{\mathcal{T}}(q)|}{|\widehat{\mathcal{T}}(q)|}. \tag{2}$$

Repository-level recall pools recovered and reference counts over all task families; group-level scores are the mean of repository recalls, and task-family means omit pairs with empty reference sets. Candidate precision (**Acc.**, Table 2) is the fraction of candidates entering final LLM verification that are judged relevant.

Tokens and monetary cost. We count query-processing and retrieval tokens as $C(q) = N_{\mathrm{in}}(q) + N_{\mathrm{out}}(q)$ and convert to USD with separate input and output prices. Cost ratios are normalized to *LLM Traversal* and exclude one-time parsing and embedding costs (Appendix G).

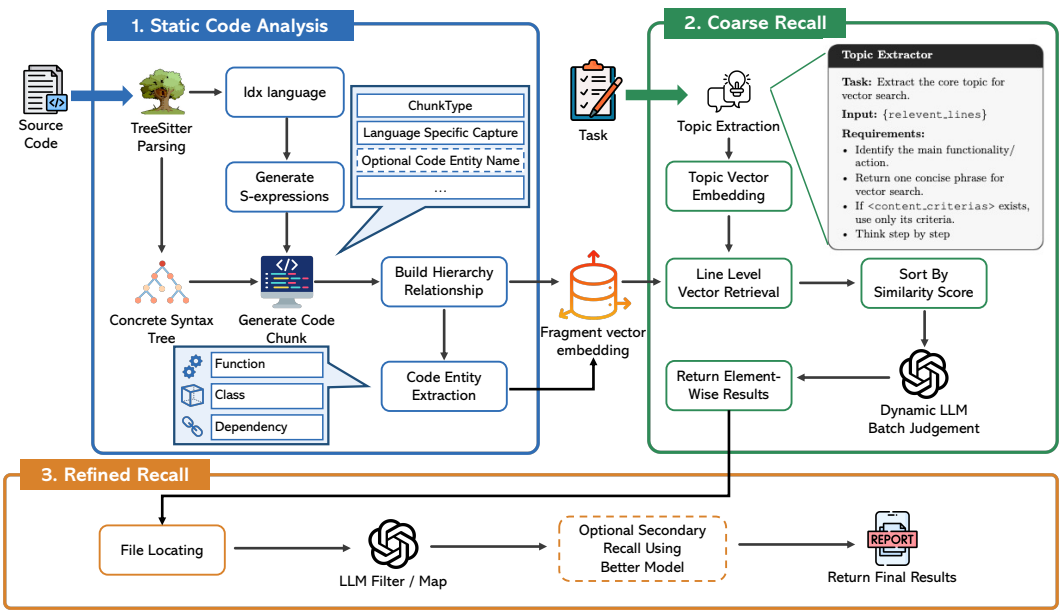

Figure 2: The CODERETRX pipeline: static analysis indexes fragments within function units; Coarse Recall selects candidates from fragment evidence; Refined Recall verifies full implementations.

5 THE CODERETRX FRAMEWORK

5.1 OVERVIEW

CODERETRX addresses the recall–cost dilemma with a two-stage, coarse-to-fine retrieval design. *Static analysis* parses the repository into function units, records their file structure, and indexes the fragments inside each unit. At query time, *Coarse Recall* searches this index and selects candidate units from LLM judgments on short fragments; *Refined Recall* then examines their full implementations. Figure 2 shows the static indexing step followed by these two retrieval stages.

Function units are the objects the system judges and returns, matching the targets in Section 3.1. The search operates on fragments, the code segments within a unit: one relevant operation suffices to flag a function even if the rest of its body is unrelated to the query. Per-unit fragment retrieval recalls each function through its own fragments rather than a single whole-body embedding, while the recorded file and unit structure supplies context for later verification. Fragments are single source lines here; statements or syntax subtrees would also fit, since the coarse-to-fine design does not depend on fragment granularity.

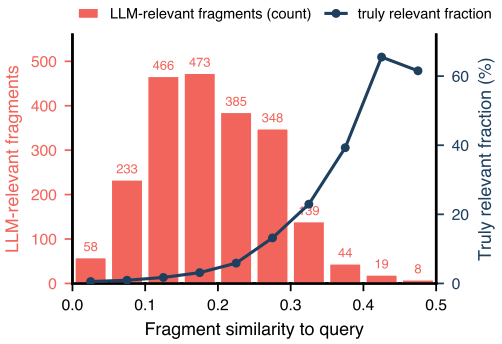

Figure 3: Fragment relevance versus embedding similarity (five repositories, 16 queries, 87,748 fragment–query pairs). The relevance rate rises with similarity (line), yet most relevant fragments fall at intermediate scores (bars).

The two stages rely on different signals: embedding similarity ranks fragments, whereas LLM judgments determine which fragments, and hence which units, are relevant. Figure 3 illustrates why both signals are needed. The fraction of relevant fragments rises from 0.6% in the lowest similarity bin to approximately 65% in the highest, so similarity is an informative ranking signal. Yet many relevant fragments receive intermediate scores: a top-30% cutoff captures only 63.0% of them, and a top-10% cutoff only 32.2%. CODERETRX therefore retrieves fragments within each unit and judges them in successive batches. Only units that pass this coarse stage incur full-body LLM inspection. Budget and stopping rules govern the residual recall–cost trade-off; we measure their effect against the traversal reference in Section 6.

Algorithm 1 CODERETRX retrieval

Require: Repository R, query q, fragment width k, round limit M, batch ratio ρ
Ensure: Relevant function units $\mathcal{S}$ with locations and judgments
1: $\mathcal{U} \leftarrow$ PARSEUNITS(R) ▷ retain file and fragment links
2: $I \leftarrow$ INDEXFRAGMENTS$(\mathcal{U})$ ▷ reuse across queries
3: $t \leftarrow$ EXTRACTTOPICS(q)
4: $Q \leftarrow$ SORTBYSIMILARITY$(\bigcup_{u \in \mathcal{U}}$ TOPK$(I_u, t, k))$
5: $E \leftarrow \varnothing$; $b \leftarrow \max(1, \lfloor (|Q|/k)\rho \rfloor)$
6: **for** $r = 1, \ldots, M$ **do**
7: $(B, Q) \leftarrow$ NEXTBATCH(Q, E, b) ▷ one fragment per unit; units in E skipped
8: **if** $B = \varnothing$ **then break**
9: $E \leftarrow E \cup$ JUDGEFRAGMENTS(q, B)
10: **end for**
11: $\mathcal{S} \leftarrow$ JUDGEFULLBODIES$(q,$ RESOLVECANDIDATES$(\mathcal{U}, E))$
12: **if** secondary review is enabled **then** $\mathcal{S} \leftarrow$ RECHECK$(q, \mathcal{S})$
13: **return** $\mathcal{S}$

5.2 RETRIEVAL PIPELINE

Static Analysis. We parse source files with Tree-Sitter (Brunsfeld & contributors, 2018) to construct concrete syntax trees (CSTs; Algorithm 1, lines 1–2). From these trees we extract function units with file paths, source spans, and parent–child links (Jiang et al., 2007; Luan et al., 2019), along with the definitions, references, and imports from which compact contexts are assembled. We embed every fragment and store it with its unit identifier, file path, and position. This index preserves structural boundaries and supports similarity search within individual units; it is constructed once and reused across queries.

Coarse Recall. An LLM first extracts key terms and phrases from the task description (line 3) to serve as retrieval cues (Nogueira et al., 2019; Gao et al., 2023). We retrieve up to k fragments within each unit and rank the resulting candidates globally by similarity (line 4) (Gao et al., 2021; Formal et al., 2021). Because every unit contributes its top-k fragments, a single function with many highly similar fragments cannot crowd others out of the candidate pool. In each round the LLM judges a batch of at most b fragments, one per unit, and the fragments judged relevant are added to the evidence set E (lines 7–9). NEXTBATCH defers additional fragments from the same unit to later rounds. Units already judged relevant are not re-examined, whereas a fragment rejected in one round does not exclude its unit from later rounds; judgment ends when no eligible candidates remain or the round budget is exhausted (lines 6–8). Appendix D specifies the budgets and batching limits.

Refined Recall. We map the fragments in E to their function units, load each unit's complete implementation, and judge it against the original query (line 11). Full-body verification thus retains only candidates that satisfy the task description and removes coarse-stage false positives. An optional secondary review by a stronger model checks the retained units (line 12); results are returned as file and unit locations with justifications. Appendix H provides the filter/map schemas and implementation details.

6 EXPERIMENTS

6.1 EXPERIMENTAL SETUP

We evaluate on CODERETRBENCH with the traversal reference and metrics in Section 4. For a controlled comparison, CODERETRX, Symbol Vector Embedding with OpenAI *text-embedding-3-large* (TextEmb3-L, top-10/20/30%), and traversal use the same parser and *gpt-oss-120b* judge; SweRankEmbed and LocAgent are the localization baselines. We use $k = 10$, $M = 30$, and $\rho = 0.2$ in Algorithm 1; Appendices D and G detail the settings and baseline protocols.

6.2 MAIN RESULTS

CODERETRX attains 88.16% repository-averaged recall at a total cost of \$13.48, compared with 61.43% at \$17.10 for TextEmb3-L-30%. Figure 4 and Table 2 report recall, candidate precision, and cost across repositories; the table also summarizes results by size.

Comparison with retrieval baselines. TextEmb3-L recovers 39.5%, 53.6%, and 61.4% of the reference units at the top-10%, 20%, and 30% cutoffs; verifying the top-30% set costs \$17.10.

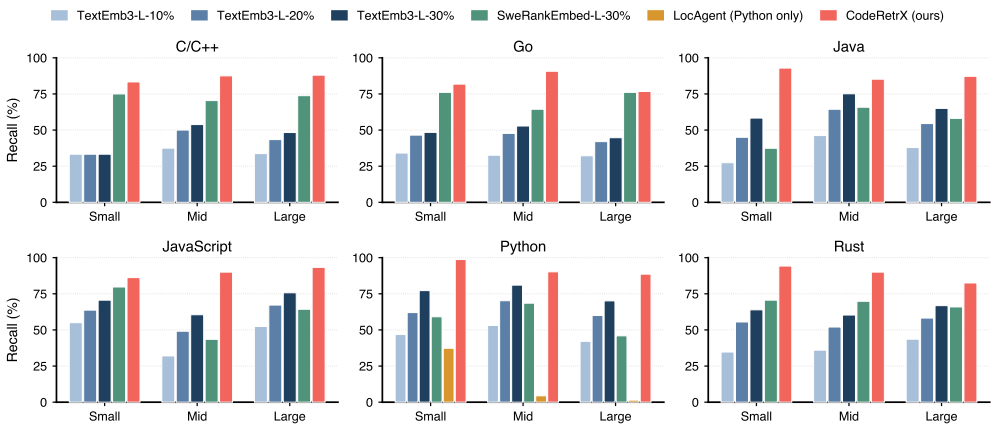

Figure 4: Recall by language and repository size. TextEmb3-L baselines use top-10%/20%/30% cutoffs; LocAgent appears only in the Python panel, the sole language it supports.

SweRankEmbed-Large (Reddy et al., 2026), a retriever trained for issue localization, reaches 29.7%, 50.6%, and 64.7% under the snapshot-aligned protocol of Appendix G. The Large variant exceeds the Small variant at every cutoff (64.7% vs. 52.2% at top-30%), confirming that retriever quality improves candidate ordering. Both top-30% sets nonetheless omit over a third of the reference units, whereas CODERETRX recovers 88.16%, 26.7 percentage points (pp) above TextEmb3-L-30% and 23.5 pp above SweRankEmbed-Large. Verifying the SweRankEmbed-Large set costs $18.58, excluding retriever inference, compared with $13.48 for the complete CODERETRX pipeline. Stronger ranking alone therefore leaves much of the gap in coverage and in full-body verification cost unclosed.

Comparison with agentic baselines. LocAgent (Chen et al., 2025) supports only Python, so we evaluate it on the three Python repositories in CODERETRBENCH. On these repositories, LocAgent averages 14.35% recall, ranging from 37.3% on `python-p2p` to 1.4% on `zulip`. CODERETRX averages 92.5% on the same repositories at a lower total cost ($2.95 vs. $3.15). A separate comparison with Cursor and Claude Code gives 3.4% and 2.3% recall, respectively, versus 87.2% for CODERETRX. Their retrieval traces illustrate how keyword-driven exploration misses implementations whose identifiers differ from the query vocabulary (Appendix A). All three agentic pipelines therefore trail CODERETRX by more than 70 pp, a gap far wider than that of any retrieval baseline.

Gains hold across task families. The gain is largest for cryptographic migration (+29.5 pp), followed by refactoring (+20.5 pp) and vulnerability-related retrieval (+18.0 pp) (Table 3). Despite their specialized vocabulary, cryptographic queries are the family on which TextEmb3-L-30% performs worst. These queries target *uses* of cryptography, so an algorithm name alone does not identify every caller or initialization routine. The functions missed in Section 3.2 illustrate this: relevance is visible in local operations that the enclosing function's name does not reflect. Fragment-level retrieval targets these local operations directly. The gain holds on every repository with a nonempty reference set: all 16 for cryptographic migration and all 17 for refactoring. Vulnerability-related retrieval is less uniform: recall rises on 13 repositories, is unchanged on one, and falls on four. Its reference sets range from 6 to 729 units, so a single miss weighs more heavily in the smaller sets.

Gains persist as repositories grow. On small, mid-sized, and large repositories, CODERETRX attains 89.5%, 88.9%, and 86.0% repository-averaged recall, against 58.6%, 63.9%, and 61.8% for TextEmb3-L-30%. On large repositories the gap is still 24.3 pp, so the advantage does not shrink as the search space grows. Candidate precision explains part of the aggregate efficiency advantage. From small to large repositories, the candidate precision of TextEmb3-L-30% falls from 26.0% to 8.2%, whereas CODERETRX retains 30.0% on the largest tier (Table 2). Consequently, a far larger share of baseline candidates undergoes full-body judgment without yielding relevant units. CODERETRX still misses some relevant code: cryptographic recall on `go-ethereum` is 75.8%, and vulnerability-related recall on `cryptomator` is 81.7%, below the baseline's 92.9% (Appendix F).

Cost–recall trade-offs. The cost advantage varies by task family. Vulnerability-related and cryptographic retrieval cost 33.3% and 30.1% less than TextEmb3-L-30%, respectively; refactoring costs 12.1% more ($4.87 versus $4.35) in exchange for a 20.5 pp recall gain. The increase is concentrated: refactoring is cheaper on 10 of the 18 repositories, but `keycloak` and `zulip` alone add $0.72,

Table 2: Candidate precision (Acc., %) and total LLM cost (USD) across all 18 repositories. TextEmb3-L denotes Symbol Vector Embedding with *text-embedding-3-large* at the top-30% cutoff. LOC is shown in parentheses; the final rows average precision and sum cost within each size tier. LocAgent is evaluated on Python repositories.

Method	**Small** (≤10K LOC)		**Mid** (10K–100K LOC)		**Large** (>100K LOC)	
	Acc.	**Cost**	**Acc.**	**Cost**	**Acc.**	**Cost**
Rust	boringtun (7,937)		rosenpass (30,063)		neon (375,060)	
TextEmb3-L	26.9%	0.05	14.4%	0.20	6.4%	2.04
SweRankEmbed-L	31.6%	0.05	19.6%	**0.14**	8.2%	1.88
CODERETRX (Ours)	**63.1%**	**0.04**	**37.6%**	**0.14**	**24.7%**	**1.23**
Python	python-p2p (1,189)		magic-wormhole (27,538)		zulip (499,528)	
TextEmb3-L	45.6%	**0.01**	7.5%	0.18	5.7%	3.47
SweRankEmbed-L	45.0%	**0.01**	9.2%	0.14	7.8%	2.11
LocAgent	59.2%	0.87	23.7%	1.04	28.1%	**1.23**
CODERETRX (Ours)	**69.9%**	**0.01**	**40.2%**	**0.13**	**33.9%**	2.81
Go	anubis (5,163)		fscrypt (18,815)		go-ethereum (272,576)	
TextEmb3-L	15.3%	0.09	23.5%	0.05	12.7%	1.29
SweRankEmbed-L	19.5%	**0.04**	25.6%	0.05	14.9%	1.26
CODERETRX (Ours)	**60.5%**	0.09	**36.6%**	**0.02**	**19.1%**	**0.72**
Java	client-encryption-java (8,654)		cryptomator (38,883)		keycloak (955,351)	
TextEmb3-L	33.0%	0.08	9.8%	0.28	9.3%	6.86
SweRankEmbed-L	38.0%	0.12	11.0%	0.29	10.1%	8.27
CODERETRX (Ours)	**39.7%**	**0.06**	**16.6%**	**0.16**	**25.9%**	**6.22**
JS/TS	swifty (8,295)		vaultwarden (62,609)		padloc (136,233)	
TextEmb3-L	12.7%	0.06	12.3%	0.19	10.9%	0.54
SweRankEmbed-L	13.1%	**0.03**	13.8%	0.23	10.2%	**0.46**
CODERETRX (Ours)	**38.1%**	0.04	**18.0%**	**0.18**	**39.8%**	0.50
C/C++	inih (2,193)		cJSON (25,423)		llama.cpp (537,200)	
TextEmb3-L	22.6%	0.01	19.7%	0.06	4.3%	1.66
SweRankEmbed-L	25.5%	<**0.01**	18.6%	0.05	5.5%	3.46
CODERETRX (Ours)	**39.3%**	0.01	**40.1%**	**0.02**	**36.5%**	**1.10**
Mean precision / total cost						
TextEmb3-L	26.0%	0.30	14.6%	0.95	8.2%	15.86
SweRankEmbed-L	28.8%	0.25	16.3%	0.88	9.5%	17.44
CODERETRX (Ours)	**51.8%**	**0.24**	**31.5%**	**0.65**	**30.0%**	**12.59**

Table 3: Task-family results. Recall (%) is averaged over the n repositories with nonempty reference sets; cost change is the total over all 18 repositories relative to TextEmb3-L-30%. Appendix F gives counts and aggregation details.

Task family	n	**TextEmb3-L-30%**	**CODERETRX**	**ΔRecall (pp)**	**ΔCost**
Vulnerabilities	18	71.97	89.95	+17.98	−33.3%
Refactoring	17	67.33	87.78	+20.46	+12.1%
Crypto/PQ migration	16	60.94	90.44	+29.51	−30.1%

which savings elsewhere only partly offset. These two repositories contain 25,109 refactoring reference units; CODERETRX recovers 6,223 more than TextEmb3-L-30%. Broad program-role queries thus trade additional verification for substantially greater coverage. Savings in the other two families outweigh this increase, yielding the 21.2% reduction overall.

6.3 ABLATION STUDIES

Raising the TextEmb3-L cutoff from 10% to 30% adds 22 pp of average recall with diminishing returns: 14 pp for 10%→20% and 8 pp for 20%→30%. The baseline remains 24–31 pp below CODERETRX across size tiers (Figure 4), indicating that a larger candidate budget alone does not close the gap. Issuing relevance judgments through the function-calling interface (Appendix H) instead of free-text responses consistently lowers output tokens and cost, and increases stability.

Table 4: All 23 public advisories found with CODERETRX as the retrieval stage; severity follows the GitHub Advisory Database, and entries without a CVE carry their GHSA identifier.

Advisory	Project	Weakness	Severity
CVE-2025-57801	gnark (Go)	EdDSA/ECDSA signature malleability	High
CVE-2025-61913	Flowise (npm)	Arbitrary file write (`WriteFileTool`)	Critical
CVE-2026-22698	sm2 (Rust)	32-bit biased nonce in SM2-PKE	High
CVE-2026-22699	sm2 (Rust)	Unchecked affine-point decoding	High
CVE-2026-22700	sm2 (Rust)	Insufficient length validation	High
CVE-2026-23965	sm-crypto (npm)	SM2-DSA signature forgery	High
CVE-2026-23966	sm-crypto (npm)	SM2-PKE private-key recovery	Critical
CVE-2026-23967	sm-crypto (npm)	SM2-DSA signature malleability	High
CVE-2026-25059	OpenList (Go)	Path traversal in copy/remove handlers	High
CVE-2026-25060	OpenList (Go)	Insecure TLS default configuration	High
CVE-2026-25160	Alist (Go)	Insecure TLS configuration	Critical
CVE-2026-25161	Alist (Go)	Path traversal in file-operation handlers	High
CVE-2026-26007	cryptography (Python)	Missing subgroup validation (SECT curves)	High
CVE-2026-27483	MindsDB (Python)	Path traversal in `/api/files`, RCE	High
CVE-2026-32614	gmsm (Go)	SM9 infinity-point ciphertext forgery	Critical
CVE-2026-39414	MinIO (Go)	Unbounded allocation in S3 Select CSV parsing	High
CVE-2026-46695	BoxLite (Go)	Permission bypass on read-only files	Critical
CVE-2026-46703	BoxLite (Go)	Path traversal to host file write	Critical
CVE-2026-47213	BoxLite (Python)	Timeout bypass	Moderate
CVE-2026-53495	containerd (Go)	CRI `ExecSync` goroutine leak (DoS)	Moderate
CVE-2026-61539	Xinference (Python)	RCE via `eval()` in tool-call parsing	Critical
GHSA-39g5-644c-qwcg	apple/container (Swift)	`pf` rule injection via domain argument	Low
GHSA-j44m-5v8f-gc9c	Flowise (npm)	Arbitrary file read (`ReadFileTool`)	High

Stage-wise breakdown. We separate the two stages on all repository–query pairs (Appendix G). Of the units selected by fragment judgments, 39.9% are relevant (48.5%, 32.8%, and 38.6% on small, mid-sized, and large repositories). The candidate set handed to Refined Recall contains 93.3% of the reference units, a recall ceiling above the final 88.16%, so Refined Recall spends its budget on discarding candidates rather than on recovering missed units. The coarse stage accounts for 59.8% of cost at *gpt-oss-120b* prices; Refined Recall takes 42.3%, 57.8%, and 38.8% of cost on small, mid-sized, and large repositories.

Limitations. Indexing a large codebase can still take tens of minutes, so reusing the index across queries matters in practice. The coarse stage also misses relevant units whose fragments do not reveal their role (Section 6.2). Incremental index updates and verification budgets scaled to query breadth would extend the approach to evolving repositories.

6.4 REAL-WORLD DEPLOYMENT

In deployment, analysts describe a vulnerability class in natural language, CODERETRX retrieves candidate functions with source locations, a downstream agent analyzes them, and experts confirm findings and coordinate disclosure.

From a task description to a confirmed finding. For `gnark`, a zero-knowledge proof library, the query asked for computational circuits that should be reviewed for underconstrained behavior. Among the 66 returned units were two signature-verification functions, in which the analysis agent and expert review confirmed signature malleability in the EdDSA and ECDSA implementations, disclosed as CVE-2025-57801 and rated High (CVSS 8.6) (GitHub Advisory Database, 2025); Appendix A gives the original query.

Deployment outcomes. The workflow has produced 23 public advisories: 21 with CVE identifiers and 20 rated High or Critical. Nine concern cryptographic libraries; the other 14 span container runtimes, storage/file servers, and LLM-application frameworks (Table 4). Two further reports on production blockchain infrastructure earned bug bounties, and in every case the vulnerable functions were among the retrieved candidates.

7 CONCLUSION

High-recall function retrieval faces a recall–cost dilemma: inexpensive retrievers are incomplete, and exhaustive LLM traversal is unaffordable. Existing benchmarks, built for general software-engineering tasks where a few correct hits suffice, label only those hits, so we built CODERETR-BENCH: 18 repositories, six language groups, and three task families, each query paired with a traversal-derived reference set and a token-accounting protocol that measures coverage and cost jointly. To escape the dilemma, we designed CODERETRX, a coarse-to-fine architecture that indexes fragments once, judges them within every unit in batches, and reserves full-body verification for the units that survive. It reaches 88.16% recall, 26.73 points above the TextEmb3-L-30% baseline at 21.2% lower cost, and has supported 23 public security advisories in deployment.

AI USE STATEMENT

In this work, we used generative AI tools to help implement parts of the evaluation scripts and to assist in interpreting aggregate results. We have not used generative AI tools to generate synthetic datasets, to formulate hypotheses or mathematical claims, or to design the research methodology or experiments; the remaining required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to edit the manuscript for readability, to draft parts of the text, to create and modify figures, and to format references. We have reviewed all AI-assisted work: all AI-assisted code was tested for correctness by the authors, and all AI-assisted text and figures were checked against the underlying experimental data. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Appendix E provides repository snapshots, queries with reference locations, index-construction scripts, prompts, and metric scripts. Appendices D and H detail the retrieval settings and implementation.

REFERENCES

Max Brunsfeld and contributors. Tree-sitter. `https://github.com/tree-sitter/tree-sitter`, 2018. Parser generator and incremental parsing library.

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

Luca Di Grazia and Michael Pradel. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31, 2023.

Andor Diera, Abdelhalim Dahou, Lukas Galke Poech, Fabian Karl, Florian Sihler, and Ansgar Scherp. Gencodesearchnet: A benchmark test suite for evaluating generalization in programming language understanding. In *Proceedings of the 1st GenBench Workshop on (Benchmarking) Generalisation in NLP*, pp. 12–24, 2023.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents, 2025. URL `https://arxiv.org/abs/2508.00083`.

Kenneth Enevoldsen, Isaac Chung, Imene Kerboua, Márton Kardos, Ashwin Mathur, David Stap, Jay Gala, Wissam Siblini, Dominik Krzemiński, Genta Winata, et al. Mmteb: Massive multilingual text embedding benchmark. In *International Conference on Learning Representations*, volume 2025, pp. 101715–101771, 2025.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. Splade v2: Sparse lexical and expansion model for information retrieval. *arXiv preprint arXiv:2109.10086*, 2021.

Luyu Gao, Zhuyun Dai, and Jamie Callan. Coil: Revisit exact lexical match in information retrieval with contextualized inverted list. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3030–3042, 2021.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1762–1777, 2023.

GitHub Advisory Database. Security advisory: Signature malleability in gnark. `https://github.com/Consensys/gnark/security/advisories/GHSA-95v9-hv42-pwrj`, 2025. Accessed: 2025-10-21.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th international conference on software engineering*, pp. 933–944, 2018.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

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

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105. IEEE, 2007.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching, 2025. URL https://arxiv.org/abs/2503.22424.

Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pp. 39–48, 2020.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22074–22091, 2025.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–28, 2019.

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

Rodrigo Nogueira, Wei Yang, Jimmy Lin, and Kyunghyun Cho. Document expansion by query prediction. *arXiv preprint arXiv:1904.08375*, 2019.

Revanth Gangi Reddy, Ye Liu, Wenting Zhao, JaeHyeok Doo, Tarun Suresh, Daniel Lee, Caiming Xiong, Yingbo Zhou, Semih Yavuz, and Shafiq Joty. Swerank+: Multilingual, multi-turn code ranking for software issue localization. *arXiv preprint arXiv:2512.20482*, 2025.

Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. Swerank: Software issue localization with code ranking, 2026. URL https://arxiv.org/abs/2505.07849.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*, pp. 31–41, 2018.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3715–3734, 2022.

Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna CS Santos. Re (gex| dos) eval: evaluating generated regular expressions and their proneness to dos attacks. In *Proceedings of the 2024 ACM/IEEE 44th international conference on software engineering: new ideas and emerging results*, pp. 52–56, 2024a.

Mohammed Latif Siddiq, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. Understanding regular expression denial of service (redos): Insights from llm-generated regexes and developer forums. In *Proceedings of the 32nd IEEE/ACM international conference on program comprehension*, pp. 190–201, 2024b.

Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quanjun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. A survey of source code search: A 3-dimensional perspective. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–51, 2024.

N Thakur, N Reimers, A Rücklé, A Srivastava, and I Beir Gurevych. A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*, 2021.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pp. 8696–8708, 2021.

Orion Weller, Michael Boratko, Iftekhar Naim, and Jinhyuk Lee. On the theoretical limitations of embedding-based retrieval. In *International Conference on Learning Representations*, volume 2026, pp. 11202–11216, 2026.

Ke Xu, Siyang Xiao, Ming Liang, Yichen Yu, Zhixiang Wang, Jingxuan Xu, Dajun Chen, Wei Jiang, and Yong Li. Learning adaptive parallel execution for efficient code localization. *arXiv preprint arXiv:2601.19568*, 2026.

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

Fuwei Zhang, Yanzhao Zhang, Mingxin Li, Dingkun Long, Lexiang Hu, Pengjun Xie, Zhao Zhang, and Fuzhen Zhuang. Core-bench: A comprehensive benchmark for code retrieval in the era of agentic coding. *arXiv preprint arXiv:2606.11864*, 2026.

Xinyu Zhang, Nandan Thakur, Odunayo Ogundepo, Ehsan Kamalloo, David Alfonso-Hermelo, Xiaoguang Li, Qun Liu, Mehdi Rezagholizadeh, and Jimmy Lin. Miracl: A multilingual retrieval dataset covering 18 diverse languages. *Transactions of the Association for Computational Linguistics*, 11:1114–1131, 2023.

APPENDIX

A CASE STUDY

Case Study I: Vulnerability Discovery in Real World. We use CODERETRX as the retrieval stage of an automated vulnerability-discovery workflow: a task description characterizes a vulnerability class, CODERETRX retrieves the candidate functions across the repository, and a downstream analysis agent, followed by human review, examines each candidate in depth. Retrieval is where recall matters most in this pipeline, because a function that is never surfaced is never analyzed. Table 4 lists the vulnerabilities in cryptographic libraries that were found with this workflow and reported by the authors' team through coordinated disclosure. All nine received CVE identifiers and are rated High or Critical by the GitHub Advisory Database; they span five libraries in four ecosystems (Python, Go, JavaScript, and Rust), including `pyca/cryptography`, one of the most widely deployed cryptographic packages in the Python ecosystem. We do not claim that CODERETRX found these bugs unaided: confirmation required the downstream agent and expert review. Its contribution is that the vulnerable functions were among the retrieved candidates in every case, at a small fraction of the cost of reading each repository exhaustively.

A representative example is **gnark**, a zero-knowledge proof library developed by ConsenSys and widely used in blockchain ecosystems. We distilled a set of common vulnerability patterns; one of them is:

> *"This code defines a Zero-Knowledge (ZK) computational circuit, possibly using a framework like Circom, Noir, or Cairo. These circuits are the primary location for non-deterministic vulnerabilities, which are caused by missing constraints. They may enable multiple ways to create a valid proof for a certain outcome. Such vulnerabilities can allow for exploits like double-spending by enabling the generation of multiple valid nullifiers for a single commitment."*

Based on this description, CODERETRX retrieved **66** semantically related code snippets, two of which corresponded to the `Verify` functions in `eddsa.go` and `ecdsa.go`. Subsequent in-depth analysis by the vulnerability-discovery agent confirmed that both implementations exhibited the *signature malleability weakness* later disclosed as **CVE-2025-57801** and rated High (CVSS 8.6) in the public advisory. Reading the two functions costs a few thousand tokens; reading the repository exhaustively would cost orders of magnitude more, and the pattern above is only one of many run against each repository.

Cross-function consistency defects. The same workflow has produced two further vulnerability reports, both confirmed and rewarded through bug-bounty programs on production blockchain infrastructure: one in the peer-to-peer layer of a Bitcoin bridge, rated **High** by the project, and one in the cover-ownership logic of a decentralized insurance protocol, rated **Low**. Both are *cross-function consistency* defects, and together they illustrate why recall, rather than precision, is the binding constraint in this setting. In the bridge, the handler for incoming gossip messages never invoked the signature-verification method that the same codebase already defined and used elsewhere, so any network participant could impersonate a bridge operator. In the insurance protocol, three entry points each authorized an edit against one ownership token, while the mutation helper they all shared modified a different, freely transferable one. Neither defect is visible from any single function. The verification helper exists and is called correctly in other paths; each individual authorization check is locally sound. The inconsistency appears only once *every* member of a class, that is, every message handler, every edit entry point, and the helpers they share, has been retrieved and compared side by side. This is exactly the regime the rest of this paper is concerned with: a retriever that returns a plausible top-k but omits one call site omits the bug along with it. As above, CODERETRX supplied the candidate sites; the vulnerability analysis, exploit construction, and disclosure were carried out by the analysis agent and the authors.

Beyond cryptographic libraries. The same workflow is applied to other classes of software, with a different vulnerability-pattern set per class. Table 4 also lists the 14 further advisories it has produced to date in container runtimes, object-storage and file servers, and LLM-application frameworks; 12 carry CVE identifiers and 11 are rated High or Critical. This brings the workflow's public record to 23 advisories, of which 21 have CVE identifiers and 20 are rated High or Critical. As in the cryptographic cases, CODERETRX supplied the candidate functions and the analysis agent and the authors confirmed and disclosed each defect.

Case Study II: Post-Quantum Migration Audit As quantum computing rapidly advances, major cryptographers predict that RSA and ECC, the backbone of today's digital security, will become insecure within the next 5–15 years. Facing this challenge, large organizations must locate and migrate all cryptographic primitives in their legacy codebases to post-quantum algorithms, a process known as **post-quantum migration**.

However, enterprise-scale repositories often span hundreds of thousands of lines of code, with cryptographic logic scattered across layers of abstraction, from low-level OpenSSL wrappers to self-implemented ECC routines. Conventional static and semantic search tools are either **too coarse**, missing subtle cases such as hybrid key exchanges, or **too costly** to run at scale.

Using **CODERETRX**, we built an **economical migration scanner** capable of automatically identifying classical cryptographic implementations with **$>$90% recall** while consuming only **$\sim$10%** of the token cost of the current best approach. For example, when auditing **NeonDatabase (375,060 lines of code)**, a production-scale open-source repository, our scanner reduced analysis cost from **$4.6902 to $0.6583 per run**, yet achieved a **higher algorithm recall rate (86.7% vs. 85.7%)**.

```rust
pub fn generate_ssl_ca_cert(&self) -> anyhow::Result<()> {
    let cert_path = self.ssl_ca_cert_path().unwrap();
    let key_path = self.ssl_ca_key_path().unwrap();
    if !fs::exists(cert_path.as_path())? {
        generate_ssl_ca_cert(cert_path.as_path(),
          key_path.as_path())?;
    }
    Ok(())
}

pub enum AuthKeys {
    /// A `ClientKey` & `ServerKey` pair for `SCRAM-SHA-256`.
    ScramSha256(ScramKeys<32>),
}
```

Traditional search-based methods failed to retrieve a set of such functions, even though they are **critical to post-quantum migration**. Representative examples include the snippet below that defines a self-managed SSL CA certificate generator, and an enum that embeds authentication key derivation for `SCRAM-SHA-256`, a component that must be upgraded to a post-quantum key exchange scheme. These are illustrative; in our audit CODERETRX surfaced multiple additional sites of the same class that conventional tools missed.

This efficiency enabled **full-repository quantum-risk audits** across multiple enterprise codebases within a single compute budget, transforming what was once a **month-long manual review** into an **overnight automated migration-plan generator**. **CODERETRX** thus empowers security teams to **quantify and mitigate quantum-risk exposure early**, ensuring practical readiness long before quantum computers pose a real threat.

Case Study III: Comparison between CODERETRX and mainstream code agents: Cursor and Claude Code In recent years, most software engineers have adopted AI code agents to assist in software development. However, during the maintenance of moderately large repositories, we frequently observe discrepancies between the agents' implementations and our intended designs—primarily due to their insufficient understanding of repository-specific details.

To investigate this issue, we evaluated the code snippet retrieval capabilities of two leading commercial code agents, Cursor and Claude Code, both of which explicitly emphasize their codebase understanding capabilities on their official websites. But does this claim truly hold?

Our experiments reveal a substantial gap: while our proposed system CODERETRX, achieves a recall rate of **87.2%**, Cursor and Claude Code reach only **3.4%** and **2.3%**, respectively. As illustrated in Figure 5, regular expressions generated by LLMs, based solely on their pre-trained knowledge and general internet knowledge, fail to capture the majority of relevant keywords within a real-world project. (e.g., authenticate, certificate, ...) Consequently, these models lose critical contextual information, leading to inaccurate code generation or misunderstandings of developer intent in complex scenarios that require comprehensive repository-level reasoning.

Cursor's retrieval method on the Keycloak repository for the cryptographic functionality identification task.

```
Thought for 4s
I'll help you find all classes and functions related to
those categories in the Keycloak codebase.  Let me search
for each category.
Searched How are synchronous server calls implemented?  Where are
blocking HTTP requests or synchronous APIs used?
Searched Where are parsing functions implemented?  What code parses
JSON, XML, or other data formats?
Searched Where are HTTP server handlers implemented?  What classes
handle HTTP requests and responses?
Searched Where are initialization functions defined?  What methods
initialize components, services, or configurations?
Let me search for more specific instances of each category:
Searched Where are JAX-RS resources and endpoints defined?  What
classes handle REST API requests?  (services)
```

```
Searched What classes have init methods or initialization logic?
Where are factory init methods defined?
Searched Where are XML parsers implemented?  What code handles XML
parsing?  (saml-core)
Searched Where are JSON parsers used?  What code handles JSON
parsing and deserialization?, thought 3s
Let me continue searching for more specific instances:
Grepped @POST|@GET|@PUT|@DELETE|@PATCH in keycloak
Grepped public\s+(void\s+)?Init\( [type:java]
Grepped Parser|parse\( in saml-core
Grepped HttpClient|HttpURLConnection| CloseableHttpClient
[type:java], thought 3s
Now let me read a few more specific files to gather more complete
information:
Read AdminRoot.java L1-100
Read JOSEParser.java L1-100
Read SimpleHttp.java L1-150
Read DefaultHttpClientFactory.java L1-150
Thought for 6s
Based on my comprehensive search of the Keycloak codebase, here's
the JSON output with all the classes and functions related to the
specified categories:
{
  "codes": [
  {
    "code_type": "The code snippets implements, references or uses
    ↪  synchronous server calls",
    "locations": [
    {
      "name": "SimpleHttp",
      "type": "Class",
      "file_path": "server-spi-private/src/main/java/
      ↪  org/keycloak/broker/provider/ util/SimpleHttp.java",
      "chunk_id": "67-150",
      "id": "simple-http-deprecated"
    },
    {
      "..."
    }
  },
  {
    "..."
  }
  ]
}
```

Figure 5: Cursor's retrieval trace on the Keycloak repository for the cryptographic functionality identification task.

Case Study IV: Comparison between CODERETRX and LocAgent. LocAgent is a recent state-of-the-art LLM-based code localization system that adopts an agentic workflow to iteratively explore a repository and identify task-relevant code regions. (Chen et al., 2025) However, LocAgent currently supports only Python repositories and doesn't have a good performance on large repositories.

To better understand the practical gap between agentic localization and our approach, we compare LocAgent with CODERETRX on a set of real-world repositories. For fairness, quantitative comparison is restricted to Python projects where LocAgent is applicable. Ground truth is defined as the union of expert-validated relevant code regions obtained through exhaustive analysis.

Across all evaluated repositories, LocAgent achieves an overall average recall rate of only **14.35%**. This indicates that even under favorable conditions, the majority of relevant code locations are missed by agentic localization alone.

A per-repository analysis reveals pronounced scale-dependent degradation. On the small-scale repository `python-p2p`, LocAgent attains an average recall of **37.3%**, demonstrating that it can

partially localize relevant code when repository size and structural complexity are limited. However, recall drops sharply on medium-scale projects: for `magic-wormhole`, the average recall decreases to just **4.4%**. On the large, production-grade repository `zulip`, LocAgent's average recall further collapses to **1.4%**, effectively missing nearly all relevant code regions.

This behavior highlights a fundamental limitation of agentic, exploration-based localization. Once early-stage reasoning fails to visit a critical file or abstraction boundary, the missed search space cannot be recovered in later steps, leading to irreversible recall loss. In contrast, CODERETRX performs fragment-level embedding and verification prior to final filtering, ensuring near-exhaustive coverage and preventing premature pruning.

Also, LocAgent is limited to Python, whereas CODERETRX natively supports nearly all popular languages including Rust, C/C++, Go, etc. As a result, for the majority of security- and systems-critical repositories, LocAgent is fundamentally inapplicable, while CODERETRX remains fully effective.

Beyond effectiveness, the cost comparison is equally telling. Across all Python repositories in this study, CODERETRX incurred a total cost of **$2.95**, while LocAgent's measured LLM usage came to **$3.15** ($0.87, $1.04, and $1.23 on `python-p2p`, `magic-wormhole`, and `zulip`, respectively). At essentially the same budget, CODERETRX recalls over $6\times$ more of the relevant code. Notably, LocAgent's per-repository cost barely grows with repository size. Its exploration visits a roughly fixed number of locations regardless of how much relevant code exists, which is precisely why its recall collapses on large repositories.

Overall, this case study demonstrates that while LocAgent is a representative state-of-the-art agentic baseline, its recall ceiling, scale sensitivity, and language restriction significantly limit its real-world applicability. CODERETRX overcomes these limitations and provides a scalable, language-agnostic solution for reliable code localization in large, heterogeneous codebases.

B MOTIVATING EXAMPLE: MISSED FUNCTIONS

This appendix lists the concrete functions from the *anubis* repository that embedding-based retrieval and LLM-grep retrieval missed in the motivating example of Section 3.2. Listing 1 shows functions whose embeddings were poorly aligned with the task description, and Listing 2 shows functions that the LLM-grep agent never proposed search terms for.

Listing 1 Examples of relevant functions missed by embedding-based retrieval

```
/* File: cmd/anubis/main.go */
func main() {
    ...
    // Pass keys into server options
    ↪ (crypto-relevant: JWT signing
    ↪ materials)
    s, err :=
    ↪ libanubis.New(libanubis.Options{
        ...
        ED25519PrivateKey: ed25519Priv,
        HS512Secret:
        ↪ []byte(*hs512Secret),
        ...
    })
    ...
}
```

```
/* File: lib/thoth/thoth.go */
func New(ctx context.Context, thothURL,
↪ apiToken string, plaintext bool)
↪ (*Client, error) {
    ...
    if plaintext {
        do = append(do,
        ↪ grpc.WithTransportCredentials(
                        insecure.NewCredenti
                        ↪ als()
                ))
    } else {
            do = append(do, grpc.WithTranspo
            ↪ rtCredentials(
                        credentials.NewTLS
                        ↪ (&tls.Config{})
                ))
    }
    ...
}
```

Listing 2 Examples of relevant functions missed by LLM-grep retrieval

```go
/* File: lib/anubis.go */
func (s *Server) issueChallenge(ctx
↪  context.Context, r *http.Request, lg
↪  *slog.Logger, cr policy.CheckResult,
↪  rule *policy.Bot) (*challenge.Challenge,
↪  error) {
        ...
        id, err := uuid.NewV7()
        ...
        chall := challenge.Challenge{
                ID:          id.String(),
                Method:
                ↪  rule.Challenge.Algorithm,
                RandomData:
                ↪  fmt.Sprintf("%x",
                ↪  randomData),
        ...
        }
        ...
        return &chall, err
}
```

```ts
/* File: web/js/algorithms/fast.ts */
export default function process(
  options: ProcessOptions,
  data: string,
  difficulty: number = 5,
  signal: AbortSignal | null = null,
  progressCallback?: ProgressCallback,
  threads: number = Math.trunc(Math.max(getH
  ↪  ardwareConcurrency() / 2, 1)),
): Promise<string> {
  ...
  for (let i = 0; i < threads; i++) {
    let worker = new Worker(webWorkerURL);
    ...
    worker.postMessage({
      data,            // input data to be
      ↪  hashed
      difficulty,      // PoW difficulty
      nonce: i,        // nonce value
      threads,         // parallelism degree
      ↪  for hash search
    });
    ...
  }
}
```

C Extended Related Work

As modern codebases scale to millions of lines across thousands of repositories, function-level code search has become a critical capability for developers engaged in bug fixing, refactoring, and rapid code comprehension. Unlike file- or class-level retrieval, function-level search offers a precise and semantically meaningful unit of code reuse—developers often seek individual functions that encapsulate specific behaviors, rather than entire files or modules. Empirical studies show that tasks like fault localization, API migration, and onboarding are significantly accelerated when relevant functions can be located directly. Industrial deployments further demand that such systems operate across multiple languages, scale to billions of functions, and return low-latency, high-recall results—goals that remain difficult to achieve jointly in practice. In response, recent research has proposed diverse approaches to function-level retrieval, differing in how they represent, index, and match code. These are the current retrieval methods and the CST constructor we use.

Embedding-based code retrieval. Dense single-vector retrievers are attractive for efficiency but often struggle to achieve high recall on large, heterogeneous repositories. Code-pretrained models such as GraphCodeBERT (with data-flow signals) and UniXcoder (cross-modal objectives) improve snippet-level retrieval, yet repository-scale tasks with varied naming and cross-file dependencies still show recall gaps in practice (Guo et al., 2020; 2022; Feng et al., 2020; Wang et al., 2021; Gu et al., 2018; Sachdev et al., 2018). To address the issue that the top-k results from embedding retrieval are still not precise enough, some approaches apply late interaction and reranking to further reorder the retrieved embeddings. Late-interaction approaches (Khattab & Zaharia, 2020; Santhanam et al., 2022) score token-level representations with MaxSim, and cross-encoders further help as rerankers (Khattab & Zaharia, 2020; Santhanam et al., 2022). However, evidence from broad IR benchmarks (Thakur et al., 2021; Muennighoff et al., 2023; Zhang et al., 2023) and multilingual settings (MIRACL) indicates that no single method dominates across tasks and languages; stronger models typically incur larger index size, query latency, and serving cost (Thakur et al., 2021; Muennighoff et al., 2023; Zhang et al., 2023). Also, late interation and rerankering can't cannot make the retrieved top-k results more accurate once k is fixed. We compared our method with the embedding-based methods and found that both the recall rate and cost efficiency improved in table 2 and figure 4.

LLM-grep and regex pipelines. Keyword/regex pipelines remain popular in agentic coding workflows, but code search surveys consistently highlight vocabulary mismatch and ambiguity as limiting factors for purely lexical matching (Di Grazia & Pradel, 2023; Sun et al., 2024; Siddiq et al., 2024a;b). Moreover, recent studies show that LLM-generated regexes can be incorrect or prone to ReDoS, requiring careful validation (Siddiq et al., 2024b;a). As shown in the third case study, it is definitely infeasible for tasks that require a high recall rate.

LLM-based code localization. A rapidly growing line of work applies LLMs to *issue localization*: given a bug report or issue description, identify the code locations that must be edited. Agentic

systems explore the repository through tool calls: LocAgent (Chen et al., 2025) navigates a pre-built heterogeneous code graph, OrcaLoca (Yu et al., 2025) adds priority-based scheduling and distance-aware context pruning, CoSIL (Jiang et al., 2025) lets the LLM construct the search graph dynamically during reasoning, and FuseSearch (Xu et al., 2026) trains agents to parallelize tool calls for efficiency. SweRank (Reddy et al., 2026) instead replaces agentic exploration with a retrieve-and-rerank pipeline and reports better cost-effectiveness than agent-based localization; its multilingual successor SweRank+ (Reddy et al., 2025) couples the ranker with an agentic search loop and reports state-of-the-art localization across languages. Optimization for a few likely edit locations can leave relevant functions unexplored or below the candidate cutoff, limiting repository-wide recall even for state-of-the-art localization systems. Security auditing, feature migration, and compliance require retrieving *all* relevant functions; recall over the complete target set is therefore the binding metric, and retrieve-and-rerank front-ends inherit the candidate-cutoff limitation quantified in Section 6. Existing benchmarks share this structural mismatch with our task: code-search suites such as CodeSearchNet and CoIR (Husain et al., 2019; Li et al., 2025) pair each query with a (near-)single gold snippet and therefore measure ranking quality rather than completeness, while SWE-bench-derived localization sets define ground truth as the handful of functions touched by one fix patch and remain largely Python-centric. Even the concurrent CORE-Bench (Zhang et al., 2026), which itself documents a sharp performance drop when moving from traditional code-search benchmarks to realistic agentic retrieval, inherits such labels: its queries target code to be edited, not the complete set of functions satisfying a semantic property. None of them provides exhaustive, repository-wide target sets for complex queries across languages, which is why we construct the benchmark of Section 4.

Structure-aware parsing. We build on Tree-Sitter (Brunsfeld & contributors, 2018) to obtain concrete syntax trees (CSTs) and issue structure-aware queries across languages. Its incremental parsing and query DSL support our line-per-symbol indexing and candidate retrieval stages (Brunsfeld & contributors, 2018).

Why the three families fall short. The core challenge in retrieving specific functions lies in the **limitations of vector embeddings**. As the embedded context length increases, vector representations inevitably lose finer-grained semantic information. Although LLMs can accurately judge whether an entire function is relevant to a task, relying on them to exhaustively traverse all files would be **cost-prohibitive**, especially for large repositories. We systematically analyzed all the existing methods and analyzed that defects:

Current practice largely falls into three families: **(1) Embedding-based retrieval** (vector search over snippets/signatures) is cheap, but under the *single-vector* paradigm it faces a *capacity bound*: the family of top-k subsets realizable by a d-dimensional embedder is limited by d (via sign-rank), producing unavoidable recall gaps even for simple queries; these limits persist even with ideal training or direct test-time optimization ("free embeddings") and are corroborated on the LIMIT benchmark (Weller et al., 2026). This motivates structure-aware or late-interaction alternatives (e.g., GraphCodeBERT; ColBERT) that move beyond single-vector geometry (Guo et al., 2020; Khattab & Zaharia, 2020; Santhanam et al., 2022) for compositional code queries. However, late-interaction multi-vector and cross-encoder reranking do not resolve *cross-task/language/repository* generalization: on heterogeneous suites such as BEIR and MTEB/MMTEB, no single retriever consistently dominates (Thakur et al., 2021; Muennighoff et al., 2023; Enevoldsen et al., 2025; Zhang et al., 2023) across datasets and languages; multilingual IR remains uneven (MIRACL); and code-retrieval benchmarks (Lu et al., 2021; Diera et al., 2023; Li et al., 2025) (CodeXGLUE, GenCodeSearchNet, CoIR) report substantial variance across languages and repositories, with rerankers improving quality yet leaving persistent Out-of-Distribution recall gaps (Thakur et al., 2021; Muennighoff et al., 2023; Enevoldsen et al., 2025; Zhang et al., 2023; Lu et al., 2021; Diera et al., 2023; Li et al., 2025). **(2) "LLM-grep" pipelines** prompt an LLM to propose keywords/regex that are then executed by classical regex engines. Such lexical pipelines are attractive in agentic coding workflows due to their simplicity and tool integration (Dong et al., 2025). However, purely textual matching struggles with *semantic* variation and aliasing: code search surveys consistently report vocabulary mismatch and ambiguity in free-form/keyword queries, motivating structure-aware search beyond regex (Sun et al., 2024). In addition, automatically generated rules/regex introduce correctness and security pitfalls: LLM-generated regexes are frequently incorrect and can be prone to ReDoS vulnerabilities, requiring careful validation (Siddiq et al., 2024a;b). **(3) LLM traverse-all** feeds all of a repository to a model to avoid misses; however, token cost scales with repository size.

D Design Parameters

Every stage of CodeRetrX exposes a small number of parameters; each one is a knob on the recall–cost frontier, and we state explicitly why it must exist.

- **Per-unit retrieval width** k (default 10): the number of top fragments retrieved from each unit during unit-scoped search. It is necessary because a unit's relevance may rest on a few non-adjacent lines; k bounds how much evidence each unit may contribute. Too small and units with diffuse relevance never surface; too large and the candidate pool, and thus the judgment cost, grows linearly for no recall gain.
- **Round budget ratio** ρ (default 0.2): each judgment round examines at most $\lfloor(|L|/k) \cdot \rho\rfloor$ candidate units, where $|L|$ is the number of retrieved fragments. It is necessary to make early stopping meaningful: the round size determines the granularity at which the pipeline may stop, so ρ trades the number of LLM round-trips against the tokens wasted in the final, mostly irrelevant round.
- **Maximum judgment rounds** M (default $3k = 30$): bounds the number of coarse-recall rounds and the number of fragments judged per query. The round limit guarantees termination even when eligible candidates remain.
- **Stopping rule:** the line-per-symbol pipeline stops when no eligible fragments remain or the round limit is reached (Algorithm 1, lines 6–8). A round with no positive judgments does not itself terminate search; a different fragment from a previously rejected unit may still provide useful evidence.
- **Optional adaptive strategy:** the separate adaptive retrieval strategy starts from 10% of the collection and expands depth by a factor of 1.8 while the trailing batch has a hit ratio above 0.3, stopping after two low-quality rounds or five expansions. These controls govern that strategy, rather than the line-per-symbol loop in Algorithm 1.
- **Batching caps** (10^4 characters or 100 elements per call): bound each judgment call's context. They are necessary both for reliability, since relevance judgments degrade when too many candidates share one context window, and for keeping per-element output lengths, and hence costs, predictable.
- **Secondary recall** (off by default): an optional re-check of retained candidates with a stronger model to prune false positives. Its additional cost depends on the retained set.

These defaults bound judgment work but do not guarantee a minimum recall for every repository or task (Section 6.2); Appendix H details the implementation constants.

E Released Artifacts

We release: (i) repository and query datasets, (ii) retrieval indices and k settings, (iii) all prompting templates, and (iv) scripts for recomputing token and cost metrics. All artifacts are hosted in the repository.

(i) Repository & query datasets. For each target codebase we provide the repository identifier and commit hash, file paths, symbol spans (start–end lines). We also provide the natural-language queries. This enables exact regeneration of recall and coverage statistics without re-crawling.

(ii) Retrieval indices & k settings. We publish prebuilt script that create line-/symbol-level indices together with per-stage configurations (e.g., top-k per symbol, batching, early-stopping thresholds). Each index is paired with checksums and the source-repo commit for verifiable reuse.

(iii) Prompting templates. Templates for topic distillation, fragment relevance judgments, and full-function verification are provided in a machine-readable schema (inputs, task, requirements), so downstream users can reproduce or ablate prompt choices.

(iv) Token/cost scripts. Evaluation scripts recompute input/output token counts and dollar costs under our counting rules (batch grouping, one-line-per-symbol constraint), and report recall curves under the same settings used in the paper.

F Recall by Task Family

Table 5 reconstructs the task-family breakdown from the archived reports used in Figure 4. For repository r and family f, let $T_{r,f}$ and $S_{r,f}$ be the unions of reference and returned unit IDs across the family's queries. We compute $|S_{r,f} \cap T_{r,f}|/|T_{r,f}|$; this measures family-level coverage, so a

unit is counted once even if it matches multiple queries in that family. The family means in Table 3 average these ratios over repositories with $|T_{r,f}| > 0$. An empty reference set has undefined recall, shown as "–", and is excluded from that mean. Cost totals include all 18 repositories. The released analysis script reproduces all 36 CODERETRX and TextEmb3-L-30% recall bars to their displayed precision and records hashes of its source reports. It also computes the reference counts in Table 1, recall comparisons by family and size, and per-repository cost differences used in Section 6.2.

Table 5: Recall (%) by repository and task family. N is the number of distinct reference units, E30 is TextEmb3-L-30%, and CRX is CODERETRX.

	Vulnerabilities			**Refactoring**			**Crypto/PQ**		
Repository	N	E30	CRX	N	E30	CRX	N	E30	CRX
boringtun	8	75.0	100.0	99	69.7	100.0	167	59.9	90.4
rosenpass	57	61.4	96.5	425	68.0	79.3	603	54.7	96.8
neon	704	86.8	94.6	3,690	64.5	78.7	863	60.0	88.6
python-p2p	12	91.7	91.7	36	86.1	100.0	31	61.3	100.0
magic-wormhole	62	91.9	93.5	261	85.4	90.4	197	71.6	88.8
zulip	597	54.1	77.2	6,274	72.0	89.7	498	65.1	87.8
anubis	73	31.5	72.6	150	54.0	83.3	35	60.0	94.3
fscrypt	21	100.0	95.2	20	70.0	95.0	152	44.1	89.5
go-ethereum	206	42.2	81.1	1,372	55.0	78.3	3,269	40.6	75.8
client-encryption-java	8	62.5	100.0	179	66.5	82.1	317	53.6	98.7
cryptomator	126	92.9	81.7	469	71.4	85.5	366	73.8	86.1
keycloak	729	84.4	100.0	18,835	59.1	86.3	8,439	76.5	88.2
swifty	22	90.9	86.4	50	90.0	92.0	88	54.5	83.0
vaultwarden	42	92.9	100.0	698	53.9	87.8	232	74.6	94.8
padloc	64	95.3	87.5	642	78.2	89.4	672	71.4	97.6
inih	6	33.3	83.3	0	–	–	0	–	–
cJSON	9	55.6	77.8	71	53.5	88.7	0	–	–
llama.cpp	429	53.1	100.0	2,235	47.2	85.7	60	53.3	86.7

G EVALUATION DETAILS

This appendix expands the dataset protocol in Section 4 and the experimental setup in Section 6.

Datasets and tasks. We evaluate on the 18 repositories in CODERETRBENCH, grouped into C/C++, Rust, Go, Java, Python, and JavaScript/TypeScript; Table 2 lists them with their sizes and retrieval results. We categorize repositories into three size tiers based on their source lines of code: *Small* ($\leq$10K LOC), *Mid* (10K $<$ LOC $\leq$ 100K), and *Large* ($>$100K LOC). Each language–size cell contains one repository, so differences between language panels also reflect repository-specific effects. For each repository, we define a set of query targets corresponding to three representative complex retrieval scenarios:

1. **Vulnerabilities:** retrieving vulnerability-related or security-critical functions.
2. **Refactor:** retrieving all the related context for large-scale refactoring.
3. **Crypto Algorithms:** retrieving cryptographic or protocol algorithms for post-quantum migration.

Ground truth. The **LLM Traversal** method supplies the reference relevance labels. It presents every parsed function unit to the LLM for judgment against each task description, producing $\widehat{\mathcal{T}}(q)$ as defined in Section 4.2. Traversal has 100% recall against its own labels and supplies the reference token and monetary costs; exhaustive coverage of the parsed units does not eliminate possible annotation errors.

Snapshot alignment for SweRankEmbed. The archived main-result analysis identifies units by their recorded IDs. The separate SweRankEmbed evaluation matches traversal positives by symbol name and file path, then intersects these pairs with units present in the indexed snapshot. This comparison uses 43,596 aligned repository–family–unit entries, while Table 1 describes the 54,670 archived entries used for the main CODERETRX/Embedding comparison. Both counts deduplicate within a family and retain separate entries when a unit belongs to multiple families.

Practical retrieval methods. **CODERETRX (ours)** indexes the fragments of each function unit, applies unit-scoped retrieval, and uses the LLM for fragment judgments and full-function verification, aiming to approximate LLM Traversal at substantially lower cost. **Symbol Vector Embedding-k (baseline)** encodes each symbol into a vector with OpenAI *text-embedding-3-large* (TextEmb3-L), the same embedding model CODERETRX uses for fragments, returns the top-$k\%$ of candidates ($k \in \{10, 20, 30\}$), and lets the LLM read each candidate's full implementation to decide whether it satisfies the task description. **SweRankEmbed** supplies a retriever trained for issue localization; we evaluate its Small and Large variants with the snapshot alignment described in Section 4.2. The variants have 137M and 7B parameters, respectively; reported recall is averaged over the 18 repositories. SweRank+ (Reddy et al., 2025) and HyperFL were excluded because their retriever weights were unavailable for these experiments. **LocAgent** is evaluated on the three Python repositories it supports. The separate Cursor and Claude Code case study uses their LLM-generated keyword search workflows (Appendix A).

Experiment setup. All methods share the same indexing and parsing pipeline (repository snapshot, language parser, and symbol table extraction). The Symbol Vector Embedding baseline encodes identifier-level semantics only, while CODERETRX indexes every fragment within a function or class body, enabling precise structure-aware retrieval. We used **gpt-oss-120b** for dataset construction and evaluation.

Stage-wise statistics. The stage-wise breakdown in Section 6.3 uses a separate coarse-only run of the line-per-symbol stage (same parser, embeddings, and *gpt-oss-120b* judge) that records, for every repository–query pair, the units selected by fragment judgments and the candidate units handed to Refined Recall. Relevance labels are per-query exhaustive traversal labels from the same judge; pairs with an empty target set are skipped. Because Refined Recall applies the same full-body judgment as traversal, the candidate set's recall is the ceiling of the final recall. Token and cost shares are computed from the per-stage token counts at the *gpt-oss-120b* prices of Table 2. Precision and recall are averaged per query within a repository, then over repositories in a tier; token and cost shares pool all pairs.

Metrics. Metrics follow Section 4.3. For each query, a reference unit counts as **retrieved** if it appears in the method's returned set and as **missed** otherwise. We report **Recall Rate**, candidate precision (**Acc.**), input/output token counts, and estimated monetary cost. Token and monetary cost ratios normalize the corresponding method costs by those of LLM Traversal. The token ratio is $C(q)/C_{\mathrm{trav}}(q)$; USD costs apply separate input and output prices before normalization. For the archived main-result summaries, returned and reference unit IDs are deduplicated within each repository–task-family pair. Repository recall pools the recovered and reference counts across its three families before averaging over repositories. Table 3 instead averages each family's recall across repositories with nonempty reference sets; these family means therefore need not average to the overall 88.16%. Costs are summed across all runs, including families with empty reference sets.

H IMPLEMENTATION

We implement CODERETRX as a system with modular components for each stage of the workflow. The implementation consists of **over 15,000 lines** of Python code.

H.1 STATIC ANALYSIS MODULE

Tree-Sitter Integration. We utilize the `tree-sitter` Python bindings to parse source code. For each supported programming language, we maintain a set of tree-sitter grammar files that define language-specific syntax patterns. The parser produces concrete syntax trees that we traverse using language-specific query patterns written in S-expressions.

Chunk Extraction Queries. We define custom tree-sitter query patterns for each language to identify functions, classes, methods, and other relevant constructs. For example, in Python, our query captures function definitions, class definitions, and async function definitions. These patterns are stored in a query repository and loaded dynamically based on the detected language.

Symbol Data Structure. We represent each symbol using a structured data model containing:

- `name`: The symbol's identifier
- `type`: Classification as function or class
- `file`: Reference to the containing file

- `chunk`: The associated code chunk with position information
- `id`: A unique identifier for deduplication

Vector Embedding Generation. For fragment embeddings (individual source lines in our implementation), we use OpenAI's *text-embedding-3-large*; other pre-trained code embedding models can be substituted via configuration. Each line within a symbol is processed individually to generate a dense vector representation. We employ batched inference to process multiple lines efficiently, utilizing GPU acceleration when available. The embeddings are stored in a vector database for efficient similarity search.

H.2 Vector Database

We support multiple vector database backends including Chroma and Qdrant, selected via configuration. The database stores:

- **Vectors**: Embeddings for each code line
- **Metadata**: Associated information including symbol ID, file path, line number, and symbol name
- **Text**: Original code line content for retrieval

For the unit-scoped search required by our fragment-level strategy, we index metadata fields to enable filtering by symbol ID during query time. This allows us to efficiently retrieve the top-k lines within each symbol without exhaustively searching the entire database.

H.3 Coarse Recall Implementation

Topic Extraction. We implement topic extraction using a lightweight LLM call with a carefully designed prompt that instructs the model to identify key concepts, technical terms, and semantic patterns from the natural language query. The extracted topics are concatenated into a query string for vector search.

Symbol-Scoped Similarity Search. Algorithm 1, line 4, retrieves the top-k fragments within each unit and sorts their union by similarity. The index supports this operation through unit-ID metadata filters.

Batch Processing with LLM. After retrieving candidate lines, we use LLM-based batch verification to select relevant symbols:

Our `LinePerSymbolBatchProcessor` class manages dynamic batch evaluation. Key implementation details include:

Batch Size Calculation. The round size is $\max(1, \lfloor(|L|/k) \times 0.2\rfloor)$, where $|L|$ is the number of retrieved fragments and k is the per-unit retrieval width. Each round is split into LLM calls subject to the context and element caps in Appendix D.

Deduplication. A visited set permits at most one fragment per unit in each round. Additional fragments are deferred to later rounds (Algorithm 1, line 7).

Prompt Design. For each batch, we construct a prompt containing the original query and candidate snippets. The LLM judges relevance using a function-calling interface that returns structured decisions.

Dynamic LLM Batch Judgement Prompt

```
<input>
{code_elements}
</input>
<task>
Here are {code_element_number} code element(s) (i.e.
file/function/dependency/class/keyword/code_line etc.).
You need to analyse whether each code element meets this condition:
<requirement>
{task_description}
</requirement>
You should think step by step.
</task>
```

```
<output_format>
return a JSON Array which elements are JSON Objects that contain
index, result and reason.
</output_format>
```

Round Queue. The NEXTBATCH operation in Algorithm 1, line 7, scans the current queue until the batch is full or the queue is exhausted. It removes consumed entries, keeps deferred fragments from repeated units at the front of the next round, and retains the unconsumed tail. Selection state persists across rounds; the visited-unit set is reset each round.

Cost Tracking. We implement comprehensive cost tracking by logging all LLM API calls with their token counts. This enables fine-grained analysis of the trade-off between retrieval quality and computational expense.

H.4 REFINED RECALL IMPLEMENTATION

Symbol Retrieval. We resolve coarse-stage evidence into candidate units and load their complete definitions from the indexed codebase. Each definition includes the full function or class body, associated documentation strings, and any relevant context such as decorators or type annotations.

LLM Filter Mode. In filter mode, we employ function calling with a structured schema that requires the LLM to provide:

- `index`: Index of the evaluated code snippet for tracking
- `result`: Boolean indicating relevance
- `reason`: Textual explanation for the decision

The function calling approach ensures structured output that can be programmatically processed, avoiding issues with free-form text parsing.

LLM Map Mode. In map mode, the schema is customized based on the specific information extraction task. For example, when extracting security vulnerabilities, the schema might include fields for vulnerability type, severity, affected code location, and remediation suggestions.

Batched LLM Calls. To improve efficiency, we batch multiple symbols in a single LLM request when their combined token count fits within the model's context window. Our implementation uses a dynamic batching strategy that groups symbols up to a configured maximum.

All model selections can be overridden via configuration, and the system supports both OpenAI and Anthropic APIs, as well as locally hosted models via compatible endpoints.

H.5 LLM USAGE

The coarse stage uses LLM calls for topic extraction and fragment relevance judgments. The refined stage evaluates the complete implementations of candidate functions against the task description.

For specific model usage, we recommend using models with **function calling** capabilities, as we found that function calling yields more stable outputs. All models used in our experiments were `gpt-oss-120b`. However, for higher accuracy, one may use `claude-4-sonnet` or `gpt-4.1-mini` in the *refined recall* stage.

H.6 CACHING AND PERSISTENCE

Codebase Cache. After initial static analysis, we serialize the complete codebase representation (symbols, chunks, files, and metadata) to JSON format. Subsequent queries on the same repository load this cached representation, avoiding repeated parsing and analysis.

Vector Index Persistence. Vector embeddings and their associated indices are persisted to disk. For Chroma, this uses the built-in persistence mechanism. For Qdrant, we serialize the collection to disk using Qdrant's persistence functionality. This persistence enables fast startup times for repeated queries on large codebases.

LLM Response Cache. We implement an optional response cache for LLM calls, keyed by the hash of the prompt and model identifier. This cache significantly reduces costs during development and testing, and can also improve response times for repeated queries.

H.7 SCALABILITY CONSIDERATIONS

Large Repository Handling. We implement several optimizations for large codebases:

- **Incremental Indexing**: Vector embeddings are generated incrementally to cope with unexpected interrupted indexing problems.
- **Checkpoint-based Resume**: When running actual tasks, we recorded checkpoints with a resume option.
- **Chunk Size Limits**: We enforce maximum chunk sizes per file to prevent memory overflow on extremely large files.

Query Optimization. For the symbol-scoped vector search, we leverage database-level optimizations:

- Indexed metadata fields (in Qdrant) enable fast filtering by symbol attributes
- Approximate nearest neighbor search using HNSW indexing reduces query latency
- Cosine similarity with configurable top-k retrieval for efficient result ranking