Streamlining Repository Tasks with Effective Snippet Retrieval

TANGZHI XU*, Nanjing University & Ant Group, China
CONG LI*, Ant Group & ETH Zurich, Switzerland
ZHAOGUI XU, Ant Group, China
YANYAN JIANG, Nanjing University, China
YUAN YAO, Nanjing University, China
XIAORUI ZHU, Nanjing XiaoZhuang University, China
FENG XU, Nanjing University, China
PENG DI, Ant Group, China
CHANG XU, Nanjing University, China
ZHENDONG SU, ETH Zurich, Switzerland

Repository-level software engineering tasks are increasingly automated using repo-level retrieval-augmented generation (RLRAG), where a retriever selects relevant snippets from a repository to assist a language model (LM) in completing tasks. However, existing retrievers often lack effective designs to support LMs of varying capacities. To bridge this gap, we introduce RepoET, a novel retriever for RLRAG. RepoET organizes LMs and tools into an agentic workflow that closely mimics human search logic: "*search-files* → *filter-files* → *search-snippets* → *filter-snippets*". In our evaluation, RepoET outperformed state-of-the-art retrievers by accurately retrieving more relevant snippets in a better order across two widely used datasets, SWE-bench Lite and RepoQA, utilizing four LMs of different capabilities and sizes (as small as 3B). RepoET improved recall by over 12% and precision by over 20%, yielding a 21% improvement in Acc@k, along with superior snippet ordering. These retrieval improvements led to significant gains in downstream tasks: (1) we resolved 136 issues (45.33%) in SWE-bench Lite with GPT-4o without any additional information, and further improved the success rate by ~12% with multiple attempts; (2) we improved the accuracy of searching for designated functions by over 23% in RepoQA.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: RAG, language models, program repair, repository-level tasks

*Equal contribution

Authors' Contact Information: Tangzhi Xu, Nanjing University & Ant Group, Nanjing, China, xutz@smail.nju.edu.cn; Cong Li, Ant Group & ETH Zurich, Zurich, Switzerland, cong.li@inf.ethz.ch; Zhaogui Xu, Ant Group, Hangzhou, China, zhengrong.xzg@antgroup.com; Yanyan Jiang, Nanjing University, Nanjing, China, jyy@nju.edu.cn; Yuan Yao, Nanjing University, Nanjing, China, y.yao@nju.edu.cn; Xiaorui Zhu, Nanjing XiaoZhuang University, Nanjing, China, zhuxiaorui@njxzc.edu.cn; Feng Xu, Nanjing University, Nanjing, China, xf@nju.edu.cn; Peng Di, Ant Group, Hangzhou, China, dipeng.dp@antgroup.com; Chang Xu, Nanjing University, Nanjing, China, changxu@nju.edu.cn; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

ACM 1557-7392/2026/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Tangzhi Xu, Cong Li, Zhaogui Xu, Yanyan Jiang, Yuan Yao, Xiaorui Zhu, Feng Xu, Peng Di, Chang Xu, and Zhendong Su. 2026. Streamlining Repository Tasks with Effective Snippet Retrieval. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (May 2026), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software *repositories* are the primary environments for a wide range of software engineering activities. A repository comprises code written in programming languages, documentation in natural languages, and configurations specified in various formats. Historically, automating repository-level software engineering tasks (often referred to as repo-level tasks) has been difficult, as it requires a comprehensive semantic understanding of the entire repository. However, the emergence of large language models (LLMs) has made this more feasible, benefiting tasks such as question answering [12], code completion [8], and program repair [41, 49, 51].

To automate these tasks, *repo-level retrieval-augmented generation* (RLRAG) serves as the key technology. It leverages a retriever and a generator to address a user *query* specific to a repo-level task. The query may be a question, an issue description, or a code fragment requiring completion.[1] The retriever extracts a list of relevant file *snippets* — code, document, or configuration fragments such as [`repo/foo.md:15-32`, `repo/bar.py:102-231`] — from the repository. The generator, typically an LLM, incorporates the retrieved snippets as context to resolve the query.

An effective retriever is critical in RLRAG since the LLM-based generator lacks prior knowledge of the repository. In the absence of high-quality retrieval, even powerful generators may fail to produce useful output. However, retrieving the most relevant snippets is challenging due to the semantic gaps between queries and the repository: (1) User queries and repository content belong to two distinct semantic spaces: the question space and the solution space. Aligning these spaces is proven to be difficult [15, 46]. (2) Queries (e.g., questions, issues) are predominantly expressed in natural language, which can be vague and noisy, whereas the repository primarily consists of code where answers may span distant modules. This further complicates the "translation" of user intent into relevant snippets.

To address these challenges, existing retrievers fall into four categories. *IR-based retrievers* [11, 20, 40] utilize standard information retrieval (IR) techniques that rely on query-snippet similarities [14]. *Agent-based retrievers* [1, 3, 10, 41, 51, 54] implement autonomous LLM-driven agents using tool-call capabilities [35], adhering to the ReAct paradigm [52]. *Agentic workflows* [4, 21, 37, 49] define structured, tool-assisted search flows, mirroring human search logic "files → classes/methods → snippets". *Finetuning-based retrievers* use models fine-tuned from repo-level data [9, 32, 39, 44].

Despite these advancements, to the best of our knowledge, no existing retriever effectively supports LLMs with limited reasoning capabilities or small language models (SLMs). While larger-scale repo-level fine-tuning is feasible, it incurs substantial data collection and model training costs, often beyond the reach of individuals and small teams. Yet, we argue that accommodating SLMs is particularly important today, as LLMs can be costly or inaccessible [30, 34], and are often impractical to deploy in environments constrained by security, computational, or financial limitations — such as industrial systems or edge devices. On the other hand, certain environments emphasize the importance of auditable retrievers to enhance observability, enable better control, and support modular tuning.

RepoET. To bridge the gap, we present a novel retriever called RepoET that organizes language models (LMs) and tools into an agentic workflow, mimicking human search logic: *"search then filter"*,

[1] We assume that users are familiar with the repository, so that their queries align with it.

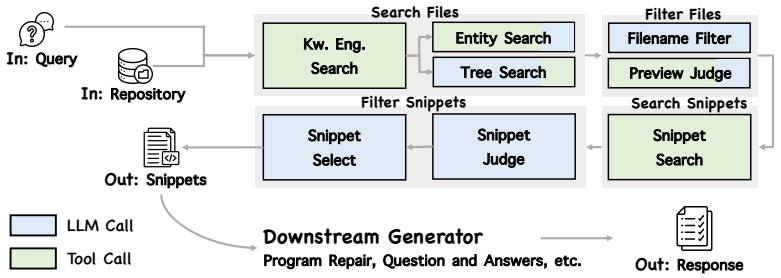


Fig. 1. RepoET's agentic workflow mimics human search logic: "search-files → filter-files → search-snippets → filter-snippets". Snippets retrieved by RepoET can be adapted for various generators. Node green: tool calls; Node blue: LM calls; Edge →: execution flow (rather than input/output data flow).

specifically, *"search-files → filter-files → search-snippets → filter-snippets"*. For the heavy-weight, more difficult searching process, we primarily utilize IR-based tools to perform coarse-grained searches, while leveraging LMs to complement the results. In terms of the lightweight, less difficult filtering process, we predominantly rely on LMs for fine-grained filtering, while using tools to construct the necessary context. This design leverages the strengths of LLMs while remaining friendly to SLMs, which excel at small-scale extraction and decision-making but struggle to identify correct answers within large-scale repositories. Furthermore, the workflow architecture ensures that intermediate decisions remain transparent and auditable, while allowing tools or LMs to be independently upgraded or replaced. Given a query, RepoET works as follows (Figure 1):

(1) *Search files*. We initially identify a list of potentially relevant files using an IR-based tool, specifically a keyword engine. To enhance this list, we allow LMs to recognize code entities (e.g., functions and classes) mentioned in the query and to analyze a reduced repository tree to include files that might be overlooked by the engine.
(2) *Filter files*. We then filter out irrelevant files incorrectly included in the previous stage. We employ LMs to judge their relevance to addressing the query based on file names and file previews that sketch their main content. Files that are judged irrelevant are discarded.
(3) *Search snippets*. Next, we obtain all snippets of each identified file stored in the keyword engine, expanding each snippet with additional lines of context above and below.
(4) *Filter snippets*. Finally, we filter out irrelevant snippets by examining their content using LMs. We select and rank the most relevant snippets to produce the final output.

Evaluation. Like other agentic workflows [49], our workflow is conceptually simple. However, we demonstrate that our organization of LMs and tools is significantly more effective for accommodating LMs — both large (LLMs) and small (SLMs) — with varied capabilities. Empirically, RepoET consistently retrieved more relevant snippets in a better order than state-of-the-art baselines, including the IR-based engine FAISS [40], the agentic workflow Agentless [49], and the agent-based method LocAgent [10]. This advantage holds across different LMs, including GPT-4o, DeepSeek-V3, Qwen-2.5-Coder-7B, and Qwen-2.5-Coder-3B. Specifically, our evaluations are based on two widely used repo-level datasets: SWE-bench Lite [22] and RepoQA [25]. With GPT-4o, RepoET achieved over 12% higher recall and over 20% higher precision, yielding a 21% improvement in Acc@k, along with superior snippet ordering. For other LMs, RepoET consistently outperformed competing methods by more than 13% in recall and 8% in Acc@k across multiple evaluation levels. Our performance on smaller LMs degrades smoothly — by significantly less than a quarter — unlike existing methods, which exhibit a sharp decline of over a half.

This superior retrieval capability directly translated into substantial gains in downstream repo-level tasks. When using snippets retrieved by RepoET as input to GPT-4o-based generators, we observed the following results:

(1) *Issue Repair*: On SWE-bench Lite, our multi-turn repair pipeline resolved 45.33% (136/300) of issues, making it one of the most competitive issue-repair agents based on GPT-4o. Remarkably, even when using SLMs as small as 7B for retrieval, RepoET-retrieved snippets enabled GPT-4o to outperform both Agentless and LocAgent using GPT-4o-based retrievers.
(2) *Function Search*: On RepoQA, our approach yielded a significant accuracy increase of over 23%, outperforming state-of-the-art techniques across six widely used programming languages.

We obtained these results by executing RepoET only once per repository using greedy decoding. This contrasts with existing retrievers that rely on high-temperature LLM sampling with large beam sizes for autonomous searches [31, 41, 54], or that merge multiple samples from LLMs [37, 49]. The generators operate simply by prompting the LLM (GPT-4o in our experiments) with the benchmark query and the snippets retrieved by RepoET, without any additional information such as test feedback or regression tests.

Novelty and Contribution. To the best of our knowledge, we are the *first effective* RLRAG retriever that *focuses on LM flexibility* without requiring LM fine-tuning. RepoET introduces a novel agentic workflow that keeps LM decisions local while delegating heavy exploration to tools. This workflow is controllable, auditable, and compatible with LMs of varying sizes — from lightweight models to state-of-the-art LLMs. This makes RepoET especially suitable for real-world, resource-constrained environments where specific requirements persist, even as frontier LLMs become more capable and cost-effective: (1) security, privacy, and offline settings where API calls are prohibited, making local deployment of small models the only viable option; (2) production systems with strict demands for predictable budgets and low tail latency; and (3) retrieval systems that prioritize explainability, auditability, and modular tuning. In summary, our contributions include:

- *Novel Workflow.* We present RepoET, a retriever with a novel agentic workflow compatible with LMs of various sizes and applicable to different repo-level software engineering tasks.
- *Thorough Evaluation.* We demonstrate RepoET's effectiveness and flexibility using two popular datasets and four different LMs, including models as small as 3B. Our evaluations show that RepoET achieves significantly higher recall, precision, and accuracy.
- *Real-world Evidence.* We highlight RepoET's utility and generality by applying retrieved snippets to two different repo-level tasks: resolving issues in SWE-bench Lite and searching for functions in RepoQA.
- *Open-source Tool.* RepoET is publicly available at https://github.com/SoftWiser-group/RepoET.

2 RepoET's Agentic Workflow

RepoET (Figure 1) orchestrates LMs and tools across four stages: *Search Files*, *Filter Files*, *Search Snippets*, and *Filter Snippets*. Each stage consists of one or two workflow nodes that emulate how humans search for relevant snippets within a repository. In this section, we describe RepoET's implementation of the four stages through an illustrative example, django__django-11848[2] (Figure 2). The example comes from SWE-bench Lite [22], a widely used program repair benchmark consisting of real-world GitHub issues. The issue states that `django.utils.http.parse_http_date` does not account for whether a two-digit year is 50 years beyond the current year, potentially resulting in an incorrect full-digit year, in violation of RFC 7231. To repair this issue, a retriever must correctly locate the buggy function (*ground-truth* or GT function) `parse_http_date` inside the buggy file

[2]https://code.djangoproject.com/ticket/28690

```
Title: django.utils.http.parse_http_date
two digit year check is incorrect
Description: (last modified by ...)

RFC 850 does not mention this, but in RFC
7231 (and there's something similar in RFC
2822), there's the following quote:
Recipients of a timestamp value in
rfc850-date format, which uses a two-digit
year, MUST interpret a timestamp that
appears to be more than 50 years in the
future as representing the most recent
year in the past that had the same last
two digits. Current logic is hard coded to
consider 0-69 to be in 2000-2069, and
70-99 to be 1970-1999, instead of
comparing versus the current year.
```

(a) Title and Body

```
def parse_http_date(...):
  yr = int(m.group('year'))
  if yr < 100:
-   if yr < 70:
-     yr += 2000
+   cyr = utc_year
+   cen = cyr-(cyr % 100)
+   if yr - (cyr % 100) > 50:
+     yr += cen - 100
  else:
-   yr += 1900
+   yr += cen
  mth = MONTHS.index(...) + 1
  day = int(m.group('day'))


# Recipe from django/utils/http.py
```

(b) Golden Patch

```
def parse_http_date(...):
  yr = int(m.group('year'))
  if yr < 100:
-   if yr < 70:
-     yr += 2000
+   rst = utc_year % 100
+   cen = (utc_year//100) * 100
+   if yr > rst + 50:
+     yr += cen - 100
  else:
-   yr += 1900
+   yr += cen
  mth = MONTHS.index(...) + 1
  day = int(m.group('day'))


# Recipe from django/utils/http.py
```

(c) SWELL-Generated Patch

Fig. 2. Illustrative Example: Issue django__django-11848 from SWE-bench Lite. This issue's title and body are used verbatim as the query for RepoET's snippet retrieval. The patch generated by SWELL, while textually different, is semantically identical to the golden patch. SWELL generated the patch within five attempts using snippets retrieved by RepoET as context.

(*GT file*) `django/utils/http.py`. Specifically, the GT snippets are [`django/utils/http.py:158-191`]. For this example, the input to RepoET is the full issue description (title and body as shown in Figure 2a) and the Django repository at revision `f0adf3b9`. The output of RepoET is an array of snippets relevant to fixing the issue.

2.1 Search Files

We begin with a coarse-grained search for potentially relevant files. Since it is impractical for LMs to comprehend an entire repository, we delegate this sub-task primarily to an IR-based search engine. We then enhance the results by using LMs to identify code entities (such as functions and classes) mentioned in the query and to analyze a reduced repository tree. This process approximates finding the *use*, *def*inition, and *dep*endency files. The two keys to this process are a *keyword engine* and a *query summary*.

Keyword Engine. We chose an IR-based engine based on the following observations: (1) While IR-based engines may lack accuracy in retrieved snippets, they can recall a moderate portion of GT files; (2) The files retrieved by search engines are often located in the same neighborhood as the GT files in the repository tree. We made these observations in a preliminary study (Appendix A) involving 45 snippet retrieval tasks across 23 GitHub repositories. The study used 45 pairs of commit messages and their corresponding code diffs, randomly selected from a state-of-the-art dataset called HQCM [24]. We used commit messages as queries and considered all edited files as GT files. We established two engines to retrieve files in their repositories: a keyword engines and a vector store. As a result, both recalled more than 45% of the GT files with a precision of around 14%. For over 70% of the missed GT files, we found that their parent directory and the directory of a retrieved file shared a common ancestor (e.g., a great-grandparent). These results align with a best practice in software engineering: functionally similar code should be grouped within the same module or in closely related modules, with module proximity often indicated by their locations in the repository tree.

In contrast to existing works that rely heavily on vector stores, RepoET uses a keyword engine. This is inspired by product database searches [13]: queries for repo-level tasks typically involve code entities, such as classes and functions, which can be uniquely identified by their names

throughout the repository. In this context, a keyword engine can establish a lexical connection between code entities and files. Unlike natural-language words, which can have synonyms or multiple meanings (polysemy) best captured by vector stores, code entities have unique names. Our preliminary study validated this: the keyword engine recalled 25% more files than the vector store (71.11% vs. 46.67%) while maintaining comparable precision (14.22% vs. 14.52%). We find these results reasonable, as snippets from vector stores are often functionally similar to the query (falling into the question space) rather than the solution space. However, most snippets obtained through the keyword engine *use* the relevant entities, which is often more critical for addressing the query.

Based on our preliminary study, we build a keyword engine for the repository. Specifically, we chunk the repository into snippets using `CodeSplitter` and `SentenceSplitter` from Sweep [43] and LlamaIndex [28] for code and text files, respectively. We then create an inverted index by tokenizing all snippets and mapping each token to a list of snippets containing it.

Query Summary. A summary distills the query's essence, eliminating extraneous details that could interfere with coarse-grained file searching. A concise summary helps the keyword engine and other LM-based nodes focus on relevant code entities. We distill the query using the evaluator-optimizer pattern for building agentic workflows [4]. Specifically, we use an LM to: (1) generate an initial summary, (2) evaluate its quality, (3) update it, and (4) iterate until the LM is satisfied.

A good summary must meet the following criteria: (1) It is natural and concise (under 40 words), ideally capturing the query's requirements in one or two sentences; (2) It helps locate relevant files in the codebase that should be reviewed or modified. `RepoET` encodes these criteria in a zero-shot CoT [47] prompt for each step of the "summarize → evaluate → update" process.

"*Modify django.utils.http.parse_http_date to dynamically interpret two-digit years based on the current year as per RFC 7231, replacing the hardcoded year ranges*," GPT-4o generated this concise summary for our illustrative example, removing for example authors and redundant descriptions.

All subsequent file searching and filtering nodes operate on the summary instead of the query.

Node: Keyword Engine Search. We start by searching the keyword engine using the summary. The engine identifies the most similar snippets based on their BM25 similarity ($k_1 = 1.2$ and $b = 0.75$). We assign each file the highest BM25 score found among its snippets. This prevents relevant files from being overshadowed by files with many snippets that have lower individual scores but a higher collective average. We retain the top 25 files to form the initial list of candidates. Essentially, these files are likely related to *use*s of involved entities, ranked by their BM25 scores.

In our example, the retrieved files include:[`django/middleware/http.py`, `django/utils/cache.py`, `django/utils/http.py`, `tests/utils_tests/test_http.py`, `django/views/static.py`, ...]. The keyword engine successfully found the GT file `django/utils/http.py`, which was ranked third.

Node: Entity Search. In repo-level tasks, queries often include code entities (e.g., module, class, or function names like `parse_http_date`) that users are interested in. An experience is that files that define these entities are often instrumental in addressing the query. However, a keyword engine search may overlook these *def*inition files if they are overwhelmed by files that *use* the entities frequently. Therefore, we use LMs and a fuzzy search tool to locate definition files in two steps: inferring file names and performing the search.

We first use LMs to infer possible file names. The LM is instructed to: (1) identify code entities in the summary, (2) analyze their names, and (3) infer the files that likely define them. The insight is that LMs, trained over a substantial amount of general and code data, understand common naming conventions. For example, an LM correctly translates `django.utils.http.parse_http_date` to `http.py`, noting that "*...modules are typically represented by a single file, and since the module is named http, the corresponding file is likely http.py.*"

We then search the repository for files whose names match the inferred names, either literally or fuzzily. Since there can be many matches, we use LMs again to identify the most likely candidates based on their paths and entity names. The identified paths must be relevant to the summary according to the LM. These files are prepended to the candidate list. For example, from over 20 files matching "http", the LM successfully identified the GT file `django/utils/http.py`.

Node: Tree Search. Humans often browse the repository tree to find *dep*endent files by their names and locations, which encode semantics like package dependencies and containment relationships. While using the repository tree is common [41, 49, 51], existing methods struggle with large repositories where the tree exceeds the LM's context window. We provide the LM with a tool-constructed *sub-tree* that ideally contains all GT files.

To build the sub-tree, our tool ascends the repository tree two layers from the previously identified relevant files, creating a fake root to connect great-grandparent directories. This is based on our observation that IR-based engines often retrieve files in the same neighborhood as GT files. The LM analyzes the sub-tree to identify files that might help address the summary. The LM outputs the most relevant files over two iterations, and these are added to our list.

In the example, `RepoET` constructed a sub-tree of 3,092 entries for `django/*` and `tests/*`. In contrast, the full tree has 9,096 entries, including many irrelevant files (e.g., `docs/*`) that could hinder the LM's analysis. As a result, the LM correctly identified `django/utils/http.py` as relevant.

Discussions. Identifying uses, definitions, and dependencies could also be done using language servers or static analyzers. However, deploying these tools can be costly and requires integration for many different languages. Furthermore, repo-level analysis can be time-consuming for large repositories, and these tools struggle with typos in entity names. In contrast, general-purpose repo-level tasks often only require *semantically* relevant files rather than precise dependencies. We found that integrating IR-based tools with LMs is effective for this purpose.

2.2 Filter Files

Node: Filename Filter. The files identified so far are based on their names and tree locations relative to the summary. However, identifying relevant files without examining their content can lead to inaccuracies. Since reading an entire file can exceed an LM's context window or introduce long, LM-untractable information, we first filter candidates based on their filenames and then provide a compact preview for each file.

Specifically, we provide the LM with the user query and the names of all files in the repository. The LM analyzes the relationships between files, their relevance to the query, and the repository structure to identify a reduced set of up to 10 filenames that are most likely to be relevant.

Node: Preview Judge. We then use a tool to generate a small *preview* of each candidate file.

- For code files, we provide a hierarchical outline showing imports, global variables, function headers, and class definitions (including fields and method headers). We maintain indentation to help the LM understand structural relationships.
- For text files, we identify the first sentence of each paragraph as the key sentence. We divide the file content into paragraphs and connect the first sentences using ellipses.

For each file, we provide the LM with its preview and the summary to assign a relevance score:

- Score 0: Completely irrelevant.
- Score 1: Weakly relevant; the summary can be addressed without it.
- Score 2: Relevant; the summary can only be partially addressed without it.
- Score 3: Strongly relevant; the summary cannot be addressed without it.

We re-rank the files based on these scores and retain only those with a score ≥ 2. For files with identical scores, we maintain their original order. Since each preview is assessed independently, we use multi-processing for parallelization [4].

In our example, the LM assigned a significantly higher score to the GT file `django/utils/http.py` than to others, leading to their removal. For instance, `django/middleware/http.py` was removed because: "*The query is about modifying the function parse_http_date in 'django.utils.http'... This file contains middleware classes related to HTTP processing and does not relate to ... parse_http_date.*"

2.3 Search Snippets

For LLMs with long context windows and strong reasoning abilities, the list of relevant files (e.g., [`django/utils/http.py`]) may be sufficient. To provide more concise context for the remaining LMs, we further extract the most relevant snippets. At this stage, we use the full query instead of the summary to avoid information loss when analyzing specific snippet content.

Node: Snippet Search. Since we chunked the repository when building the keyword engine, we can directly gather all snippets from each identified file. We organize these candidate snippets by their file order and starting line numbers. For each snippet, we add extra lines of context above and below to maintain continuity.

2.4 Filter Snippets

We filter out irrelevant snippets by examining the content of each candidate snippet using LMs. We optionally re-rank the snippets to produce the final list.

Node: Snippet Judge. This node operates similarly to the preview judge, assigning relevance scores from 0 to 3, indicating completely irrelevant, weakly relevant, relevant, and strongly relevant. We remove snippets with scores ≤ 1. If retained snippets are adjacent, we merge them and assign the highest score among the original. For example, `/repo/src/foo.py:1-100` (score: 2) and `/repo/src/foo.py:100-200` (score: 3) will be merged into `/repo/src/foo.py:1-200` (score: 3). Finally, we re-sort the snippets, maintaining the original file order for those with the same score.

In our example, the LM removed all snippets of `django/utils/http.py` except for lines `1-89` and `158-253`, with the latter closely surrounding the GT snippet at lines `158-191`.

Node: Snippet Select. We then apply a two-step selection procedure. The first step focuses on the top five snippets, while the second handles the rest. In each step, the LM selects up to five snippets deemed truly relevant. We merge the results of both steps to form the final retrieved set. This node typically requires a more powerful, long-context LM because of the complexity and length of the combined snippets. In our example, the final snippets are [`django/utils/http.py:158-253`, `django/utils/http.py:1-89`].

2.5 Discussions

RepoET's workflow decomposes retrieval into manageable sub-tasks such as inferring filenames, analyzing sub-trees, and assessing previews and snippets. These tasks are significantly less complex than generating full retrieval plans or comprehending entire repositories, making them accessible to LMs of various sizes. The workflow is also general and does not include task-specific nodes. For example, it does not generate tests to reproduce issues or perform iterative dependency tracking for code completion. To connect the design with practical usage, we conclude this section with two notes: one showing how retrieved context is consumed downstream, and another showing how later nodes can recover the GT file even when coarse retrieval misses it.

Issue Repair Generator. We use the issue-repair case as a concrete example of how retrieved snippets are used. With snippets retrieved by RepoET, generators can adapt the context (e.g., extracting full functions or classes containing them) to their needs. For our illustrative example (Figure 2a), our repair agent SWELL generated a correct patch (Figure 2c). Although textually different, the patch is semantically identical to the golden patch (Figure 2b) provided by Django developers. Both patches determine if a two-digit yr is 50 years beyond the current year rst (Line 8) and calculate the full-digit year accordingly. They also account for an arbitrary century rather than limiting to years around the 20th and 21st centuries (Lines 4–5 & 11–12). We provide the implementation details of SWELL in Section 5.4 as it falls beyond this paper's scope – streamlining repo-level tasks via effective snippet retrieval.

When Keyword Engine Fails. We now present a case illustrating why subsequent nodes are necessary even when keyword matching is imperfect. In practice, the keyword engine may miss GT files because it relies on lexical similarity. However, it often returns neighboring files that help later nodes recover the GT files. Consider the issue django__django−14855 with the summary: "*Investigate and potentially revise 'get_admin_url' in 'django.contrib.admin.helpers' to correctly generate URLs for readonly ForeignKey fields in custom Admin Sites by using 'current_app' parameter in 'reverse' function.*" In this case, the keyword engine failed to identify the GT file `django/contrib/admin/helpers.py`. However, it found `django/contrib/admin/checks.py` in the same directory, allowing the sub-tree search to succeed. Furthermore, the summary helped the entity search node identify the GT file. Both nodes successfully recalled the GT file, and subsequent nodes filtered the candidates to include the correct segment `django/contrib/admin/helpers.py:209-216`.

3 Implementation Details

We describe the implementation of the various tools used by RepoET. Many of these tools represent standard community practices rather than new inventions. RepoET's primary contribution is its agentic workflow (Figure 1), which effectively organizes these tools to support and integrate various LMs. The workflow architecture ensures that these tools are observable and auditable, and their implementations can be seamlessly upgraded or replaced to allow for future improvements.

3.1 Keyword Engine Tool

We implement a trigram keyword engine. RepoET chunks the codebase into snippets and creates an inverted index that maps each token to a list of snippets.

Chunking into Snippets. Existing works typically chunk a file into snippets using a fixed number of characters, words, or lines. While this method works for text files, in code files, it often divides a continuous, semantically self-contained code block or even a code element (e.g., a string) into multiple snippets. To alleviate this issue, RepoET's chunking algorithm for code files operates on abstract syntax trees (ASTs); this is inspired by Sweep [43]. RepoET utilizes tree-sitter to parse code files into ASTs due to its native support for multiple mainstream programming languages. Additionally, tree-sitter offers flexibility for extension, allowing new languages to be supported by simply integrating the corresponding tree-sitter grammars. Specifically, for each code file, the algorithm traverses its AST to determine whether each AST node should be chunked: (1) If a node is small enough (smaller than 1,500 words according to [43]), it retains the node in the current snippet; (2) Otherwise, it treats the node as belonging to the next snippet. The algorithm also merges overly small snippets into one of its adjacent snippets as they are less meaningful. As for text files and files with programming languages yet not supported, RepoET falls back to the traditional line-based method. In such cases, line-based chunking ensures adequate coverage for retrieval when tree boundaries are unavailable. This choice is common in RLRAG systems such as Sweep and

LangChain [23]. We believe this is a reasonable choice according to the locality and naturalness of software [18]. Therefore, RepoET followed this established practice. Currently, RepoET excludes binary files from its considerations.

Creating Inverted Index. After chunking, RepoET tokenizes the snippets and maps each token to its respective snippet in an inverted index. Unlike modern LM tokenizers that use byte-pair encoding, we use a classical trigram tokenizer. This tokenizer: (1) treats each snippet as a sequence of whitespace-separated tokens; (2) splits camelCase and snake_case tokens; (3) includes unigrams, bigrams, and trigrams as tokens; and (4) outputs case-insensitive tokens. For example, tokenizing "myGood friend" results in: [my, good, friend, my_good, good_friend, my_good_friend].

3.2 Entity Search Tool

Searching by Entities. RepoET searches for files by computing the similarity between file names and entities extracted from the query. We use two types of fuzzy similarity: global similarity and local similarity. Global similarity computes a normalized score based on the Levenshtein distance. Local similarity evaluates how well a shorter string aligns with the most similar substring of a longer one. In particular, RepoET prioritizes file names that contain the target string, ranking them by global similarity to form the initial candidate list. It then adds the most similar remaining names, followed by those with strong local similarity to improve recall on incomplete or noisy queries. The merged list is truncated to produce the final set of candidates. We utilize the `rapidfuzz` library to calculate both global (i.e., `ratio()`) and local similarity (i.e., `partial_ratio()`).

3.3 Repository Sub-Tree Tool

Constructing Sub-tree. RepoET creates the sub-tree as follows. Starting from the retrieved files, RepoET ascends the file tree by two layers (`2-up`) to construct a sub-tree that includes all (direct and indirect) files. We use `2-up` as the default because `1-up` may miss nearby dependencies, while deeper expansion introduces more irrelevant branches and noise. This choice is an empirical trade-off based on our experiments (Appendix B and Section 5.3) rather than a theoretical optimum. Specifically, for each file, RepoET first records its k-indirect parent directory. For instance, the 0-indirect parent directory of `path/to/my/file.txt` is `path/to/my/`. Next, RepoET removes any directories that are children of another recorded directory. It then retrieves the entire sub-tree rooted at each recorded directory. Finally, RepoET creates a fake root to connect these sub-trees if there is more than one.

3.4 File Preview Tool

Generating Preview. For code files, RepoET generates previews using their ASTs. Starting from an empty preview and the AST's root node, RepoET performs a breadth-first traversal: (1) If a node exceeds a line-count threshold, it is traversed recursively; (2) If a node is smaller than the threshold, the preview includes all its contents; (3) For intermediate nodes, the preview retains only the first and last lines, omitting the rest. This typically generates a preview that includes the modifiers, names, and arguments of functions, classes, and methods while removing the implementation details. For text files or unsupported languages, RepoET splits the content into paragraphs and connects their first sentences using ellipses. This preview strategy provides compact summaries, while snippet construction for these files still follows the line-based fallback described above.

4 Experimental Setup

We evaluated RepoET through the following research questions:

RQ1 *Effectiveness: Proprietary LLMs.* Can RepoET effectively retrieve snippets from real-world repositories when supported by leading proprietary LLMs? How does it compare with the state-of-the-art retrievers?

RQ2 *Effectiveness: Other LMs.* Is RepoET still effective when supported by other open-source LMs (large or small)?

RQ3 *Stepwise Analysis.* How do the various steps contribute to RepoET's overall performance? Is the keyword engine fundamental? What improvements does LM-based refinement offer?

RQ4 *Case Study: Issue Repair*. How effective is integrating RepoET-retrieved snippets for resolving repo-level issues?

RQ5 *Case Study: Repository Q&A.* How effective is integrating RepoET-retrieved snippets for answering repo-level questions?

4.1 Evaluation Datasets

Since we were unaware of any dataset specifically designed for snippet retrievers, we used two datasets from real-world repo-level tasks: SWE-bench Lite [22] for issue repair and RepoQA [25] for question answering.

SWE-bench Lite. SWE-bench Lite (or SWE-bench in this paper) is widely used to evaluate software engineering agents on real-world GitHub issues. It consists of 300 issues spanning 12 popular Python repositories. Resolving these issues requires editing an average of 1.7 files, 3.0 functions, and 32.8 lines, with the most complex cases involving up to 31 files. For each issue, we used the `problem_statement` (the issue title and body) as the query and the `base_commit` (the buggy revision) as the repository. We parsed the `patch` (the golden patch provided by developers) to extract unchanged and removed lines; these formed our GT snippets. We did not use information from `hints_text` (developer discussions) or `FAIL/PASS_TO_PASS` (regression tests), as these could provide unfair guidance on which files to retrieve.

RepoQA. The RepoQA dataset includes 600 repo-level Q&A instances across 60 repositories in six programming languages (Python, Java, C++, Go, TypeScript, and Rust). Each question requires searching for a specific function (the *needle function*) based on a natural language description. For each instance, we used the question as the query. We defined GT snippets using all lines of the needle function. We did not use `code_context` as it provides hints for identifying the function.

4.2 Evaluation Metrics

For the upstream retrieval tasks (RQ1–3), we used two metrics: MPRF (Macro Precision, Recall, and F-score) and Acc@k. For the downstream generation tasks (RQ4–5), we used the standard metrics for each dataset: the ratio of resolved issues and the accuracy of retrieved needle functions.

MPRF. This metric is order-insensitive. For evaluating the retrieval performance in terms of snippets, our calculation for precision and recall was unit-based on lines. We mark lines as true positives if they are included in GT snippets and false positives otherwise; lines in GT snippets yet not retrieved were false negatives. In addition to F1, we also considered F2 that weights recall twice as much as precision. This adjustment is in line with existing works that focuses merely on "% Correct Location" [41, 49] (i.e., recall). After calculating these metrics for each instance in a dataset, we averaged them to represent the overall effectiveness on the dataset. In addition to evaluating the MPRF in the retrieval granularity of snippets, we also considered that at the granularity of retrieved functions, as well as files where the retrieved snippets reside. Calculating MPRF for these two granules is straightforward and we omit the details.

Acc@k. This is an order-sensitive metric popularly used in existing work [10, 49]. Specifically, for each instance, we selected the top-k results and deem it a success if all GTs are accurately

identified within the top-k results. This metric captures the ability to precisely pinpoint all snippets that require modification, balancing the recall rate and the accuracy of the ranking. Our evaluation spans multiple k values, including Acc@1, Acc@3, Acc@5, and Acc@10, at the retrieval granularity of snippets, functions, and files.

4.3 Selected Retrievers

We selected three state-of-the-art retrievers:

- A widely used search engine FAISS [40]. We configured FAISS to perform vector-based retrieval with Jina embeddings (`jina-embeddings-v2-base-code`)[3].
- An agentic workflow Agentless [49]: We selected Agentless as it employs a similar yet different (See Appendix B) workflow to ours and its adoption by OpenAI [36]. We used Agentless v1.5.0 for comparison.
- An agent-based retriever LocAgent [10]: We chose LocAgent because it achieved the state-of-the-art performance in the repo-level code localization and retrieval to date.

To ensure multilingual support, we replaced Agentless' file previewer with ours (Section 3) for non-Python projects. We could not do the same for LocAgent, as it is specifically designed for Python. Therefore, we used the Python subset of RepoQA when evaluating LocAgent for a fair comparison.

4.4 Selected Models

We selected language models to achieve an optimal balance between their capability and cost efficiency. For LLMs, we utilized the propriertary LLM GPT-4o and open-source LLM DeepSeek-V3, accessed through paid APIs. At the time of our experiment, these were cost-effective LLMs leading LM leader boards such as [29], with a remarkable user reputation. For SLMs, we chose Qwen-2.5-Coder-7B and Qwen-2.5-Coder-3B; we deployed them locally on an NVIDIA GeForce RTX 3090 GPU – a consumer-grade GPU.

4.5 Research Question Setup

While we acknowledge the existence of other promising retrievers such as SpecRover [41], datasets like SWE-Gym [38], and LMs such as the Claude series, budget constraints is the major factor in limiting our selection. Our choices are informed by existing research and benchmarks, with an emphasis on the balance of cost-efficiency and capability at the time of conducting the experiments. Furthermore, we could not assess `RepoET` and all chosen retrievers across every selected dataset and LM. As a result:

- RQ1 (Section 5.1): We evaluated all selected retrievers (FAISS, Agentless, LocAgent, and `RepoET`) over SWE-bench and RepoQA using the proprietary LLM GPT-4o.
- RQ2 (Section 5.2): This involves the other three selected open-source LMs (DeepSeek-V3, Qwen-2.5-Coder-7B, and Qwen-2.5-Coder-3B) and the three LM-based retrievers. We only used SWE-bench due to its popularity and significance. It has become a de facto benchmark for evaluating capabilities in software engineering tasks; almost all prominent LMs, including the GPT, Qwen, and DeepSeek series, incorporate it in their performance evaluations, to the best of our knowledge. We excluded FAISS in this experiment because it is not reliant on LMs.
- RQ3 (Section 5.3): We analyzed the node-wise result of `RepoET` on SWE-bench Lite when using the proprietary LLM GPT-4o, based on the result of RQ1. This ablation analysis aimed to quantify the contribution of each tool and assess whether each node offers tangible improvements in recall, precision, or ranking quality.

[3]It achieved state-of-the-art results on various code search benchmarks [2, 17].

Table 1. Effectiveness of RepoET (with GPT-4o) compared to baseline methods.

Dataset	Granularity	Tool	Recall	Precision	F1	F2	Acc@1	Acc@3	Acc@5	Acc@10
SWE-bench Lite	File	FAISS	0.687	0.128	0.211	0.152	0.267	0.503	0.573	0.687
		Agentless	0.813	0.411	0.524	0.656	0.697	0.750	0.800	0.810
		LocAgent	**0.940**	0.340	0.465	0.641	0.737	**0.900**	**0.933**	**0.940**
		RepoET	0.873	**0.451**	**0.558**	**0.691**	**0.743**	0.850	0.867	0.873
	Function	FAISS	0.438	0.013	0.025	0.057	0.058	0.101	0.146	0.217
		Agentless	0.628	0.042	0.075	0.144	0.304	0.393	0.465	0.524
		LocAgent	0.713	0.144	0.218	0.345	0.477	0.630	0.678	0.705
		RepoET	**0.755**	**0.304**	**0.400**	**0.529**	**0.506**	**0.712**	**0.739**	**0.745**
	Snippet	FAISS	0.442	0.008	0.015	0.010	0.119	0.268	0.341	0.442
		Agentless	0.660	0.023	0.041	0.084	0.515	0.614	0.651	0.659
		LocAgent	0.740	0.029	0.043	0.079	**0.598**	0.725	0.739	0.740
		RepoET	**0.808**	**0.037**	**0.065**	**0.128**	0.586	**0.765**	**0.791**	**0.804**
RepoQA (Python)	File	FAISS	**0.850**	0.437	0.555	0.475	0.640	**0.850**	**0.850**	**0.850**
		Agentless	0.460	0.392	0.412	0.434	0.460	0.460	0.460	0.460
		LocAgent	0.610	0.355	0.418	0.494	0.300	0.590	0.600	0.610
		RepoET	0.790	**0.738**	**0.755**	**0.772**	**0.780**	0.790	0.790	0.790
	Function	FAISS	0.680	0.050	0.090	0.061	0.430	0.680	0.680	0.680
		Agentless	0.430	0.287	0.315	0.354	0.410	0.430	0.430	0.430
		LocAgent	0.580	0.340	0.390	0.452	0.260	0.560	0.580	0.580
		RepoET	**0.780**	**0.591**	**0.633**	**0.688**	**0.720**	**0.780**	**0.780**	**0.780**
RepoQA (All)	File	FAISS	**0.783**	0.417	0.523	0.451	0.633	**0.783**	**0.783**	**0.783**
		Agentless	0.419	0.353	0.373	0.395	0.414	0.419	0.419	0.419
		RepoET	0.780	**0.706**	**0.729**	**0.753**	**0.770**	0.780	0.780	0.780
	Function	FAISS	0.648	0.048	0.085	0.058	0.442	0.648	0.648	0.648
		Agentless	0.385	0.257	0.284	0.319	0.358	0.385	0.385	0.385
		RepoET	**0.760**	**0.549**	**0.595**	**0.654**	**0.722**	**0.760**	**0.760**	**0.760**

- RQ4–5 (Sections 5.4 and 5.5): We connected the selected retrievers with GPT-4o-based generators to perform the respective downstream generation tasks. The generators operate simply by calling an LM with a prompt that combines the query and the retrieved snippets. For RQ5, we excluded LocAgent since it is specifically designed for Python, not generalize to other programming languages.

Finally, we enabled greedy decoding whenever using RepoET to try to ensure deterministic retrieval in all RQs, i.e., `temperature = 0` and `do_sample = false` when using the transformers [19] library. For generators in RQ4–5, we enabled sampling and set the temperature to 0.8 to introduce a level of diversity; for these cases, we reported their average results over three independent runs.

5 Experimental Results

5.1 RQ1: Effectiveness with Proprietary LLMs

Tables 1 present the evaluation results across two datasets and multiple granularities. Overall, RepoET demonstrates superior performance across most metrics. It achieves the best F2 in every setting, with gains of 5% to 36% over the second-best baseline; this also applies to the F1 score. For Acc@1, it ranks #1 in 6 out of the 7 settings while remaining highly competitive on Acc@3 and above. Note that, for the RepoQA dataset, snippet-granule metrics are meaningless and we did not include them because the task aims at finding the relevant function based on a description.

For this research question, we observe that the primary strength of RepoET lies in its significant improvement in precision. On average, RepoET improves the retrieval precision by 22%, up to 38%. It might be noted that the absolute precision at the snippet granule tends to be low across methods on the SWE-bench dataset. This is because our snippet-level supervision is based on lines, which is significantly sparser compared to functions or files at the other two granules for the SWE-bench dataset. Although our recall at the file granularity is occasionally lower than some baselines, this shallow disadvantage is always justified by the precision, crucial for the overall effectiveness. As a result, the F2 improvements over the strongest baseline in SWE-bench Lite at the granularities of the file, function, and snippet are 5%, 4% and 18%, respectively; on RepoQA, F2's gains are even more pronounced, ranging from 23% to 34% across evaluated settings. It is also worth noting that at finer granularities, RepoET not only maintains higher precision but also demonstrates recall advantages. We believe that such results indicate the effectiveness of our retrieval approach, especially in more detailed retrieval granularities. We also observed that FAISS's recall is higher or close to that of RepoET and always higher than the other baselines in RepoQA. This is because the description of a needle function contains the function's documentation, resulting in considerably higher cosine similarity.

The ranking quality, measured by Acc@k metrics, reveals RepoET's strength in positioning relevant results at top ranks. On SWE-bench, RepoET achieves the highest Acc@1 scores for the file and function granularities and competitive performance for snippets. Similar to F2, the advantage becomes more pronounced on RepoQA: RepoET achieved Acc@1 scores of 0.77 and 0.722, representing improvements of 14% and 28% over the strongest baselines respectively. We believe that this superior ranking capability is particularly valuable for LLM-based downstream tasks, where top-ranked results receive disproportionate attention due to context length limitations and reasoning capabilities. RepoET's ability to consistently place ground-truth items in top positions (as evidenced by strong Acc@1 performance) ensures that relevant code snippets are readily available for generation and reasoning tasks. Regarding other Acc@k metrics, LocAgent excells at file granularity on SWE-bench, likely due to its sophisticated code graph; FAISS outperforms the others at file granularity on RepoQA due to the reasons explained above. Despite this, RepoET remains the most competitive tool on average, particularly at finer granularities.

The combination of high precision and superior ranking makes RepoET especially suitable for real-world applications. By retrieving fewer but more relevant code snippets at finer granularities, RepoET enables the composition of shorter yet more precise contexts for LLM-based generation. This approach reduces noise while maintaining comprehensive coverage of relevant functionality, leading to more accurate and efficient code generation as demonstrated in our downstream evaluation (i.e., RQ4–5).

5.2 RQ2: Effectiveness with Open-Source LMs

Table 2 reports the retrieval performance of all retrievers on the three open-source LMs. The general trend is that stronger LMs yield higher scores for every approach. Nevertheless, with the same LM setting, RepoET consistently surpasses Agentless and LocAgent in all Acc@k metrics, while remaining competitive (i.e., ranking #1 on 7 out of 9 settings) on F2. Unlike RQ1, RepoET does not remain the higher precision advantage at the snippet granularity for models evaluated in this research question: Agentless (and LocAgent) may outperform RepoET in terms of precision. In fact, RepoET excels in recalling a higher number of relevant results (as reflected in the "recall" metric), as well as in positioning the most relevant results at top ranks (as indicated by "Acc@k" metrics). Aligning with previous work [10, 41, 49], we argue that recall is more important than precision for LMs. since downstream LLMs, having no prior knowledge, can be considerably less effective

Table 2. Effectiveness of RepoET with various open-source LMs. "QC-2.5-xB" denotes Qwen-2.5-Coder-xB.

Granularity	Model	Tool	Recall	Precision	F1	F2	Acc@1	Acc@3	Acc@5	Acc@10
File	QC-2.5-3B	Agentless	0.310	0.295	0.299	0.304	0.307	0.307	0.307	0.310
		LocAgent	0.108	0.044	0.057	0.075	0.024	0.096	0.108	0.108
		RepoET	**0.583**	**0.393**	**0.450**	**0.512**	**0.443**	**0.567**	**0.577**	**0.583**
	QC-2.5-7B	Agentless	0.467	0.343	0.379	0.420	0.440	0.467	0.467	0.467
		LocAgent	0.390	0.168	0.213	0.272	0.197	0.343	0.362	0.371
		RepoET	**0.730**	**0.491**	**0.560**	**0.638**	**0.640**	**0.713**	**0.723**	**0.730**
	DeepSeek-V3	Agentless	0.667	**0.507**	0.559	0.612	0.617	0.653	0.660	0.667
		LocAgent	0.695	0.229	0.315	0.442	0.599	0.671	0.681	0.695
		RepoET	**0.823**	0.493	**0.579**	**0.683**	**0.717**	**0.807**	**0.817**	**0.823**
Function	QC-2.5-3B	Agentless	0.150	0.081	0.098	0.120	0.079	0.136	0.150	0.150
		LocAgent	0.044	0.024	0.028	0.036	0.004	0.034	0.044	0.044
		RepoET	**0.376**	**0.175**	**0.218**	**0.276**	**0.269**	**0.352**	**0.367**	**0.373**
	QC-2.5-7B	Agentless	0.286	0.114	0.149	0.199	0.091	0.235	0.262	0.286
		LocAgent	0.178	0.084	0.113	0.157	0.071	0.135	0.151	0.168
		RepoET	**0.585**	**0.283**	**0.347**	**0.436**	**0.434**	**0.568**	**0.581**	**0.585**
	DeepSeek-V3	Agentless	0.505	0.137	0.197	0.289	0.123	0.388	0.445	0.493
		LocAgent	0.558	0.157	0.236	0.361	0.321	0.438	0.525	0.554
		RepoET	**0.675**	**0.294**	**0.370**	**0.477**	**0.473**	**0.660**	**0.670**	**0.673**
Snippet	QC-2.5-3B	Agentless	0.152	**0.052**	**0.066**	**0.088**	0.129	0.142	0.142	0.145
		LocAgent	0.044	0.018	0.021	0.026	0.004	0.036	0.044	0.044
		RepoET	**0.408**	0.023	0.040	0.078	**0.241**	**0.380**	**0.405**	**0.408**
	QC-2.5-7B	Agentless	0.306	**0.055**	**0.078**	0.119	0.254	0.303	0.304	0.306
		LocAgent	0.187	0.043	0.057	0.083	0.073	0.144	0.158	0.180
		RepoET	**0.603**	0.039	0.068	**0.132**	**0.426**	**0.576**	**0.596**	**0.603**
	DeepSeek-V3	Agentless	0.505	**0.078**	**0.124**	**0.207**	0.483	0.504	0.505	0.505
		LocAgent	0.558	0.050	0.072	0.117	0.383	0.463	0.548	0.558
		RepoET	**0.733**	0.041	0.071	0.138	**0.551**	**0.698**	**0.723**	**0.733**

at the downstream generation if fewer GTs are provided. Therefore, we believe that these results also confirmed RepoET's generality to varying model capacities.

It is noteworthy that RepoET maintains competitive performance with even smaller models such as Qwen-2.5-Coder-3B, while Agentless and LocAgent degrade sharply. For example, when switching the LM from DeepSeek-V3 to Qwen-2.5-Coder-7B, the performance of the baselines diminished by nearly or more than half, whereas RepoET maintained reductions of less than a quarter. Manual inspection suggests that Agentless suffers from SLM's inability to accurately identify the relevant files in a large repository, whereas LocAgent is limited by weak function-call reliability, on which its interaction-heavy pipeline depends. By contrast, RepoET organizes LMs and tools in a more reasonable way. This not only leads to less challenging sub-tasks, but avoids complex autonomous tool invocation. We believe these findings indicate that RepoET not only alleviates the performance floor imposed by small models but also scales smoothly with stronger LMs – achieving a better balance across model capacities.

5.3 RQ3: Stepwise Analysis

Stepwise Analysis. The results are shown in Table 3. Despite its simplicity, the keyword engine alone recovers 79.3% of ground-truth files, providing a strong initial recall baseline. Although not

Table 3. Effectiveness of RepoET's workflow nodes for file and snippet retrieval on SWE-bench Lite using GPT-4o. Metrics are reported at the file or snippet granularity based on the node. "Filter Files" refers to the results after *"filter-files"* in *"search-files → filter-files → search-snippets → filter-snippets"*; these results are calculated by treating all filtered files as single-snippet candidates.

Granularity	Workflow Node	Recall	Precision	F1	F2	Acc@1	Acc@3	Acc@5	Acc@10
File	**Kw. Eng. Search**	0.793	0.032	0.061	0.137	0.243	0.470	0.530	0.657
	Entity Search	0.867	0.033	0.063	0.142	0.517	0.657	0.717	0.770
	Tree Search	**0.890**	0.033	0.064	0.145	0.517	0.657	0.717	0.773
	Filename Filter	0.880	0.089	0.161	0.315	0.620	0.790	0.843	**0.880**
	Preview Judge	0.873	0.230	0.345	0.521	0.653	0.817	0.857	0.873
	Snippet Judge	0.873	0.236	0.353	0.528	0.693	0.8	0.867	0.873
	Snippet Select	0.873	**0.451**	**0.558**	**0.691**	**0.743**	**0.85**	**0.867**	0.873
Snippet	**Filter Files**	**0.873**	0.005	0.009	0.022	0.653	0.817	0.857	0.873
	Snippet Judge	0.859	0.012	0.022	0.049	0.494	0.691	0.761	**0.823**
	Snippet Select	0.808	**0.037**	**0.065**	**0.128**	**0.586**	**0.765**	**0.791**	0.804

presented in the paper, its effectiveness is pronounced further with open-source LMs: Across various SLMs, we observed that the keyword engine contributes over 90% of the final recall at the file granularity. This makes the keyword engine a robust and essential node in RepoET.

Each successive node in the workflow contributes either to improved recall, precision, or result ranking. In particular: (1) The entity search and tree search nodes offer complementary benefits to the keyword engine, recovering an additional 10% of ground-truth files. (2) The nodes of the filename filter and preview judge significantly improve precision by 170% on average via eliminating irrelevant files. This significantly reduces the number of files that followup nodes needs to process. (3) The final snippet selection offers the highest precision among all file-level nodes, and improves Acc@1 to 0.743 and Acc@3 to 0.85, without sacrificing recall.

Among all nodes, the entity search contributes the most to recall, while the preview judge and the snippet selection contribute most to precision and F scores, confirming the effectiveness of RepoET's LM-based refinements. Compared with autonomous search, our sub-tasks such as inferring file names and exploring sub-trees are less challenging and can be better handled by various LMs. Intriguingly, at the file granularity, the snippet judge node yields only marginal improvements in precision, whereas the snippet select node leads to a substantial increase. This is mainly because GPT-4o may assign relatively high scores to many isolated snippets during judgment, making it hard to reject noise early. However, when snippets are re-evaluated collectively during re-ranking, the model can better identify false positives and down-rank irrelevant content.

In summary, this stepwise analysis confirms that each module in RepoET contributes meaningful improvements in either recall coverage, ranking quality, or filtering precision. These results validate our design principle: Reasonable organization of LMs and tools – combining high-recall lightweight retrieval with targeted LM reasoning modules – yields both robust coverage and fine-grained precision.

Keyword Engine Performance. We further analyzed the performance of the keyword engine regarding different types of user queries. We categorized user queries into four disjoint groups in decreasing order of priority, using the ground-truth (GT) file paths and code entities as anchors: (**G0**) Queries that directly reference GT file paths. (**G1**) Queries that explicitly mention at least three code entities; (**G2**) Queries that reference fewer than three code entities; and (**G3**) Queries

Table 4. Keyword engine performance across different query types on SWE-bench Lite.

Group/Type	#Instances	Recall@5	Recall@10	Recall@20	Recall@25
G0 (GT file path mentioned)	48	0.583	0.688	0.813	0.854
G1 (≥3 code entities mentioned)	145	0.545	0.669	0.786	0.814
G2 (<3 code entities mentioned)	57	0.491	0.632	0.772	0.772
G3 (the remaining queries)	50	0.500	0.580	0.700	0.720

Table 5. Performance of the Tree Search node for various configurations denoted by `L-up-I-it`: `up` refers to the number of ascending layers and `it` indicates the number of iterations, where $\texttt{L} \in [1, 2, 3] \wedge \texttt{I} \in [1, 2]$. "*" marks RepoET's default configuration.

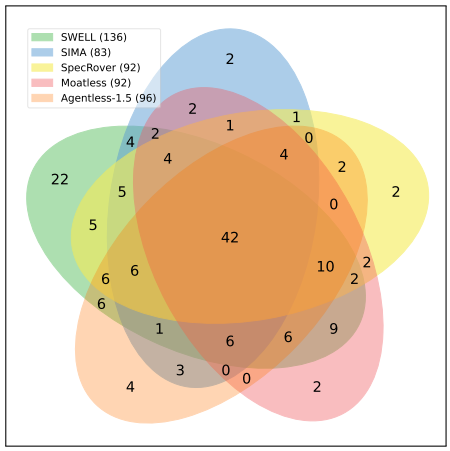
Granule	Config.	Recall	Precision	F1	F2	Acc@1	Acc@3	Acc@5	Acc@10
File	1-up-1-it	0.816	0.490	0.574	0.677	0.699	0.801	0.810	0.816
	2-up-1-it	0.821	0.494	0.578	0.681	0.697	0.806	0.815	0.821
	3-up-1-it	0.831	0.503	0.589	0.692	0.716	0.819	0.825	0.831
	1-up-2-it	0.824	0.498	0.582	0.684	0.699	0.812	0.818	0.824
	2-up-2-it*	0.823	0.493	0.579	0.683	0.717	0.807	0.817	0.823
	3-up-2-it	0.835	0.495	0.585	0.692	0.697	0.819	0.829	0.835
Snippet	1-up-1-it	0.731	0.039	0.069	0.136	0.544	0.697	0.718	0.731
	2-up-1-it	0.733	0.040	0.070	0.136	0.528	0.697	0.714	0.733
	3-up-1-it	0.742	0.043	0.074	0.143	0.546	0.717	0.730	0.742
	1-up-2-it	0.737	0.041	0.072	0.140	0.544	0.706	0.731	0.737
	2-up-2-it*	0.733	0.041	0.071	0.138	0.551	0.698	0.723	0.733
	3-up-2-it	0.740	0.042	0.072	0.139	0.531	0.713	0.736	0.740

that do not meet the criteria for the above groups. Groups assigned higher priority indicate the presence of more robust anchors (e.g., direct mentions of file paths or a significant number of GT code entities). If a query satisfies the criteria for multiple groups, it is assigned to the group with the highest priority. Using this classification, we evaluated the performance of our keyword engine. Specifically, we ran the Keyword Engine Search node, leveraging the SWE-bench Lite dataset, which contains a total of 300 user queries. For this experiment, we measured *recall@k*—the recall of the top-k retrieved results—at the file granularity. These results (Table 4) show a clear anchor-strength trend: queries with more robust lexical anchors such as G0 and G1 achieve higher recall, while weaker anchor queries are harder for keyword-only retrieval, and would thus benefit more from later LM-assisted nodes.

Tree Search Performance. We also analyzed how the Tree Search node performs regarding different tree-search parameters: the number of ascending layers (i.e., `up`) and the number of iterations (i.e., `it`). Each configuration is denoted as `L-up-I-it` where $\texttt{L} \in [1, 2, 3] \wedge \texttt{I} \in [1, 2]$. All configurations were evaluated with the same benchmark as in Appendix A and with DeepSeek V3 (temperature = 0) to find relevant files and snippets. We reported retrieval metrics in the file and snippet granules, respectively. `RepoET`'s default setting is `2-up-2-it` (marked by "*"). The result is displayed in Table 5. Overall, among the evaluated configurations, larger values of `up` and `it` generally—though not always—lead to improved results. However, the differences in performance remain relatively minor. Additionally, Appendix A demonstrates that `2-up` ascending, when applied based on the results of the Keyword Engine Search node, covers a significantly broader file

Techniques	LLMs	#Resolved
RAG	GPT-4	8 (2.67%)
SWE-agent	GPT-4o	55 (18.33%)
AppMap Navie	GPT-4o	65 (21.67%)
Agentless 1.0	GPT-4o	82 (27.33%)
SIMA	GPT-4o	83 (27.67%)
SpecRover	GPT-4o	92 (30.67%)
Moatless Tools	DeepSeek-V3	92 (30.67%)
CodeShellTester	GPT-4o	94 (31.33%)
Agentless 1.5	GPT-4o	96 (32.00%)
SWELL	GPT-4o	**98 (32.67%)**
	DeepSeek-V3	**101 (33.67%)**
	GPT-4o (Multi)	**133 (44.33%)**
	DeepSeek-V3 (Multi)	**136 (45.33%)**

(a) All Repaired Issues with Different LMs



(b) Uniquely Repaired Issues with DeepSeek-V3

Fig. 3. Performance of SWELL and state-of-the-art techniques using GPT-4o or DeepSeek-V3 on SWE-bench Lite. Baseline data are from the leaderboard. "(Multi)" denotes SWELL results using ≤10 attempts per issue.

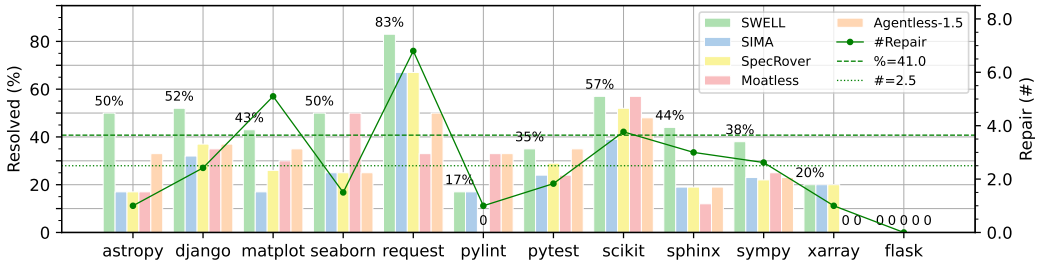


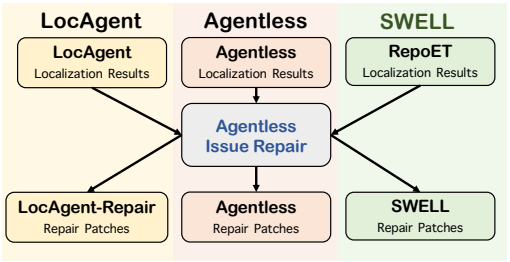
Fig. 4. Repo-level Resolved% (bar chart) and #Repair (line chart) for SWELL and other techniques on SWE-bench Lite. "%=41.0" represents SWELL's repo-level average for Resolved%, while "#=2.5" is for #Repair.

neighborhood compared to vector-only retrieval, without causing a substantial increase in token usage. This supports the choice of `2-up` as the default in the main pipeline. We therefore frame `2-up-2-it` as a practical default of RepoET (rather than a theoretical optimum).

5.4 RQ4: Case Study - Issue Repair

Generator. Upon RepoET-retrieved snippets, we prompt an LLM to generate a plausible patch for the issue. For consistency, we utilized GPT-4o and DeepSeek-V3 as the repairing model in conjunction with the Agentless repairing method.[4] We test whether the patch passes all regressions within SWE-bench Lite. If successful, we output the patch and stop. Note that generating patches using multiple attempts is a common practice [6, 49], so we repeat up to 10 times until a successful patch is generated. If all attempts fail, we report a repair failure. For retrieval, we directly input RQ1's and RQ2's results without re-execution as RepoET is deterministic. We denote the resulting system as SWELL (an integration of RepoET with the above repair process).

[4]We chose to use the repair component of Agentless 1.0 instead of 1.5 because the newer version's built-in test-feedback loops can make it harder to see how well the retrieval component works.



(a) Workflow of SWELL and Baselines

Tool	Ret.	Rep.	pass@1	pass@10
Agentless	GPT-4o	GPT-4o	22.67%±0.67%	33.67%
LocAgent	GPT-4o	GPT-4o	26.00%±1.67%	37.33%
SWELL	QC-2.5-3B	GPT-4o	20.33%±0.33%	28.33%
		DS-V3	20.66%±0.66%	28.00%
	QC-2.5-7B	GPT-4o	25.67%±1.00%	35.67%
		DS-V3	26.67%±1.33%	37.67%
	DS-V3	GPT-4o	29.67%±1.33%	41.33%
		DS-V3	30.67%±1.00%	**45.33%**
	GPT-4o	GPT-4o	32.00%±0.67%	44.33%
		DS-V3	**32.33%±1.33%**	43.33%

(b) SWELL's Pass Rate

Fig. 5. Issue repair workflow and pass rate for selected tools on SWE-bench Lite. Retrieval and repair were performed using various LMs. The "pass@1" values show the mean and standard deviation. "QC-2.5-xB" and "DS-V3" are short for Qwen-2.5-Coder-xB and DeepSeek-V3, respectively.

Results. The results are presented in Figures 3a and 3b. We also compared with state-of-the-art methods which used GPT-4o or DeepSeek-V3 in the leaderboard[5]. We did not present FAISS specifically as RAG applied the same process.

» *Resolved Issues (45.33%, 136/300).* Notably, SWELL successfully resolved a top-one number of issues compared to all state-of-the-art GPT-4o based agents on the leaderboard. SWELL uniquely fixed 22/136 issues, the highest number across all methods. As shown in Figure 4, SWELL was able to fix issues in 11 out of 12 repositories, averaging 41% of issues resolved per repository. We fixed the most issues in all repositories except `pylint-dev/pylint`. Apart from `pallets/flask` (0%, 0/3), we addressed the highest percentage of issues in `psf/requests` (83.3%, 5/6), while `pylint-dev/pylint` (16.7%, 1/6) had the lowest; SWELL resolved each issue in 2.5 repair attempts. Notably, for 74.3% (101/136) of the issues, SWELL attempted repair only once. SWELL successfully fixed 86.0% (117/136) of the issues in five or fewer attempts. The line chart in Figure 4 displays the average attempts required to resolve issues for each repository.

» *Repair Failures (54.67%, 164/300).* For the remaining issues, SWELL failed to fix within 10 attempts. For 64.6% (106/164) of issues, even though RepoET successfully retrieved all snippets – the retrieved snippets are a superset of the GT snippets – RepoET still failed to repair them. For 12.2% (20/164) of the issues, RepoET successfully retrieved the GT files but did not identify all GT snippets. RepoET failed to retrieve 23.2% (38/164) issues' GT files.

RepoET's Improvements. To investigate how RepoET's retrieval benefits downstream tasks, we also connect our baselines Agentless and LocAgent to the above repair process (see Figure 5a). In this experiment, we represent the average "pass@1" in the format of "mean±std" to show the variance of the three independent runs. As shown in Figure 5b, integrating RepoET into the repair pipeline significantly improves downstream patch success rates. When using GPT-4o as the repair model, SWELL achieves a pass@1 of 32.00%±0.67%, outperforming Agentless (22.67%±0.67%) and LocAgent-Repair (26.00%±1.67%). When paired with DeepSeek-V3, the pass@1 is 32.33%±1.33%,

[5]https://www.swebench.com/index.html

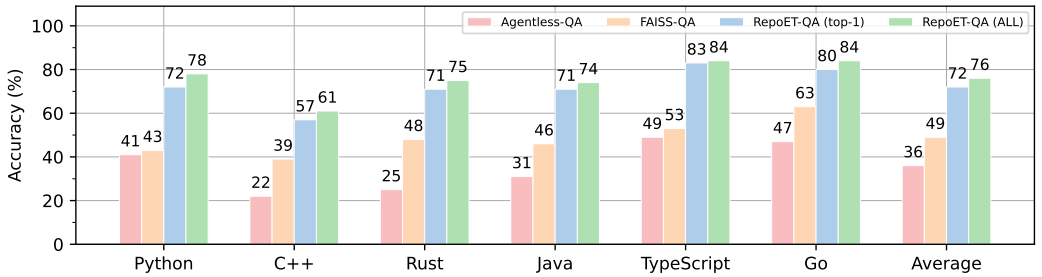


Fig. 6. Performance of RepoET-QA and other selected tools on RepoQA. Results are based on the top-ranked function. RepoET-QA (top-1) uses only the top-ranked, while RepoET-QA (ALL) uses all retrieved functions.

showing consistently strong performance. In terms of pass@10, SWELL combined with DeepSeek-V3 achieves the highest success rate of 45.33%, indicating that more accurate localization substantially benefits patch generation. Notably, even with SLMs (e.g., Qwen-2.5-coder-3B), SWELL still remains competitive. Even when using a smaller retriever (Qwen-2.5-coder-7B), SWELL achieves repair performance comparable to LocAgent with GPT-4o-based retrieval. Specifically, with DeepSeek-V3 as the repair model, SWELL reaches pass@1/pass@10 of 26.67%±1.33%/37.67%, surpassing LocAgent with GPT-4o (26.00%±1.67%/37.33%); with GPT-4o as the repair model, SWELL's results remain near this level (25.67%±1.00%/35.67%). We believe this highlights the effectiveness of the context retrieved by RepoET and its robustness in different LMs.

5.5 RQ5: Case Study - Repository Q&A

Generator. We leverage GPT-4o to identify the function associated with the RepoET-retrieved top-ranked snippet (directly from RQ1) and best matching the query (i.e., the description of the needle function), or to return nothing if no such matching functions exist. We invoked GPT-4o only once in this process to output the function name.

Since we were not aware of LM-based tools specially designed for this task, we connected our generator to RepoET, FAISS, and Agentless for comparison, leading to RepoET-QA, FAISS-QA, and Agentless-QA. We output the accuracy of the retrieved functions.[6]

Results. The first three bars in Figure 6 display the results. Overall, 72% of the top-1 functions retrieved by RepoET are correct. This result significantly surpasses the others by at least 23%. As per programming language, RepoET-QA consistently outperformed the others. An interesting finding is that all three retrievers performed better on TypeScript and Go, but they struggled with C++.

For the same reasons mentioned earlier – the query often includes the needle function's documentation, leading to considerably higher cosine similarity scores – Agentless-QA performed worse than FAISS-QA in this dataset. Although Agentless typically retrieves no more than two functions per query – demonstrating a conservative approach in terms of precision – its workflow design limits its generality across diverse task settings.

We also investigated how RepoET-QA performed when employing GPT-4o to identify functions from all snippets retrieved by RepoET. The result is presented as "RepoET-QA (ALL)" in Figure 6. It shows that RepoET-QA incorrectly orders the function for only 4% of the instances. The largest gap occurs with Python, followed by Rust, while the smallest is with TypeScript.

[6]Leaderboard: https://evalplus.github.io/repoqa.html. Note that all listed tools are LLMs that are provided with context containing GT snippets, making them unsuitable for evaluation. We did not use RepoQA's BLEU-based metric as we employed GPT-4o to output the function name rather than code.

Table 6. Average retrieval cost (in USD) and latency (in seconds) of RepoET. "DS-V3" and "QC-2.5-xB" denote DeepSeek-V3 and Qwen-2.5-Coder-xB, respectively.

Cost	**GPT-4o**	**DS-V3**	**QC-2.5-7B**	**QC-2.5-3B**
FAISS		0.089		
Agentless	0.039	0.021	0.003	0.001
LocAgent	0.560	0.031	0.004	0.001
RepoET	0.710	0.042	0.008	0.002

Latency	**GPT-4o**	**DS-V3**	**QC-2.5-7B**	**QC-2.5-3B**
FAISS		16		
Agentless	63	80	46	36
LocAgent	82	161	101	189
RepoET	121	169	93	57

6 Discussion

RepoET decomposes repo-level retrieval into a structured and controllable agentic workflow based on the "search then filter" paradigm: IR tools provide coarse-grained retrieval, while LMs focus on bounded, local decisions. This modular design mitigates reliance on open-ended, long-context planning and ensures that intermediate decisions remain auditable and transparent. We believe our experiments also demonstrated this architecture's advantage for LMs with various capability. On one hand, RepoET retrieved more expected GT snippets in the expected order compared to state-of-the-art retrievers, whether using proprietary LMs (Section 5.1) or open-source LMs with varying scales (Section 5.2). Specifically, for proprietary large models (RQ1), RepoET demonstrates improvements in precision and ranking accuracy, while for open-source small models (RQ2), RepoET's strength lies in enhancing recall and ranking accuracy. On the other hand, RepoET-retrieved snippets positively impacted two repo-level tasks: SWELL outperformed existing GPT-4o or DeepSeek-V3 agents in SWE-bench Lite (Section 5.4) and RepoET-QA demonstrated significantly higher accuracy in searching for needle functions in RepoQA (Section 5.5).

Performance Ceiling. In practice, RepoET's performance is jointly bounded by: (1) *candidate coverage*—the extent to which the GT resides within the neighborhood of retrieved candidates, and (2) *ranking and selection*—the quality of prioritization and filtering applied to the candidates. Stronger LMs have the potential to improve both factors. For instance, they can expand candidate coverage through improved query rewriting and enhance ranking/selection quality via more accurate relevance judgments. On the other hand, RepoET's architecture is complementary to advances in LM capabilities: node-wise components (i.e., tools and LMs) can be upgraded independently for improved performance.

Retrieval Overhead. Table 6 compares the cost (in USD) and latency (in seconds) of the selected retrievers when applied to the SWE-bench Lite dataset. These data were derived from the results of RQ1 and RQ2, where RepoET (Preview Judge and Snippet Judge with 10-thread parallelization), Agentless (single-thread execution), and LocAgent (single-thread execution) were evaluated using their respective default parallelization settings for each instance, while FAISS was executed once since it is deterministic and does not rely on LMs. The reported values represent per-instance averages and correspond exclusively to the retrieval process. Cost calculations were based on the official pricing information from OpenAI (for GPT-4o), DeepSeek (for DeepSeek-V3), Alibaba (for Qwen-2.5-Coder-3B and Qwen-2.5-Coder-7B), and Jina (for Jina embeddings). Combined with Tables 1 and 2, RepoET improves the retrieval effectiveness (especially recall and Acc@k) across different LMs, with promising quality-overhead trade-off.

Early-termination Exploration. One might be interested what happens if we terminate RepoET early based on heuristics or confidence-based mechanisms? Will this improve RepoET's performance while reducing its overhead? To evaluate its potential benefits, we performed a lightweight study. We considered four practical early-termination heuristics based on our experiences, with

Table 7. Performance of RepoET using early-termination heuristics E1–E4. "HIT" indicates the number of instances where early termination was triggered.

Granule	Variant	Recall	Precision	F1	F2	Acc@1	Acc@3	Acc@5	Acc@10	Cost	Latency	HIT
File	RepoET$_{E1}$	0.673	0.058	0.106	0.215	0.253	0.460	0.520	0.603	–	–	300
	RepoET$_{E2}$	0.450	0.063	0.111	0.203	0.253	0.343	0.360	0.397	–	–	300
	RepoET$_{E3}$	0.823	0.513	0.593	0.691	0.724	0.807	0.817	0.823	0.009	75	33
	RepoET$_{E4}$	0.820	0.559	0.629	0.712	0.727	0.810	0.814	0.820	0.008	142	135
	RepoET	0.823	0.493	0.579	0.683	0.717	0.807	0.817	0.823	0.008	77	–
Snippet	RepoET$_{E3}$	0.730	0.046	0.078	0.148	0.555	0.701	0.725	0.735	0.032	82	–
	RepoET$_{E4}$	0.728	0.063	0.106	0.194	0.539	0.696	0.721	0.728	0.031	84	–
	RepoET	0.733	0.041	0.071	0.138	0.551	0.698	0.723	0.733	0.034	92	–

fixed thresholds chosen a priori (i.e., without task-specific tuning): (E1/KWS-ScoreCut): in the Keyword Engine Search (KWS) node, terminate RepoET by retaining only files with scores greater than 60 points before ranking the searched results. (E2/KWS-DropCut): in the KWS node, terminate RepoET by truncating the results post ranking at the first file whose following file's score drops more than 10 points. (E3/KWS-LMCheck): perform an LM-based sufficiency check immediately after KWS; if the retrieved files are judged not sufficient, proceed with the original workflow (Figure 1); otherwise, directly branch to the snippet stage (i.e., "Search Snippets"). (E4/PVJ-LMCheck): during the Preview Judge node, repeatedly assess sufficiency and terminate once the currently selected files are judged sufficient; otherwise, continue with the original workflow. We name RepoET variants with the four heuristics RepoET$_{E1}$, RepoET$_{E2}$, RepoET$_{E3}$, and RepoET$_{E4}$, respectively. We ran each variant with the same experimental setup as RQ2 using SWE-bench Lite and DeepSeek V3. Greedy decoding was enabled. The results are displayed in Table 7, where "HIT" denotes the the number of instances where the early termination heuristic was successfully triggered. For variants E3 and E4, non-HIT instance automatically executed the original workflow; they are conservative wrappers around RepoET's default pipeline. For SWE-bench Lite, all 300 instances successfully triggered both E1 (KWS-ScoreCut) and E2 (KWS-DropCut). However, this resulted in substantial file-granule performance degradation, due to which, we did not continue the snippet-granule evaluation for them. In contrast, E3 (KWS-LMCheck) and E4 (PVJ-LMCheck) demonstrated better trade-offs. At the file granule, they improved both precision and F2 scores while maintaining comparable recall. At the snippet granule, they also yielded improvements in precision and F2, albeit with a slight drop in recall. Despite these advantages, E4 incurred approximately 2× latency, while E3 showed marginal improvements in latency. Given these trade-offs, we decided to retain our original pipeline (i.e., Figure 1), but we leave adaptive, early termination for future exploration.

Complexity. Although the overall principle of "*search-files* → *filter-files* → *search-snippets* → *filter-snippets*" is intuitive and straightforward, the implementation complexity of RepoET, which involves eight workflow nodes, may raise concerns. However, this design enables a controllable and auditable process, where intermediate results are accessible and individual nodes can be replaced as improved implementations become available. Moreover, our experiments show that this detailed decomposition achieves promising performance, even with models as small as 3B.

Task Adaptation. RepoET's task-general retrieval introduces a trade-off: broader applicability may come with some loss compared with fully task-specialized retrievers. We adopt the task-general design to (1) provide a unified and reusable retrieval backbone across repository-level

tasks, (2) keep workflow behavior and intermediate artifacts auditable and consistent across scenarios, and (3) enable fair cross-benchmark comparison without task-specific tuning. Our current results suggest that RepoET is better aligned with tasks requiring precise evidence localization or cross-file dependency discovery, for example, the issue repair task. For document-driven tasks, a different operating point between recall and filtering strictness may be preferable. We also acknowledge that RepoET is not ideal for strongly latency-sensitive scenarios such as real-time code completion. We believe that task adaptation can be achieved without changing the overall workflow. In practice, we suggest adjusting configurations, signals, and/or outputs of specific nodes, such as using tests (or logs and stack traces) for bugfix-oriented retrieval, prioritizing docs (or README and API references) for QA-oriented retrieval, and tuning scoring criteria for different downstream objectives.

Limitations. Several limitations remain in RepoET. First, the workflow currently executes the entire retrieval process without early termination, which can result in unnecessary computation for simpler issues. Although our previous study indicates that the benefit is not currently significant, it is possible that effective adaptive heuristics could be developed in the future. Second, the downstream generators we used are intentionally simplified to isolate the effects of localization. Integrating more advanced repair strategies that interact with the retrieval process could potentially improve results. Third, the staged workflow introduces non-trivial orchestration overhead. In the absence of requirements for controllability and auditability, a single-shot pipeline might achieve lower latency, particularly for very small repositories or straightforward queries.

Threats to Validity. It might be concerning that the evaluation datasets were already included in the pre-training of LMs. However, all LM-based agents, including closed-source commercial ones on the SWE-bench Lite Leaderboard, failed to achieve scores above 61%. Similarly, no LMs reached 100% on the RepoQA Leaderboard, even when needle functions were explicitly included in the prompt. We also made efforts to mitigate this issue by preventing the inclusion of any potentially guiding information, such as `hints_text`. In addition, our evaluation covers only a limited set of LMs, which may not generalize to models with different capacities. To mitigate this, we selected LMs spanning both proprietary and open-source models, ranging in size from over 170B (MoE architecture) to 3B parameters.

7 Related Work

IR-based Techniques. These works employ sophisticated information retrieval (IR) techniques, based on the framework of "rewrite → retrieve → re-rank" [16], with or without the use of LMs. LangChain [23] and LlamaIndex [28] are two widely adopted libraries that effectively enable the integration of this framework today. HyDE [15] rewrites queries by generating hypothetical responses, and Query2Doc [46] expands the user query to a document with greater detail. In addition to keyword search, engines such as FAISS [40] and Chroma [11] support semantic retrieval by utilizing distributed vectors through embedding models such as BGE [7], MPNet [42], and Jina [2], among others. There are also pre-trained re-rankers [7, 26] that evaluate the relevance of a query to the retrieved information. Surveys have also explored the use of LMs to aid in this process [33, 53, 55]. Our workflow follows a similar framework but with much simpler techniques and more reasonable organization of LLMs and tools.

Finetuning-based Techniques. Recent work on leveraging large language models (LLMs) for code-repository tasks has followed multiple trajectories. On the fine-tune side, approaches such as SWE-Fixer [50] frame issue resolution as a retrieve-then-edit pipeline: a file retrieval module identifies relevant files, and an editing module generates patches. They collected a dataset of 110k instances in two separate stages and conducted training accordingly, achieving competitive results

on SWE-Bench Lite and verified. Meanwhile, SweRank [39] casts the issue localization sub-task as a code ranking problem by constructing the SweLoc dataset and training a bi-encoder retriever plus an LLM reranker, thereby outperforming both traditional retrieval models and expensive agent-based systems on localization benchmarks. CGM [44] explores how open-source LLMs can directly handle repository-level tasks without an agent by integrating a code-graph representation of file/-function dependencies into the LLM's attention mechanism. Using Graph-RAG, it achieves 43% resolution on SWE-Bench Lite with a 72B open model. SWE-RL [48] further advances this direction by employing reinforcement learning with reward signals derived from software evolution histories, enabling LLMs to perform iterative reasoning and verification over candidate patches. In parallel, SWE-Gym [38] establishes a standardized reinforcement learning environment for training software-engineering agents and verifiers, bridging the gap between simulated reasoning and real-world code repositories.

Agent-based Techniques. These works heavily depend on the ability of LMs. They develop retrieval agents centered around LMs and enable automated semantic search, entrusting the control and orchestration of the retrieval process (i.e., the start, the execution pipeline, and the stop) to LMs. In particular, these agents provide LMs with an API set and utilize their tool call capabilities [35] and the ReAct [52] paradiagm to iteratively retrieve information from the repository. For example, SWE-agent [51] and Sweep [3] allow LMs to navigate the repository through file trees, preview files, and view details. Other tools such as Bloop [1], AutoCodeRover [54], and SpecRover [41] enable LMs to search for the snippets of specific code entities, such as functions, methods, and classes. MarsCode Agent [27], LingmaAgent [31], and CodePlan [8] model the repository as a knowledge graph, describing relationships between various code entities as well as between entities and documentation. They summarize the repository based on this knowledge graph or perform graph-based searches before their retrieval agents start invoking the aforementioned APIs iteratively. More recently, LocAgent [10] constructs a graph of the codebase so that an LLM agent can perform multi-hop reasoning over code relationships. Unlike them, RepoET follows a controllable workflow to produce auditable results. Anthropic released five frequently used agent patterns [5]. Importantly, our workflow is designed to accommodate LMs with different abilities and sizes.

Agentic Workflow. A notable work in this line is Agentless [49]. It directs LMs to identify all relevant snippets by sequentially retrieving files, code entities, and snippets through an analysis of the repository's file tree, previewing relevant files, and viewing specifics of entities. RepoGraph [37] further improves this process by incorporating a knowledge graph. Although these works share some similarities with ours, i.e., using file trees to retrieve relevant files and viewing file details to assess relevance, these difficult sub-tasks are fundamental to their retrieval component, which otherwise struggle when backed by weaker LLMs or SLMs. In contrast, we assign LMs with sub-tasks more accessible for various LMs, such as inferring file names, analyzing sub-trees, and assessing previews and snippets. Furthermore, our workflow follows a distinct human search logic. We provide a more detailed comparison with these works in Appendix B.

8 Conclusion

We present RepoET, a novel RLRAG retriever specifically designed to support language models of varying sizes and capabilities. Our evaluation shows that RepoET outperforms existing retrievers by accurately retrieving more relevant snippets in a better order across four LMs of different sizes. We also obtained significant results in two repository-level tasks using snippets retrieved by RepoET. We believe that the effectiveness, flexibility, and generality of RepoET make it highly suitable for real-world software engineering applications.

Acknowledgments

This work is supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), the "111 Center" (No. B26023), the National Natural Science Foundation of China (Grant #62522209, #62272218, #92582201), an Ant Group Research Fund, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yuan Yao and Zhaogui Xu are the corresponding authors.

References

[1] Bloop AI. 2024. bloop. https://github.com/BloopAI/bloop Accessed: 2026-03-17.
[2] Jina AI. 2024. Elevate Your Code Search with New Jina Code Embeddings. https://jina.ai/news/elevate-your-code-search-with-new-jina-code-embeddings/ Accessed: 2026-03-17.
[3] Sweep AI. 2024. Sweep. https://github.com/sweepai/sweep Accessed: 2026-03-17.
[4] Anthropic. 2024. Building Effective Agents. https://www.anthropic.com/research/building-effective-agents Accessed: 2026-03-17.
[5] Anthropic. 2026. Multi-Agent Coordination Patterns: Five Approaches and When to Use Them. https://claude.com/blog/multi-agent-coordination-patterns Accessed: 2026-05-14.
[6] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. 2025. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. In *Proceedings of the 2025 International Conference on Learning Representations (ICLR '25)*.
[7] BAAI. 2024. BGE: BAAI General Embedding. https://github.com/FlagOpen/FlagEmbedding Accessed: 2026-03-17.
[8] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.* 1, FSE (2024).
[9] Jianming Chang, Xin Zhou, Lulu Wang, David Lo, and Bixin Li. 2026. Bridging Bug Localization and Issue Fixing: A Hierarchical Localization Framework Leveraging Large Language Models . *IEEE Transactions on Software Engineering* (2026). https://doi.org/10.1109/TSE.2026.3668601
[10] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. In *Annual Meeting of the Association for Computational Linguistics (ACL '25)*.
[11] Chroma Core. 2024. chroma. https://github.com/chroma-core/chroma Accessed: 2026-03-17.
[12] Xin Luna Dong. 2024. The Journey to a Knowledgeable Assistant with Retrieval-Augmented Generation (RAG). In *Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*.
[13] Huizhong Duan, ChengXiang Zhai, Jinxing Cheng, and Rohit Kumar. 2013. Supporting Keyword Search in Product Database: a Probabilistic Approach. *Proc. VLDB Endow.* 6 (2013).
[14] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. High-Dimensional Similarity Search for Scalable Data Science. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE '21)*.
[15] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. Precise Zero-Shot Dense Retrieval without Relevance Labels. In *Proceedings of the 2023 Annual Meeting of the Association for Computational Linguistics (ACL '23)*.
[16] Ed Greengrass. 2001. Information Retrieval: A Survey. (2001).
[17] Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2024. Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents. arXiv:2310.19923 [cs.CL] https://arxiv.org/abs/2310.19923
[18] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE '12)*.
[19] HuggingFace. 2024. transformers. https://github.com/huggingface/transformers Accessed: 2026-03-17.
[20] Pinecone IO. 2024. Pinecone. https://github.com/pinecone-io Accessed: 2026-03-17.
[21] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Issue Localization via LLM-Driven Iterative Code Graph Searching. In *Proceedings of the 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE '25)*.
[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *Proceedings of the 2024 International Conference on Learning Representations (ICLR '24)*.
[23] LangChain. 2024. Build a Retrieval Augmented Generation (RAG) App. https://python.langchain.com/docs/tutorials/rag/ Accessed: 2026-03-17.

[24] Cong Li, Zhaogui Xu, Peng Di, Dongxia Wang, Zheng Li, and Qian Zheng. 2024. Understanding Code Changes Practically with Small-Scale Language Models. In *Proceedings of the 2024 IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*.

[25] Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024. RepoQA: Evaluating Long Context Code Understanding. In *Proceedings of the 2024 Workshop on Long-Context Foundation Models at the 2024 International Conference on Machine Learning (LCFM '24)*.

[26] Qi Liu, Bo Wang, Nan Wang, and Jiaxin Mao. 2024. Leveraging Passage Embeddings for Efficient Listwise Reranking with Large Language Models. arXiv:2406.14848 [cs.CL] https://arxiv.org/abs/2406.14848

[27] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode Agent: AI-native Automated Bug Fixing. arXiv:2409.00899 [cs.SE] https://arxiv.org/abs/2409.00899

[28] LlamaIndex. 2024. Retrieval Augmented Generation (RAG). https://docs.llamaindex.ai/en/stable/getting_started/concepts/#retrieval-augmented-generation-rag Accessed: 2026-03-17.

[29] LMSys. 2024. LMArena Leaderboard. https://lmarena.ai/leaderboard Accessed: 2026-03-17.

[30] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D. Lane, and Mengwei Xu. 2024. Small Language Models: Survey, Measurements, and Insights. arXiv:2409.15790 [cs.CL] https://arxiv.org/abs/2409.15790

[31] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? arXiv:2406.01422 [cs.SE] https://arxiv.org/abs/2406.01422

[32] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. SoRFT: Issue Resolving with Subtask-oriented Reinforced Fine-Tuning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL '25)*.

[33] Marc Najork. 2023. Generative Information Retrieval. In *Proceedings of the 2023 International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23)*.

[34] Chien Van Nguyen, Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, Junda Wu, Ashish Singh, Yu Wang, Jiuxiang Gu, Franck Dernoncourt, Nesreen K. Ahmed, Nedim Lipka, Ruiyi Zhang, Xiang Chen, Tong Yu, Sungchul Kim, Hanieh Deilamsalehy, Namyong Park, Mike Rimer, Zhehao Zhang, Huanrui Yang, Ryan A. Rossi, and Thien Huu Nguyen. 2024. A Survey of Small Language Models. arXiv:2410.20011 [cs.CL] https://arxiv.org/abs/2410.20011

[35] OpenAI. 2024. Function Calling. https://developers.openai.com/api/docs/guides/function-calling Accessed: 2026-03-17.

[36] OpenAI. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/ Accessed: 2026-03-17.

[37] ozyyshr. 2024. RepoGraph. https://github.com/ozyyshr/RepoGraph Accessed: 2026-03-17.

[38] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training Software Engineering Agents and Verifiers with SWE-Gym. In *Proceedings of the 2025 Forty-second International Conference on Machine Learning (ICML '25)*.

[39] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan-Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2026. SWERank: Software Issue Localization with Code Ranking. In *Proceedings of the 2026 Fourteenth International Conference on Learning Representations (ICLR '26)*.

[40] Facebook Research. 2024. FAISS. https://github.com/facebookresearch/faiss Accessed: 2026-03-17.

[41] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*.

[42] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-Training for Language Understanding. In *Proceedings of the 2020 International Conference on Neural Information Processing Systems (NIPS '20)*.

[43] Sweep. 2024. Chunking 2M+ Files a Day for Code Search using Syntax Trees. https://github.com/sweepai/sweep/blob/main/docs/pages/blogs/chunking-2m-files.mdx Accessed: 2026-03-17.

[44] Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. 2025. Code Graph Model (CGM): A Graph-Integrated Large Language Model for Repository-Level Software Engineering Tasks. In *Proceedings of the 2026 Thirty-ninth Annual Conference on Neural Information Processing Systems (NeurIPS '25)*.

[45] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2022. A Large-Scale Empirical Study of Commit Message Generation: Models, Datasets and Evaluation. *Empirical Softw. Engg.* 27, 7 (2022). https://doi.org/10.1007/s10664-022-10219-1

[46] Liang Wang, Nan Yang, and Furu Wei. 2023. Query2doc: Query Expansion with Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*.

[47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS '22)*.

[48] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2026. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. In *Proceedings of the 2026 Thirty-ninth Annual Conference on Neural Information Processing Systems (NeurIPS '25)*.

[49] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* FSE (2025).

[50] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. [n. d.]. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. In *Findings of the Association for Computational Linguistics: ACL 2025 (ACL '25)*.

[51] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Proceedings of the 2024 Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS '24)*.

[52] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the 2023 International Conference on Learning Representations (ICLR '23)*.

[53] ChengXiang Zhai. 2024. Large Language Models and Future of Information Retrieval: Opportunities and Challenges. In *Proceedings of the 2024 International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24)*.

[54] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 2024 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*.

[55] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Haonan Chen, Zheng Liu, Zhicheng Dou, and Ji-Rong Wen. 2024. Large Language Models for Information Retrieval: A Survey. arXiv:2308.07107 [cs.CL] https://arxiv.org/abs/2308.07107

A Preliminary Study

We conducted a preliminary study to investigate the retrieval performance of classical keyword engines and vector stores. Our study concerns the following questions:

Q1 How effective are keyword engines and vector stores for snippet retrieval in RLRAG? How effective is file retrieval?

Q2 How far are engine-retrieved files from the ground-truth files?

Task Setup. We set up the study using the MCMDP dataset [24], a subset of the state-of-the-art MCMD dataset [45] for commit message generation. Each item in the dataset consists of a code change represented in the unified diff format along with its corresponding commit message and metadata. We chose this dataset because it targets a repo-level task. Furthermore, the commit messages in MCMDP have been automatically refined to remove human biases and to remain concise, i.e., conveying the key information; they have also been automatically verified to ensure that the short commit messages accurately correspond to their code changes. Although the dataset is not large, it includes a diverse range of edited file types, such as `.py`, `.java`, `.c`, `.xml`, and `.md`, making it suitable for conducting a general-purpose preliminary experiment.

In our study, we created 45 retrieval tasks across 23 repositories by randomly selecting 45 pairs of commit messages and their corresponding code changes from the dataset. For each task, we treated all edited files in a code diff as ground-truth files to be retrieved, i.e., GT files. To obtain GT snippets, we adopted our approach to processing issues of SWE-bench Lite (Section 5), parsing all unchanged and removed lines associated with a code diff. We also downloaded from GitHub the repository revision before applying each git commit and conducted snippet retrieval in the revision using the commit message as the query.

Engine Setup. We established two IR-based engines for each repository revision: a keyword engine and a vector store. The keyword engine was created following the methods outlined in Section 2 and Section 3. The vector store FAISS was configured as described in Section 5, where Jina Embedding and cosine similarity were integrated. Given a query, we retrieved the top five snippets and/or files.

Studied Metrics. We assessed them with MPRF, which includes precision, recall, F1, and F2, to compare the retrieved snippets with GT snippets. These metrics were calculated as outlined in Section 5.

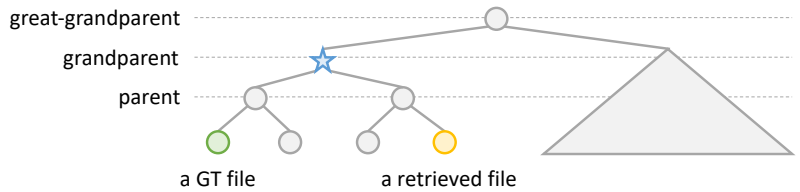
Q1's Results. The table below displays the retrieval results for both snippets and files. The results indicate that both engines have difficulty retrieving GT snippets for the 45 RLRAG retrieval tasks that we have set up. On one hand, both of them recalled less than 37% of GT snippets. On the other hand, for the snippets that are recalled, GT snippets occupy less than 6%.

Task	Engine	Recall	Precision	F1	F2
Snip	Key. Eng.	0.297	**0.051**	**0.080**	**0.130**
	Vec. Sto.	**0.361**	0.030	0.053	0.100
File	Key. Eng.	**0.711**	0.142	**0.237**	**0.395**
	Vec. Sto.	0.467	**0.145**	0.210	0.305

However, in terms of the files where the retrieved snippets are located, both engines recalled more than 45% of the GT files, whereas the keyword engine demonstrated more satisfactory retrieval performance. More specifically, the keyword engine recalled 70% of GT files, resulting in a precision increase of approximately 9%. Although the vector store maintained a similar precision,

it is less effective than the keyword engine, resulting in a recall of 46.7%. Upon manual inspection, we discovered that the keyword engine succeeded in capturing most of the code entities mentioned in the query and accurately mapping them to corresponding files within the repository. This can be attributed to the repository being primarily developed using programming languages in which the defined code entities are uniquely identified by their names. Such a characteristic is more effectively supported by keyword engines compared to vector stores, which are better at handling natural language words with synonyms and polysemy. On the other hand, while the keyword engine effectively located some related files, it struggled to identify the specific GT snippets within the files. This could be due to the GT snippets not directly containing the code entities in query, but rather involving their references or calling to their methods, among other factors. We find these results reasonable, as the snippets retrieved from vector stores are functionally similar to the query, more likely falling into the question space, rather than the solution space. However, most snippets obtained through the keyword engine are extensively using the involved entities for certain purposes, which may be more pivotal for effectively addressing the query.

Q2's Results. We traced upward along the repository tree from the parent directory where the retrieved files are located to assess the distance between the retrieved files and the GT files. Our investigation for the keyword engine revealed that for over 77% of the GT files, their residing directory and the directory of the retrieved files share the same grandparent directory, with 80% sharing the same great-grandparent directory. In contrast, the vector store showed percentages of 57.8% and 71.1% for sharing a grandparent and great-grandparent directory, respectively. We deem these results reasonable, because they align with a best practice in software engineering: Functionally similar code should be grouped within the same module or in closely related modules, with the proximity of modules often indicated by their locations in the repository tree. These findings further underscore the importance of the keyword engine: It creates more opportunities for LMs to identify GT files in close proximity to the retrieved files. Below is an illustrative demonstration.



Observation. This preliminary study suggests the potential of employing the retrieved files (rather than snippets) from a keyword engine as a fundamental and foundational intermediate result. This result can help reduce the repository to the neighborhood of retrieved files, which forms a small, tractable space that might be well-refined through a series of LM calls. For vector stores, it might be useful to further improve the precision after we recall a portion of correct files.

B More Discussions

Keyword Engine. Building the keyword engine can be fast as it requires only two passes over the repository, thanks to its simple design. Additionally, the engine is created once for archived repositories. Evolving repositories can be built incrementally by their version differences, achieved by removing snippets of deleted files, chunking modified files and replacing their snippets with new ones, or adding snippets for newly added files; these operations require simultaneously updating the inverted index accordingly.

Agentless [49]. Agentless is the most similar tool to RepoET; however, there are significant differences.

Firstly, they are based on different human search logic. Agentless follows "files → classes/methods → snippets", whereas RepoET adheres to the flow of "search-files → filter-files → search-snippets → filter-snippets". While most nodes in Agentless rely heavily on LLMs, RepoET effectively coordinates LMs and tools, assigning more complex sub-tasks to the tools while utilizing LMs for small-scope refinements. This design enables RepoET to work effectively with various LMs, whereas Agentless struggles in this regard.

Additionally, Agentless is designed as an issue repair workflow with task-specific nodes, such as generating reproduction tests and crafting issue-specific prompts. In contrast, RepoET is applicable to a broader range of repo-level tasks.

Finally, the tools utilized by them are distinct. Agentless does not use any IR-based tools initially. In its recent 1.5 version, it employs a vector store to identify functionally similar files, enhancing its tree exploration results. Conversely, RepoET utilizes a keyword engine to search for files based on a query summary, this approximates finding files that extensively use the involved code entities. While Agentless uses file previews to identify relevant code elements, RepoET leverages this tool to filter out irrelevant files. Furthermore, Agentless finds relevant snippets (or edit locations) based on the relevant code elements, whereas RepoET directly judges candidate snippets from either the keyword engine or the vector store.