

Repository-Level Code Understanding by LLMs via Hierarchical Summarization: Improving Code Search and Bug Localization

Amirkia Rafiei Oskooei[1,2(✉)], Selcan Yukcu[1], Mehmet Cevheri Bozoglan[1], and Mehmet S. Aktas[2]

[1] Intellica Business Intelligence Consultancy, R&D Center, Istanbul, Turkey
{amirkia.oskooei,selcan.yukcu,cevheri.bozoglan}@intellica.net
[2] Department of Computer Engineering, Yildiz Technical University, Istanbul, Turkey
amirkia.oskooei@std.yildiz.edu.tr, aktas@yildiz.edu.tr
https://www.intellica.net, https://www.yildiz.edu.tr

Abstract. Bug localization and semantic code search within large software repositories is a significant and time-consuming challenge for developers, particularly when dealing with bug reports from end-users who lack technical expertise. Traditional similarity-based code search methods struggle with the inherent domain and vocabulary mismatch between end-user reports and codebase semantics, while directly applying Large Language Models (LLMs) is hampered by their limited context windows and lack of repository-level understanding. To address these limitations, this paper introduces a novel, structure-aware methodology for creating repository-aware LLMs using hierarchical summarization. Our approach comprises a pre-processing phase that constructs an abstract repository tree, creates a context-aware LLM primed with project knowledge, and generates hierarchical summaries at project, directory, and file levels. The inference phase employs a top-down search strategy, guiding the LLM to progressively narrow down the search space from directory-level to file-level, effectively localizing bug-relevant code. This method mitigates the context window bottleneck and leverages LLMs' semantic understanding to overcome domain gap issues. Evaluated on a real-world dataset of Jira issues from a large-scale industrial project, our approach significantly outperforms both Flat Retrieval baselines and state-of-the-art LLM + Retrieval-Augmented Generation (RAG) systems, achieving a Pass@10 of 0.89 and Recall@10 of 0.33. The results demonstrate the efficacy of hierarchical summarization in enabling scalable, task-agnostic, and structure-aware repository-level code comprehension for improved bug localization and code search, particularly in scenarios involving non-technical end-user bug reports.

Keywords: Software Engineering · Large Language Models (LLMs) · Semantic Code Search · Automatic Program Repair · Defect Detection · Applied Machine Learning

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025 Workshops, LNCS 15886, pp. 88–105, 2026.
https://doi.org/10.1007/978-3-031-97576-9_6

1 Introduction

Locating and fixing bugs within codebases is a notoriously time-consuming and demanding task for software developers. This process typically necessitates a thorough review of the codebase, requiring developers to meticulously analyze potentially vast amounts of code to pinpoint the files or code segments responsible for a reported bug or defect. The challenge is amplified significantly in large code repositories, where the sheer volume of code can make bug localization feel like searching for a needle in a haystack.

Automated Program Repair (APR), encompassing Code Defect Detection and Repair, has long been a vital and actively researched area within software engineering. The promise of automated tools to identify and resolve bugs is highly attractive to developers seeking to improve efficiency and code quality. The rapid advancement of neural networks, driven by increased computational power and the explosion of large-scale datasets, has significantly propelled progress in this domain, offering new avenues for automating bug localization from issue reports.

For instance, early approaches to automated code search and bug localization have successfully employed language models like UnixCoder, CodeBERT, and ModernBERT. These encoder-only models operate by transforming both code and natural language into vector representations (embeddings). This embedding space enables the system to identify code files or segments that are semantically similar to a natural language query, such as a bug report from platforms like GitHub or Jira, by calculating a similarity score between the embeddings of query and code.

More recently, Large Language Models (LLMs), celebrated for their remarkable ability to understand and generate text, code, and visuals [14,19,24,26] thanks to their huge pre-training big data [23,28], have also been adopted for bug localization. While LLMs demand greater computational resources for both training and inference and are not as cost-effective as encoder-only models, their enhanced capabilities in understanding both codebase semantics and query content, coupled with their general intelligence, make them compelling alternatives.

Despite the increasing adoption of both encoder-only and LLM-based approaches for code search and bug localization, significant limitations persist. Similarity-based methods, which rely on embedding models to bridge natural language queries (bug reports) and code segments, often suffer from a "domain gap and vocabulary mismatch" problem, as shown in Fig. 1. This issue arises when the natural language query is poorly formulated or originates from a domain significantly different from the codebase itself. Consider, for example, an end-user reporting a graphical user interface issue, lacking any understanding of the underlying codebase or programming concepts. Their issue description, focused solely on UI functionality, may contain vocabulary and concepts that are semantically distant from the technical vocabulary of the codebase. Consequently, directly calculating similarity between embeddings of such end-user issue descriptions and code segments becomes ineffective. Developers are then forced to manually

refine queries to bridge this gap, effectively requiring them to pre-analyze the codebase and understand the bug before even initiating an automated search.

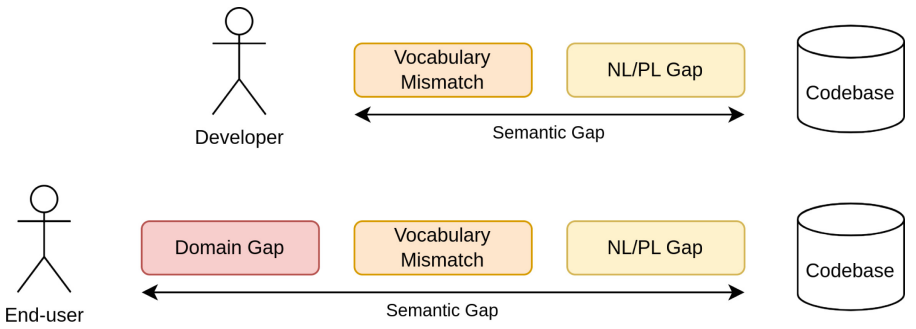

Fig. 1. Our approach in addition to vocabulary mismatch and NL/PL gap problems which are widely recognized and addressed in semantic code search, solves the issue of domain gap (the red box). (Color figure online)

LLMs, with their advanced semantic understanding, offer a potential avenue to mitigate the domain gap and vocabulary mismatch problem to some extent. However, even with their superior comprehension of natural language and code relationships, LLMs face their own set of challenges in bug localization, primarily stemming from **context window limitations**. This constraint becomes acutely apparent when dealing with large, enterprise-scale repositories containing hundreds or thousands of files. Prompting an LLM with the entirety of such a codebase is simply infeasible due to these context limits.

Furthermore, even when utilizing long-context LLMs capable of processing moderately sized repositories, simply feeding code files sequentially is insufficient for effective repository-level understanding. LLMs require a structured context that captures the intricate interconnectivity and functional relationships between files within the repository. Therefore, beyond merely addressing context window size, **pre-processing and structuring contextual information** becomes crucial to enable LLMs to truly comprehend code at the repository level.

To overcome these challenges, this paper introduces a **structure-aware, highly scalable, and task-agnostic approach** designed to empower LLMs with repository-level code comprehension, effectively creating **context-aware LLMs**. Our methodology centers around **hierarchical summarization**, representing expansive repositories as reduced-size, tree-structured prompts. This strategy effectively mitigates the context window limitations of LLMs while simultaneously leveraging their inherent strengths in semantic code search to address the domain gap and vocabulary mismatch problem prevalent in traditional similarity-based methods. This approach enables more effective code search and bug localization across diverse query types and large codebases.

In this study, our main objectives are:

1. To address the domain gap and vocabulary mismatch problem inherent in similarity-based code search approaches for bug localization, particularly when dealing with end-user bug reports.
2. To overcome the limited context window challenge faced by Large Language Models (LLMs) when applied to large-scale code repositories
3. To develop a scalable and task-agnostic approach that enables LLMs to effectively understand and reason about code at the repository level
4. To introduce a novel hierarchical summarization technique for representing large code repositories in a reduced-size format suitable for LLM prompting, while preserving essential structural and functional information.

This paper introduces a novel, hierarchical summarization methodology to enable repository-aware Large Language Models (LLMs). Our approach constructs a hierarchical tree of code summaries, significantly reducing the input size for LLMs while preserving crucial repository-level context. This addresses the LLM context window limitation and enables effective code understanding for downstream tasks. Specifically, we develop a context-aware LLM architecture for code search and bug localization, tackling the domain gap and vocabulary mismatch problem inherent in traditional similarity-based methods. Our hierarchical summarization allows the LLM to bridge the gap between natural language bug reports (even from end-users) and the codebase, effectively localizing bugs without requiring developers to manually craft specialized queries. Furthermore, this architecture forms a scalable and task-agnostic framework applicable to other software engineering tasks requiring repository-level code comprehension. A rigorous empirical evaluation on a real-world industry-level bug localization dataset demonstrates that our approach is highly effective. The results reveal that our method significantly surpasses the performance of both similarity-based and existing LLM-based methods, achieving quantifiable improvements in localization accuracy. This empirical evidence strongly validates the practical applicability of our technique for enhancing bug localization in complex, real-world software environments.

2 Related Works

To contextualize our proposed methodology, this section provides an overview of existing research in code search and bug localization, focusing on two dominant paradigms: similarity-based approaches and more recent Large Language Model (LLM)-driven techniques. We examine the core principles, strengths, and limitations of each category, highlighting how these prior works have attempted to address the challenges of bridging the gap between natural language queries and codebases. This review serves as a foundation for understanding the novelty and contributions of our hierarchical summarization approach, which aims to integrate the benefits of both paradigms while mitigating their respective drawbacks to achieve more effective repository-level code understanding for bug localization.

Similarity-based code search methods have long been a cornerstone in the field, leveraging the power of embedding spaces to bridge the gap between natural language queries and programming language code. These techniques, exemplified by models like CodeBERT [11], UniXcoder [12], and most recently ModernBert [31], encode both natural language queries and code snippets into a shared embedding space. Similarity metrics such as cosine similarity or dot product are then used to measure the semantic relatedness between queries and code. This approach effectively addresses the inherent Natural Language/Programming Language (NL/PL) gap by comparing queries and code in a semantically rich embedding space [8].

However, similarity-based methods are not without limitations. They often struggle with poorly formulated queries, such as vague or incomplete bug descriptions, and suffer from the vocabulary mismatch problem. For instance, end-user bug reports often use non-technical language that is semantically distant from the code's vocabulary, leading to poor retrieval performance. To mitigate these challenges, researchers have explored query expansion and augmentation techniques. Query expansion aims to broaden the scope of the original query by adding semantically similar terms, while query augmentation seeks to generate diverse query formulations to better capture the user's intent. Some studies have even leveraged LLMs to enhance similarity-based code search through query augmentation and expansion, as well as for flat retrieval using similarity calculation [16,17,30]. These hybrid approaches represent a significant step towards improving the robustness of similarity-based code search, but they still inherit some limitations, particularly in handling the domain gap and vocabulary mismatch problems when dealing with non-technical end-user bug reports.

In recent years, Large Language Models (LLMs) have emerged as powerful tools for various Software Engineering (SE) tasks, including repository-level code auditing [13], bug localization [25], code completion [32], and code question answering [6]. Unlike traditional similarity-based methods, LLM-driven approaches leverage the models' deep semantic understanding to directly process and reason about code and natural language queries. These methods have shown remarkable promise in tackling complex SE tasks at the repository level, offering several advantages over similarity-based techniques.

LLM-based approaches excel in semantic understanding, enabling them to bridge the domain gap and vocabulary mismatch that often plague similarity-based methods. For example, LLMs can effectively interpret non-technical end-user bug reports and relate them to the technical vocabulary of the codebase [18]. They also demonstrate superior context awareness, capable of processing and reasoning over long code sequences and complex inter-file dependencies [9,33]. Agentic workflows and planning have also recently received significant attention. CodePlan [5] introduces a neuro-symbolic framework that synthesizes multi-step plans of code edits, leveraging LLMs and combining incremental dependency analysis with adaptive planning for repository-level coding tasks. AutoCodeRover [34] proposes an autonomous program improvement approach that combines LLMs with sophisticated code search techniques, utiliz-

ing a stratified search strategy and test-based fault localization for effective GitHub issue resolution. REPOAUDIT [13] presents an autonomous LLM-agent for repository-level code auditing, employing an agent memory, data-flow analysis, and a validator to enable precise and efficient bug detection in real-world software systems.

However, LLM-based approaches also face challenges. Context window limitations remain a significant bottleneck, especially when dealing with large-scale code repositories [18,25]. Moreover, training and deploying LLMs for large codebases can be prohibitively expensive for many research groups and organizations, and inference latency can become a bottleneck for interactive developer workflows. While long-context LLMs offer expanded context windows, they may still struggle with massive repositories exceeding these limits [7]. Despite these challenges, LLM-based approaches represent a significant advancement in repository-level code understanding, offering a more robust and semantically aware alternative to traditional techniques.

The authors of [3] addressed dynamic information service reliability and scalability in distributed Grid environments, laying foundational work for system-level fault tolerance. The authors of [27] contributed a systematic examination of mobile application verification techniques, emphasizing formal and empirical evaluation methods. In the domain of behavioral modeling, the authors of [21,22] explored the use of embeddings for representing user interaction patterns in web environments. The authors of [29] proposed a provenance collection platform to ensure traceability in large-scale environmental simulations, while the authors of [4,20] developed collaborative and complexity-aware Grid platforms to support scientific computing and high-performance workflows. Although these works contributed significantly to distributed computing, provenance management, and user behavior modeling, none of them focus on repository-level code comprehension using hierarchical summarization or on bridging the semantic gap between natural language bug reports and codebases using LLMs, which is the central novelty of this study.

Our methodology, Repository-Level Code Understanding by LLMs via Hierarchical Summarization, uniquely synergizes the strengths of both similarity-based and LLM-based approaches while addressing their limitations. Like similarity-based methods, we leverage efficient retrieval techniques to navigate large codebases. However, unlike traditional similarity-based approaches that rely solely on embedding similarity, we incorporate LLMs to create a hierarchical summarization of the repository, enabling a more nuanced and context-aware representation of the codebase.

Similar to LLM-based approaches, we harness the semantic understanding capabilities of LLMs to bridge the domain gap and vocabulary mismatch problem. However, unlike existing LLM-based methods that often struggle with context window limitations, our hierarchical summarization significantly reduces the input size for LLMs, allowing them to process and reason over large codebases efficiently. Furthermore, our approach enhances the explainability of bug localization by providing structured, hierarchical summaries that facilitate both

machine and human comprehension of the codebase, in contrast to the often opaque nature of embedding-based or end-to-end LLM approaches.

In essence, our methodology offers a novel, structure-aware approach to repository-level code understanding that combines the scalability of similarity-based methods with the semantic richness of LLMs. By leveraging hierarchical summarization, we overcome the context window bottleneck and domain gap challenges, enabling more effective and scalable code search and bug localization, particularly in complex, real-world software repositories.

3 Methodology

This research introduces a novel methodology for enhancing Large Language Model (LLM) performance in repository-level code understanding, specifically for the task of automated bug localization. Our approach centers on a two-phased system architecture encompassing a comprehensive pre-processing stage and an inference phase optimized for efficient and accurate bug localization within large codebases. The pre-processing phase focuses on transforming a raw code repository into a structured, LLM-interpretable format, while the inference phase leverages this structured representation to effectively guide the LLM in identifying bug locations based on natural language queries or bug reports. This methodology is designed to address key challenges in applying LLMs to large codebases, including context window limitations and the semantic gap between natural language bug reports and the technical vocabulary of source code.

3.1 Pre-processing Phase

The pre-processing stage is crucial for preparing the code repository and enabling effective LLM-based analysis. This phase consists of three key steps designed to create a hierarchical, context-rich representation of the codebase.

Step 1: Building Repository Tree. The initial step involves constructing an abstract **repository tree**, a hierarchical representation of the codebase tailored to project needs and not strictly bound by the physical file system structure. This abstract tree organizes the repository into three distinct levels: root, directory, and file, as shown in Fig. 2. While the depth of this tree can be adjusted to accommodate repositories with varying levels of nesting, for this study, we employ a depth of three, incorporating a single directory level beneath the root. This depth is strategically chosen to balance representational granularity with computational efficiency, particularly relevant for managing the complexity of large repositories. The flexibility to define the tree abstractly allows developers and stakeholders to structure the repository representation according to the logical modules and organizational principles most pertinent to the project's architecture and comprehension.

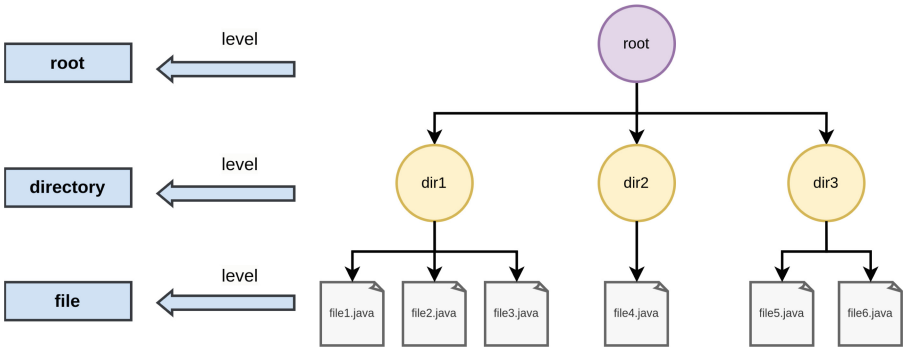

Fig. 2. The repository tree is an abstract tree created during the pre-processing phase that represents the codebase as file and directory level(s).

Step 2: Creating Context-Aware LLMs. To imbue the LLM with project-specific context, a crucial step is the creation of what we term a **"context-aware LLM."** This involves providing the LLM with a rich set of resources during the pre-processing phase to generate a comprehensive project summary. These resources, supplied by project stakeholders, include the raw codebase, project-specific rules and guidelines, README files, and project documentation. The repository tree, defined in the preceding step, also plays a vital role by illustrating the project's structural organization to the LLM. These resources collectively provide the LLM with essential information regarding the repository's general functionality, intended use cases, underlying technology stack, and overarching architecture. Specifically, to generate a detailed root-level project summary, these resources are input to a long-context LLM, capable of processing extensive textual information. The LLM then outputs a concise project summary in natural language. This project summary is subsequently integrated into the system-level prompts of any off-the-shelf LLM, effectively transforming it into a "context-aware LLM." Throughout this paper, the term "context-aware LLM" refers to an LLM that is systemically primed with this project summary and the aforementioned resources, enabling it to operate with an inherent understanding of the repository's structure, functionality, and architectural nuances.

Step 3: Building Knowledge Tree. Following the creation of the context-aware LLM and the definition of the repository tree, a **bottom-up** summarization approach is employed to generate concise, natural language summaries for each code file and directory within the repository. Starting at the leaf nodes of the repository tree (file level), the context-aware LLM is prompted to generate a summary for each code file (Fig. 3). These file-level summaries capture essential aspects of the code, including its functionality, imports, and interrelationships with other code components.

Subsequently, these file-level summaries are aggregated and provided as input to the context-aware LLM to generate directory-level summaries (Fig. 4). Each

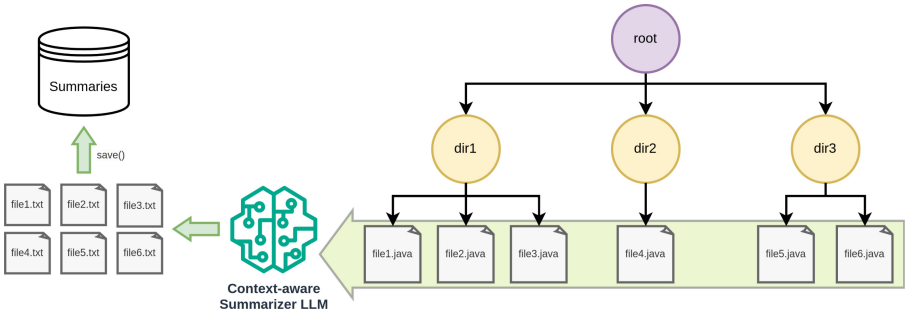

Fig. 3. Illustration of file-level summarization of repository tree using a context-aware LLM

directory-level summary concisely represents the collective functionality and purpose of all files contained within that directory.

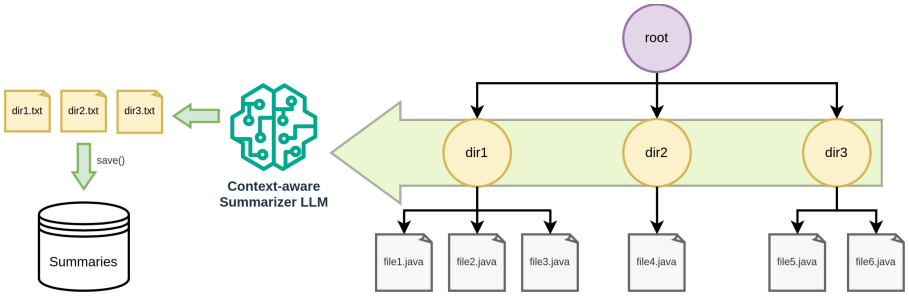

Fig. 4. Illustration of directory-level summarization of repository tree using a context-aware LLM

The culmination of this bottom-up summarization process is the creation of a **knowledge tree**. This knowledge tree mirrors the abstract repository tree structure defined earlier; however, instead of containing raw code files, each node in the knowledge tree is populated with a natural language summary, as shown in Fig. 5. At the root level, the node embodies the project-level summary. At the directory level, each node represents the directory's summary, encapsulating the responsibilities of the directory and its constituent files. At the file level, each node provides a summary of the file, detailing its attributes, methods, imports, and relationships with other files. Crucially, the context-aware nature of the LLM during the summarization process ensures that the generated natural language summaries are not only human-readable and understandable but also implicitly encode the interconnectedness and relationships between files within the larger repository context. This repository-level awareness, stemming from the LLM's initial exposure to the project's tree structure and comprehensive resources, results in high-quality, information-dense summaries that

facilitate both machine and human comprehension of the codebase. While the knowledge tree provides a structured, hierarchical representation, it is important to acknowledge the presence of **implicit semantic connections** between nodes (as shown by orange arrows in Fig. 5), arising from the summaries' content, which explicitly mentions imports and file dependencies, further enriching the representation.

3.2 Inference Phase

The inference phase marks the operational stage of our methodology, where the hierarchically structured knowledge tree, generated during pre-processing, is leveraged for downstream tasks, specifically bug localization. In contrast to the bottom-up approach of the pre-processing phase, the inference phase adopts a **top-down** traversal strategy. Initiating from the root node representing the project-level summary, the context-aware LLM is tasked with progressively narrowing down the search space to identify the code components most likely implicated in a given bug report. The overall inference process is structured as a hierarchical search, assigning distinct responsibilities to the LLM at both the directory and file levels, executed in a top-down sequence. For bug localization, these responsibilities are defined as **"directory-level search"** and **"file-level search."** In the directory-level search stage (red arrows in Fig. 5), the context-aware LLM is provided with the bug description, which can originate from sources such as GitHub Issues or Jira bug reports, alongside the pre-computed directory summaries. The LLM's objective at this stage is to analyze these inputs and identify the **top-k directories** deemed most pertinent to the bug description, thus representing the most probable areas within the codebase where the bug might reside. Subsequently, the process proceeds to file-level search (blue arrows in Fig. 5). Here, the LLM is furnished with the file-level summaries of code files located within the top-k directories identified in the preceding directory-level search stage. The task at this level is to pinpoint the **top-n code files** within these selected directories that are most likely to be the source of the reported bug or defect.

This initial directory-level search plays a crucial role as a **filtering mechanism**. Instead of directly exposing the LLM to the entirety of code files within the repository, which would be computationally expensive and potentially overwhelm the model with irrelevant information, it strategically focuses the LLM's attention. By first identifying the most relevant directories, the system significantly mitigates the inherent **context window** limitations of LLMs. The subsequent file-level search then operates within a reduced search space, confined to the top-k directories. Furthermore, the file-level search benefits from the enriched context provided by file-level summaries. Rather than naively prompting the LLM with raw, unstructured code files devoid of explicit inter-file and intra-file relationship information, the system presents pre-processed, enhanced summaries. These summaries not only encapsulate the individual functionality of each file but also implicitly encode the critical interconnectivity and relationships between files, derived from the context-aware summarization process. This

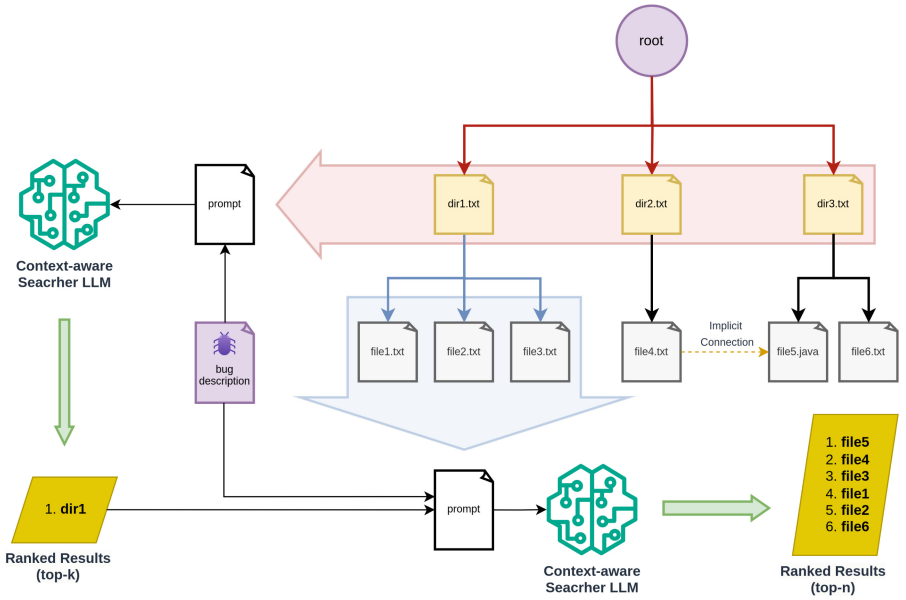

Fig. 5. The knowledge tree is traversed in a top-down manner by a context-aware LLM to identify the most relevant directories (red arrows) and rank the most relevant code files within that directory (blue arrows). (Color figure online)

strategic provision of pre-verified, ready-to-use information obviates the need for the LLM to expend computational resources on understanding the functionality of each code file and deciphering complex inter-file relationships during the inference phase. Instead, the LLM can efficiently focus its analytical capabilities on the core task of bug localization, leveraging the pre-processed knowledge to accurately pinpoint the most probable bug locations.

This hierarchical inference approach, in an abstract sense, draws parallels to the principles of **Chain-of-Thought (CoT)** prompting, a technique recognized for its effectiveness in enhancing LLM performance on complex tasks. CoT prompting advocates for decomposing a complex problem into a sequence of simpler sub-problems, guiding the LLM through a step-by-step reasoning process rather than directly presenting the entire problem at once. Analogously, our bug localization methodology implicitly embodies the spirit of CoT. During the pre-processing phase, we progressively build contextual knowledge: first, creating a context-aware LLM; second, utilizing this LLM to generate granular file-level summaries; and third, aggregating these summaries to form directory-level abstractions. In the inference phase, mirroring the step-by-step reasoning of CoT, the LLM is first tasked with identifying the most relevant directories and subsequently with pinpointing the most probable bug-causing files within those directories. This multi-phase, hierarchical, and step-wise approach inherently aligns with the theoretical underpinnings of CoT prompting, enhancing the effi-

ciency and accuracy of bug localization by guiding the LLM through a structured reasoning process.

4 Experimental Study

To rigorously evaluate the efficacy of our proposed hierarchical summarization methodology for repository-aware bug localization, we conducted a comprehensive experimental study. This section details the experimental setup, including the dataset employed, the baseline methods chosen for comparison, the evaluation metrics utilized, and a discussion of the results obtained. Our study is designed to empirically validate the advantages of our approach in addressing the challenges of domain gap and context window limitations, as well as to assess its overall performance against established code search and bug localization techniques.

4.1 Dataset

For our experimental evaluation, we assembled a dataset of real-world Jira issues, representing bug reports, extracted from a large-scale, industry-level software project. The project's codebase, hosted in a private GitHub repository, provides a standardized solution for the rapid integration of partner products into an existing catalog system. Specifically, it empowers service providers to directly feed partner systems with technical product descriptions, streamlining the product integration process. This codebase is characterized by a microservice architecture and adheres to TM Forum standards, reflecting the complexity and architectural patterns typical of enterprise-level commercial software. The Jira issues within our dataset are particularly valuable as they are authored by end-users, not developers. This provenance introduces a significant domain gap and vocabulary mismatch between the natural language query (the Jira issue description) and the technical vocabulary of the codebase itself. Such end-user reported issues are recognized as among the most challenging for developers to effectively localize and resolve due to this semantic gap. Therefore, this case study provides an ideal benchmark to assess our methodology's capability to overcome the context window limitations of Large Language Models (LLMs), develop a scalable and task-agnostic approach for repository-level code understanding, and explicitly address the domain and vocabulary mismatch problem inherent in similarity-based bug localization, especially when dealing with non-technical end-user reports. The detailed statistics of our dataset are summarized in Table 1.

As indicated in the table, a single Jira issue often necessitates modifications across multiple code files. Solving a typical issue frequently requires developers to identify and modify several files, amplifying the complexity of the bug localization task. The automated system must therefore be capable of locating multiple relevant files within a substantial repository, akin to finding multiple needles within a haystack. This dataset, therefore, serves as a robust and realistic benchmark for evaluating the performance of our proposed system in real-world scenarios.

Table 1. Statistics of our datasest, consisted of queries and codes

Codebase	Repository	Language	Files	Java Files	Lines
	private	Java	743	474	95873
Query	Issue Type	Source	Reporter	Issues	Buggy Files
	Bug	Jira	End-users	9	86

4.2 Baselines

To provide a comparative performance analysis, we benchmarked our proposed methodology against established and state-of-the-art approaches in code search and bug localization. We selected similarity-based methods as a primary baseline, recognizing their widespread adoption and established presence in the field. We categorize this group of approaches as "Flat Retrieval", reflecting their fundamental retrieval-based nature. Specifically, we employed UnixCoder and ModernBERT to generate embeddings for each code file within our repository. Subsequently, we utilized the FAISS [10] library's efficient KNN (K-Nearest Neighbors) algorithm to retrieve the top-k code files most semantically similar to the natural language query (the bug description). As a further comparative baseline, we evaluated systems employing LLMs augmented with Retrieval-Augmented Generation (RAG). We selected Codeium [2] and Cursor AI [1], two state-of-the-art commercial tools widely adopted by developers for repository-level code understanding and "chatting with the codebase" functionalities. Both Codeium and Cursor AI offer capabilities to interact with an entire repository, rather than individual files, and both are understood to utilize RAG techniques to index the codebase and retrieve relevant code segments in response to user queries. While detailed, both systems are black-boxes and precise information regarding the proprietary methodologies employed by Codeium and Cursor AI is not publicly available, their widespread use and claims of RAG and context-aware approaches make them highly relevant benchmarks. Despite the lack of complete transparency in their implementation details, these tools represent the current state-of-the-art in commercially available, repository-aware code understanding systems, making them a compelling comparison point for our research. We term this comparative group "LLM + RAG." To ensure a fair and consistent comparison across LLM-based systems, we selected gpt-4o [15] as the underlying Large Language Model for both our proposed system and the "LLM + RAG" baseline systems, allowing us to isolate the impact of our hierarchical summarization methodology relative to standard RAG approaches.

4.3 Metrics

To quantitatively assess the performance of each approach, we adopted Pass@10 (indicates if at least one relevant file is in the top 10) and Recall@10 (measures the proportion of relevant files found in the top 10) as our primary evaluation metrics. The selection of these metrics is grounded in their established use

within the code search and information retrieval domains, directly aligning with the nature of our bug localization downstream task. Furthermore, given that our benchmark dataset consists of issues that often require modifications across multiple code files, metrics like Recall@10 are particularly informative, indicating the extent to which the relevant files are successfully retrieved within the top-ranked results. Complementarily, Pass@10 provides a valuable measure of practical utility. It evaluates whether at least one file relevant to the bug is present within the top 10 retrieved files. In the context of large, enterprise-level repositories, even retrieving a single correct file within the top 10 results offers significant value to developers, effectively reducing their search space from thousands of files to a manageable set of just ten, thereby substantially accelerating the bug localization process.

4.4 Results

For the "Flat Retrieval" baseline, we directly queried the system using the raw Jira issue description as input. For the LLM-based approaches, including our system and the "LLM + RAG" baselines, we prompted the models with the raw Jira issue description, supplemented by a concise instruction explicitly requesting the models to return the top-10 files most relevant to the provided bug description. The quantitative results of this experimental evaluation, comparing the performance of our hierarchical summarization methodology against the baselines across the chosen metrics, are presented in Table 2.

Table 2. The results of experimental study using three different approaches

Approach	Model	Pass@10	Recall@10
Flat Retrieval	ModernBERT	0,22	0,11
Flat Retrieval	UnixCoder	0,22	0,14
LLM + RAG (Codeium)	gpt-4o	0.67	0.24
LLM + RAG (Cursor AI)	gpt-4o	0.67	0.26
Ours	gpt-4o	**0.89**	**0.33**

The results presented in Table 2 clearly demonstrate the superior performance of our proposed hierarchical summarization methodology compared to both Flat Retrieval and LLM + RAG baseline approaches across both Pass@10 and Recall@10 metrics. Specifically, our system, utilizing gpt-4o and hierarchical summarization, achieves a Pass@10 score of **0.89** and a Recall@10 score of **0.33**, significantly outperforming the best-performing baseline, LLM + RAG (Cursor AI), which achieves a Pass@10 of 0.67 and Recall@10 of 0.26. The Flat Retrieval baselines, using both ModernBERT and UnixCoder, exhibit considerably lower performance, with Pass@10 scores of 0.22 and Recall@10 scores of 0.11 and 0.14 respectively. These findings strongly suggest that our structure-aware, hierarchical approach effectively leverages the capabilities of LLMs for repository-level

code understanding, leading to significantly improved bug localization accuracy. The substantial performance gains over both traditional similarity-based methods and contemporary LLM + RAG systems highlight the effectiveness of hierarchical summarization in addressing the challenges of domain gap and context window limitations inherent in bug localization within large code repositories, particularly when dealing with end-user reported issues.

5 Conclusion

This paper has presented a novel and effective methodology for enhancing Large Language Model (LLM) performance in repository-aware bug localization through hierarchical summarization. Our structure-aware approach addresses critical limitations of existing code search techniques, particularly the domain and vocabulary mismatch inherent in end-user bug reports and the context window constraints of LLMs when applied to large codebases. By introducing a hierarchical summarization technique, we enable the creation of context-aware LLMs capable of understanding and reasoning about code at the repository level, facilitating more accurate and scalable bug localization. Empirical evaluation on a real-world, industry-scale dataset demonstrates the significant performance gains achieved by our methodology compared to Flat Retrieval and LLM + RAG baselines, showcasing a substantial improvement in both Pass@10 and Recall@10 metrics. These results underscore the efficacy of hierarchical summarization in bridging the semantic gap and mitigating context window limitations, leading to more effective bug localization, especially for complex, enterprise-level software projects and user-reported issues. Furthermore, the task-agnostic nature of our approach suggests its potential applicability to a broader range of software engineering downstream tasks requiring repository-level code comprehension. Future work will focus on exploring further refinements to the summarization process, investigating dynamic adaptation of the hierarchical structure, and extending the evaluation to diverse codebases and downstream tasks to fully realize the potential of repository-aware LLMs in software engineering practice.

Acknowledgments. This manuscript underwent limited AI post-processing solely for enhanced readability and grammatical accuracy, with no alteration to its original content, conceptual framework, or research findings. This research was conducted as part of the R&D efforts within the TBTAK TEYDEB 1501 project, grant number 3240105, and gratefully acknowledges the support of **The Scientific and Technological Research Council of Turkey (TÜBTAK)**.

A Appendix

A.1 Prompts

Figure 6 illustrates sample prompt templates employed in this study. The left box presents a template used for generating code file summaries (file-level summaries) during the pre-processing phase. Conversely, the right box displays a sample

prompt template designed for localizing buggy files within a specific directory (file-level search). It is important to note that these templates are provided as examples; the actual implementation utilized specific and optimized instructions tailored to our particular codebase.

Summarization Prompt Template

```
======================
Your Tasks
======================
As an expert software engineer, summarize the provided code file for bug
localization. Your concise summary should cover the file's purpose,
functionality, dependencies, key data structures/algorithms, potential error
areas, contextual information within the project, and key code elements like
classes and main functions. This information will aid another LLM in
efficiently identifying bug locations. You have access to the project
summary and the repository tree as hints to guide you to generate more
informative summaries.

======================
Project Summary (high_level summary)
======================
{project_summary}

======================
Repository Tree
======================
{repository_tree}

======================
Code File to Summarize
======================
{code_file}
```

Bug Localization Prompt Template

```
======================
Your Tasks
======================
As a senior software engineer specializing in {Java} code search and bug
localization, your task is to identify the top 10 most relevant files to a given
bug ticket within a large {Java} codebase. Utilize the provided repository
tree, project summary, and project code summaries as resources to guide
your search. Your output should be a ranked list of the top 10 {Java}
methods with a similarity score (1-10) and a brief justification for each,
ordered by relevance.

======================
Bug Ticket
======================
{bug_ticket}

======================
Repository Tree
======================
{repository_tree}

======================
Project Summary (high_level summary)
======================
{project_summary}

==============================================
Project Code Summary (file-level summaries)
==============================================
{code_file_summaries}
```

Fig. 6. Example prompt templates for summarizing code files (left) and localizing buggy code files (right)

A.2 Version Information

For code interaction, we utilized the Codeium v1.38 JetBrains extension, specifically its "Chat with Codebase" mode, released on February 13, 2025. Additionally, we employed Cursor IDE version 0.46.x, released on February 19, 2025, in its "Ask Codebase" mode.

References

1. Cursor - The AI Code Editor — cursor.com. https://www.cursor.com/en. Accessed 05 Mar 2025
2. Windsurf editor and codeium extensions — codeium.com. https://codeium.com/. Accessed 05 Mar 2025
3. Aktas, M.S., Fox, G.C., Pierce, M.: Fault tolerant high performance information services for dynamic collections of grid and web services. Futur. Gener. Comput. Syst. **23**(3), 317–337 (2007)

4. Aydin, G., Aktas, M.S., Sayar, A.: ServoGrid complexity computational environments CCE integrated performance analysis. In: 2005 6th International Workshop on Grid Computing (GRID), pp. 256–261 (2005)
5. Bairi, R., et al.: CodePlan: repository-level coding using LLMs and planning. In: Proceedings of the ACM on Software Engineering **1**(FSE), pp. 675–698 (2024)
6. Chen, J., et al.: CoreQA: uncovering potentials of language models in code repository question answering. arXiv preprint arXiv:2501.03447 (2024)
7. Chen, Y., et al.: When large language models confront repository-level automatic program repair: how well they done? In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, pp. 459–471 (2024)
8. Di Grazia, L., Pradel, M.: Code search: a survey of techniques for finding code. ACM Comput. Surv. **55**(11), 1–31 (2023)
9. Ding, Y., et al.: CoCoMIC: code completion by jointly modeling in-file and cross-file context. arXiv preprint arXiv:2212.10007 (2022)
10. Douze, M., et al.: The Faiss library. arXiv preprint arXiv:2401.08281 (2024)
11. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. Findings of the association for computational linguistics: EMNLP 2020 (2020)
12. Guo, D., et al.: UnixCoder: unified cross-modal pre-training for code representation. arXiv preprint arXiv:2205.07754 (2022)
13. Guo, J., Wang, C., Xu, X., Su, Z., Zhang, X.: RepoAudit: an autonomous LLM-agent for repository-level code auditing. arXiv preprint arXiv:2501.18160 (2024)
14. Huang, R., et al.: AudioGPT: understanding and generating speech, music, sound, and talking head. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 38, pp. 23802–23804 (2024)
15. Hurst, A., et al.: GPT-4o system card. arXiv preprint arXiv:2410.21276 (2024)
16. Jain, S., Dora, A., Sam, K.S., Singh, P.: LLM agents improve semantic code search. arXiv preprint arXiv:2408.11058 (2024)
17. Li, H., Zhou, X., Shen, Z.: Rewriting the code: a simple method for large language model augmented code search. arXiv preprint arXiv:2401.04514 (2024)
18. Ma, Y., Yang, Q., Cao, R., Li, B., Huang, F., Li, Y.: How to understand whole software repository? arXiv preprint arXiv:2406.01422 (2024)
19. McKinzie, B., et al.: Mm1: methods, analysis and insights from multimodal LLM pre-training. In: European Conference on Computer Vision, pp. 304–323. Springer (2024). https://doi.org/10.1007/978-3-031-73397-0_18
20. Nacar, M.A., Aktas, M.S., Yuen, D.A.: VLAB: collaborative grid services and portals to support computational material science. Concurr. Comput. Pract. Exp. **19**(12), 1717–1728 (2007)
21. Olmezogullari, E., Aktas, M.S.: Representation of click-stream data sequences for learning user navigational behavior by using embeddings. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 3173–3179 (2020)
22. Olmezogullari, E., Aktas, M.S.: Pattern2Vec: Representation of clickstream data sequences for learning user navigational behavior. Concurr. Comput. Pract. Exp. **34**(9) (2022)
23. Oskooei, A.R.: On the use of data parallelism technologies for implementing statistical analysis functions. In: The 14th International Workshop on Computer Science and Engineering (WCSE 2024), pp. 94–102 (2024)
24. Oskooei, A.R., Babacan, M.S., Yağcı, E., Alptekin, Ç., Buğday, A.: Beyond synthetic benchmarks: assessing recent LLMs for code generation. In: The 14th Inter-

national Workshop on Computer Science and Engineering (WCSE 2024), pp. 290–296 (2024)
25. Qin, Y., et al.: AgentFL: Scaling LLM-based fault localization to project-level context. arXiv preprint arXiv:2403.16362 (2024)
26. Rafiei Oskooei, A., Aktaş, M.S., Keleş, M.: Seeing the sound: multilingual lip sync for real-time face-to-face translation. Computers **14**(1), 7 (2024)
27. Sahinoglu, M., Incki, K., Aktas, M.S.: Mobile application verification: a systematic mapping study. In: Gervasi, O., et al. (eds.) ICCSA 2015. LNCS, vol. 9159, pp. 147–163. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21413-9_11
28. Shen, Z., et al.: SlimPajama-DC: understanding data combinations for LLM training. arXiv preprint arXiv:2309.10818 (2023)
29. Tufek, A., Gurbuz, A., Aktas, M.S.: Provenance collection platform for the weather research and forecasting model. In: 2018 14th International Conference on Semantics, Knowledge and Grids (SKG), pp. 17–24 (2018)
30. Wang, Y., et al.: You augment me: Exploring ChatGPT-based data augmentation for semantic code search. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 14–25. IEEE (2023)
31. Warner, B., et al.: Smarter, better, faster, longer: a modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. arXiv preprint arXiv:2412.13663 (2024)
32. Zhang, F., et al.: RepoCoder: repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570 (2024)
33. Zhang, W., et al.: GraphCoder: enhancing repository-level code completion via code context graph-based retrieval and language model. arXiv preprint arXiv:2406.07003 (2024)
34. Zhang, Y., Ruan, H., Fan, Z., Roychoudhury, A.: AutoCodeRover: autonomous program improvement. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1592–1604 (2024)