

ToCoLo: A File-level Context-aware Approach for User Review-based Code Localization

KUO CHI and CHUANYI LI*, State Key Laboratory of Novel Software Technology, Nanjing University, China

ZHOU YANG, Singapore Management University, Singapore

CHANGAN NIU and BIN LUO, State Key Laboratory of Novel Software Technology, Nanjing University, China

DAVID LO, Singapore Management University, Singapore

VINCENT NG, Human Language Technology Research Institute, University of Texas at Dallas, USA

Platforms like the Google Play Store have become the backbone of the mobile internet era. User reviews play a key role in the development of mobile applications. These reviews help developers identify issues, suggest new features, and report bugs, which drives continuous app improvement and platform evolution. An important research problem has emerged: *given a user review, how to automatically identify relevant code snippets from the codebase to assist developers?* Existing solutions typically compute similarity between user reviews and individual method code snippets in isolation. However, highly similar code snippets (e.g., code clones) are widespread across the codebase. Although such snippets may appear semantically similar, their actual relevance strongly depends on the file-level and contextual usage in which they reside. By ignoring this contextual distinction, existing approaches are easily misled by similar but irrelevant methods, resulting in incorrect or unreliable code localization results. This observation motivates us to propose ToCoLo (Two-Stage Approach for User Review-based Code Localization), a two-stage approach that explicitly incorporates file-level context to constrain method-level localization. In the first stage, we leverage an LLM to generate natural language summaries for code files and use them to identify a small set of candidate files that are most relevant to a given user review. In the second stage, we perform fine-grained method-level matching only within the retrieved files, and return the most relevant code snippets as localization results. Our experimental evaluation demonstrates that ToCoLo consistently improves user review-based code localization across different backbone models. With UnixCoder as the backbone, ToCoLo achieves absolute improvements of 8.3% in Top-1 accuracy, 7.0% in Top-3, 4.5% in NDCG@3, and 6.9% in Mean Reciprocal Rank (MRR) compared to the one-stage baseline. Using GraphCodeBERT, ToCoLo attains a Top-1 gain of 6.0%, along with 4.1% in NDCG@3 and 6.2% in MRR. These results indicate that ToCoLo is more effective at ranking truly relevant code snippets at the very top of the localization results. Moreover, to measure the effectiveness of ToCoLo in mitigating interference from similar method code snippets, we introduce a new evaluation metric, the *Similar Mislocalization Rate (SMR)*. SMR quantifies, from the user review perspective, how often localization results are dominated by different but similar code snippets that are incorrectly ranked above the Ground Truth. Experimental results show

*Corresponding Author.

Authors' Contact Information: Kuo Chi, chikuo@smail.nju.edu.cn; Chuanyi Li, lcy@nju.edu.cn, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Zhou Yang, zy25@ualberta.ca, Singapore Management University, Singapore, Singapore; Changan Niu, niu.ca@outlook.com; Bin Luo, luobin@nju.edu.cn, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore, Singapore; Vincent Ng, vince@hlt.utdallas.edu, Human Language Technology Research Institute, University of Texas at Dallas, Richardson, Texas, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-7392/2025/10-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

that ToCoLo substantially reduces SMR (e.g., from 6.7% to 0.89% and from 7.7% to 1.3%), demonstrating that ToCoLo effectively mitigates similar code interference and yields more reliable localization.

CCS Concepts: • **Software and its engineering** → **Software evolution**.

Additional Key Words and Phrases: AI4SE, Requirements Engineering, Change Localization

ACM Reference Format:

Kuo Chi, Chuanyi Li, Zhou Yang, Changan Niu, Bin Luo, David Lo, and Vincent Ng. 2025. ToCoLo: A File-level Context-aware Approach for User Review-based Code Localization. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (October 2025), 33 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Recent reports [49] predict that by 2025, the global number of mobile users will reach 7.49 billion. The growing demand for mobile applications, driven by the increasing feedback of users, has led to the need for frequent app updates. It is reported that 28% of apps are updated weekly, while 72% receive updates monthly [50]. This highlights the continuing need for app maintenance and optimization, which will drive rapid growth on platforms. Users' feedback, often in the form of user reviews (URs), plays a critical role in software development, testing, and maintenance. Users provide valuable feedback on applications through platforms like the Google Play Store [17, 32, 37, 41, 56]. For example, user reviews can help developers identify overlooked functional needs in an application or areas where users expect improvements. McIlroy et al. [32] reported that developers in apps first read user reviews, then resolve the issues reported in the user reviews and update the apps. Zhang et al. [61] investigate top 200 most active developers in top 100 popular mobile apps, and find that 85.7 percent responders depend on user reviews to find and resolve issues from source code. Developers can use this feedback to make informed decisions during development planning. This evidence supports a practical workflow in real-world development: developers first identify valuable reviews, then locate the relevant code, and finally update the application accordingly.

However, the sheer volume of user reviews can be overwhelming [14, 31]. While some reviews may be less informative, a considerable subset contains actionable feedback. Prior studies have categorized user reviews into several types, most notably Problem Discovery (PD) and Feature Request (FR), corresponding to bug reports and enhancement requests, respectively [30]. For instance, a user mentioned in a review of the *AntennaPod* mobile application: "*It's a nice app. I wish you could do Playlists (multiple) instead of just one queue,*" which is a typical FR explicitly asking for support for multiple playlists. Another user mentioned in a review of the *Anki* mobile application: "*I download the shared flash cards and it shows 0 of 153 cards added Idk y this happens plz fix this bug,*" which is a PD indicating a failure to correctly import shared flash card content, making the flash cards unusable. In real-world development scenarios, developers typically first manually identify valuable reviews based on criteria such as clear problem descriptions, reproducibility, impact severity, and relevance to recent software versions. These selected reviews can then be used to locate relevant code snippets for implementing new features or addressing issues. Motivated by this practical workflow, our research focuses on a key challenge: *How to automatically locate relevant code fragments in a software codebase given user reviews that have been identified as valuable?* To address this challenge, the task of user review-based code localization has been proposed [6], aiming to automatically map user feedback to relevant code snippets within a software project. Depending on the granularity of localization, this task can be categorized into method-level and file-level localization, each suited for different development and maintenance scenarios. In this task, we directly associate user reviews with code elements rather than relying on issues and pull requests (PRs), for two main reasons. The first is the limited coverage of issue/PR-based linkage. Haering et al. [11] found that nearly 47% of reviews appear before any related issue is created. Thus,

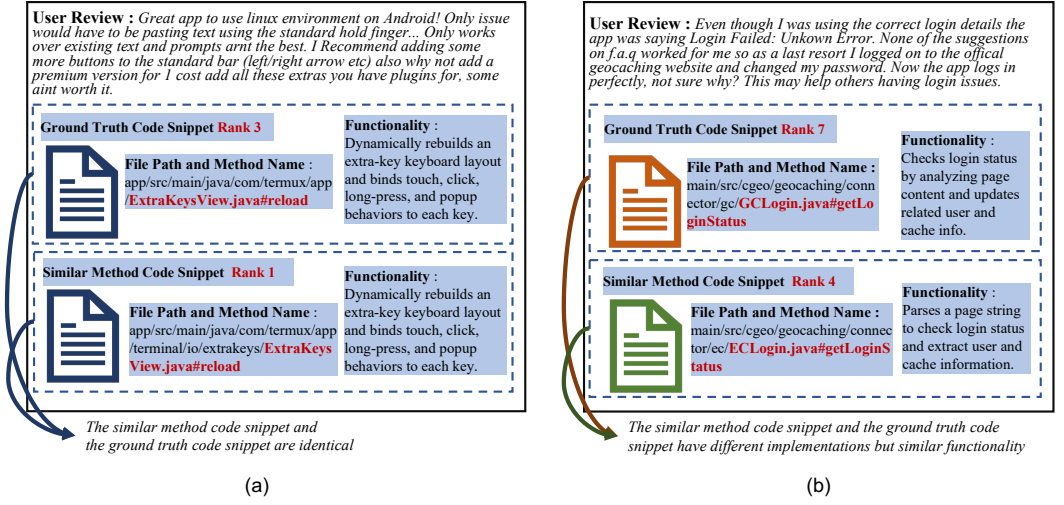


Fig. 1. High similarity between *similar method code snippets* and the *ground-truth snippet* leads to mislocalization, resulting in the ground-truth snippet receiving a lower ranking

relying on issues would exclude a large portion of valuable feedback, as developers often skip creating issues for lower-priority or hard-to-reproduce reviews, and issue/PR practices vary widely across projects, with many lacking systematic review tracking. The second reason is to reduce bottlenecks and improve timeliness. Khatoonabadi et al.[20] analyzed 265,325 PRs from ten major GitHub projects and showed that 4,450 valid PRs were abandoned due to missing timely feedback, leading to the loss of intended contributions. This demonstrates that passing through intermediate stages like issues and PRs makes the process fragile and delay-prone. By directly linking reviews to methods or classes, we eliminate such bottlenecks and ensure feedback flows immediately into code localization, thereby enhancing automation and responsiveness—crucial for rapid-iteration mobile app development.

A critical challenge in user review-based code localization is **similar method interference**: method-level similarity alone often fails to distinguish between functionally similar methods located in different files. Existing user-review-based code localization methods all follow a one-stage design that directly matches reviews to method snippets. However, one-stage approaches fundamentally lack file-level context, making them vulnerable to similar snippet confusion: multiple methods in different files share highly similar lexical or semantic patterns, leading to consistent mislocalization. Mislocalizing similar methods can lead developers to inspect incorrect code, wasting time and potentially introducing errors. This issue has never been systematically addressed in prior work. Empirical studies on large-scale Android applications provide strong evidence that similar or identical functionality often appears across different files. Linares-Vásquez et al. [27] show that many classes contain repeated or similar code. Furthermore, source-level clone detection with DroidCC [1] confirms that code clones commonly occur across files, indicating that developers frequently implement functionally related methods in multiple locations. This problem is widespread in large-scale Android applications with duplicated or boilerplate code, affecting a significant portion of user reviews. For example, file A may handle data storage while file B manages user interactions, yet both files may contain methods for input validation or data format conversion. We refer to such cases as **similar method code snippets**. Ignoring file-level context can cause interference

during code localization. As shown in Figure 1(a), a *similar method code snippet* identical to the *ground truth method snippet* but located in a different file can receive a higher similarity score than the correct snippet, displacing it in the ranking. Similarly, Figure 1(b) illustrates a *similar method code snippet* that is not identical but functionally related to the ground truth snippet; it still receives a high similarity score, potentially affecting localization accuracy. To quantify this effect, we analyzed all 678 user reviews in the Review2Code test dataset using the one-stage approach of Hu et al., measuring how often *similar method code snippets* interfered with the correct localization. In 7.71% of cases, *similar method code snippets* were ranked higher than the correct code. This issue was even more pronounced in the AntennaPod app, where 11.67% of reviews suffered from similar interference. These observations indicate that method-level similarity alone is insufficient for accurate code localization in modular, large-scale applications, highlighting the need to incorporate higher-level contextual information, such as file-level structure, into the localization process.

To mitigate the interference caused by similar method code snippets across files, we propose a two-stage user review-based code localization framework named ToCoLo, inspired by the recall-and-ranking paradigm in information retrieval [18, 21, 38, 57]. While two-stage retrieval-ranking paradigms are widely adopted in general IR and code search, none of the existing approaches for user review-based code localization employ this design. This design specifically addresses the interference that one-stage approaches often encounter when similar methods appear across multiple files. In the first stage (recall phase), we leverage a large language model (LLM) to generate structured natural language descriptions for each source code file, following a predefined template that summarizes functionality, role, and typical modification scenarios. This structured generation reduces stylistic randomness and enforces a consistent description format across files, providing a reliable semantic representation that facilitates more accurate retrieval of relevant files for a given user review. By incorporating file-level context, the recall phase effectively filters out irrelevant files, mitigates interference from similar but incorrect method snippets, and reduces computational complexity for the subsequent ranking stage. In the second stage (ranking phase), all method-level code snippets within the retrieved files are considered as candidates. Each user review and candidate snippet is encoded into vector representations via an encoder, and relevance is determined through similarity computation. Candidate snippets are then ranked based on their similarity scores, and the top-ranked snippet is selected as the final localization result.

We evaluated the effectiveness of ToCoLo on the Review2Code dataset, comparing it with the one-stage approach [15]. Top-1 accuracy, NDCG@3, and Mean Reciprocal Rank (MRR) are critical metrics for evaluating user review-based code localization because they measure a model's ability to rank truly relevant code snippets at the very top of the retrieval results, directly reflecting users' concern for the top few results and influencing efficiency and satisfaction in real-world usage. With UNIXCODER as the backbone, ToCoLo achieves absolute improvements of 8.3% in Top-1 accuracy, 7.0% in Top-3, 4.5% in NDCG@3, and 6.9% in MRR compared to the one-stage baseline. Using GRAPHCODEBERT, ToCoLo attains a Top-1 gain of 6.0%, along with 4.1% in NDCG@3 and 6.2% in MRR. To further quantify a critical failure mode overlooked by conventional metrics, we introduce the *Similar Mislocalization Rate (SMR)*, which measures the proportion of reviews in which the model ranks different but highly similar method code snippets above the ground truth; higher SMR values indicate greater susceptibility to mislocalization. Experimental results show that ToCoLo substantially reduces SMR compared to the one-stage baselines: for the UnixCoder backbone, SMR drops from 6.73% to 0.89%, and for GraphCodeBERT, from 7.71% to 1.32%. This reduction demonstrates that ToCoLo not only improves top-ranking accuracy but also minimizes the risk of misleading users with "lookalike but incorrect" code snippets. By addressing this overlooked risk, ToCoLo provides direct practical value in real-world code localization scenarios.

In summary, our contributions of this work are as follows:

- We propose ToCoLo, a two-stage recall-and-ranking framework for the user review-based code localization task. Our design explicitly addresses interference caused by similar method code snippets across files, and existing localization methods can be readily incorporated into this framework.
- We identify the interference caused by highly similar method code snippets across different files as a critical limitation of existing one-stage localization approaches. To systematically analyze this issue, we introduce a diagnostic evaluation metric, the *Similar Mislocalization Rate (SMR)*, and show that ToCoLo effectively mitigates such interference.
- Extensive experiments on two backbone models demonstrate that ToCoLo consistently improves ranking quality, achieving significant gains in Top-1, MRR, MAP and NDCG over one-stage baselines on the user review-based code localization task.

2 Related Work

2.1 User Review

User reviews play a critical role in software development, testing, and maintenance practices. [37, 41, 55, 56], especially in DevOps [24, 64]. Specifically, during the release planning phase and development activities, user reviews can expose specific problems encountered by users in the use process, such as delays, inconvenience, crashes, etc., which provide the most direct feedback to the development team and help to solve problems with a wide range of impacts [7]. In terms of the phase of testing activities, a critical part of quality assurance for a new version of code, problems, or crash scenarios mentioned in user reviews provides the test team with real usage scenarios, from which the importance of the user review can be seen [13, 40]. Consequently, the widespread use of user reviews throughout the development process provides both the theoretical foundation and practical motivation for the code localization approach proposed in this paper.

2.2 Bug localization

Bug localization aims to identify faulty source code given a bug report [16, 22, 26, 60]. Early IR-based work heavily relied on the Vector Space Model (VSM) due to its simplicity and reasonable effectiveness. For instance, BugLocator [62] uses a revised VSM to compute similarity between bug reports and source files, while Locus [54] further explores matching bug reports to fine-grained changes (hunks). Topic-model approaches [34, 42] and co-change analysis (e.g., RAT [36]) also demonstrate the benefits of leveraging historical bug-fixing patterns. Beyond classical IR-based methods, some studies incorporate additional signals to improve bug localization. For example, AmaLgam [51] combines version history, similar bug reports, and code structure, while Cross-Task Artifact Dependencies [2] leverages developer interactions to capture co-modified files. With the rise of deep learning, many models further enhance textual-code matching through neural representations, transfer learning, or multi-modal signals, such as TRANP-CNN [23], COOBA [65], and Bug2Commit [33].

Although these approaches provide a strong foundation for text-code relevance modeling, bug localization and user-review-based code localization differ in fundamental ways, making direct adaptation non-trivial. Unlike bug reports, which are typically written by developers or technical staff and contain structured diagnostic information such as stack traces, reproduction steps, and error logs, user reviews are informal, ambiguous, and often lack explicit technical cues. This substantial gap in linguistic style, informativeness, and intent makes user-review-based code localization a far more context-dependent challenge.

2.3 Code Localization

Many approaches [6, 15, 39, 59, 61, 63] have been proposed to address the user review-based code localization problem. URR [6] classifies user reviews and source code files into predefined categories and then uses a retrieval tool to search and return the highest-scoring source files. ChangeAdvisor [39] groups user reviews that express similar needs, then uses cosine similarity to measure the similarity between source code snippets and these clusters. Where2Change [61] is similar to ChangeAdvisor [39] in clustering and similarity calculation, but it additionally measures the similarity between user review clusters and problem reports, which are detailed descriptions of software issues submitted by users or developers to help fix bugs. ReviewSolver [59] leverages contextual information in user reviews and correlates reviews with the bytecode of the app to locate functional errors in mobile apps. RISING [63] is similar to ChangeAdvisor [39] and Where2Change [61], with two main steps: clustering and similarity calculation. Additionally, it uses commit messages to bridge the lexical gap between user reviews and source code snippets. Most recently, Hu et al. [15] propose a large-scale, manually labeled (User Review, Code) dataset named Review2Code, consisting of 68,359 pairs of user reviews and their corresponding code, they evaluated the above methods, as well as CodeBERT and GraphCodeBERT on the Review2Code dataset, with CodeBERT and GraphCodeBERT surpassing all other methods.

2.4 Code File Description Generation

Code understanding is an important research direction in software engineering and artificial intelligence. In recent years, code summarization [8, 10, 29, 53] and code documentation generation [3, 12, 52] have received widespread attention for improving code readability and maintenance efficiency. The code summarization task aims to generate concise natural language descriptions for code snippets, helping developers quickly understand code functionality. Most studies focus on method-level summarization and file-level summarization. However, these methods are still limited to the content of the code itself and cannot provide information about the role of a code file within the entire project. The code documentation generation task aims to generate and maintain repository-level code documentation, facilitating developers' quick understanding of the code repository. Two recent works demonstrate the benefits of incorporating project-level context into code summarization. ProConSuL [28] explicitly provides large language models with project-level context, such as file dependencies and call graphs, and shows that LLMs can generate more accurate and reliable code summaries when they are aware of a code file's position and relationships within the repository. Code-Craft [48] constructs hierarchical representations of the entire codebase, capturing parent-child and module-level relationships between files. Experiments show that using such hierarchical context improves the quality of summaries and the effectiveness of downstream retrieval tasks. Inspired by these works and the significant advancements of large language models (LLMs) in code understanding and generation tasks [5, 19, 28, 35, 44, 58], we leverage project-level information to provide structured context to LLMs for code file description generation. Specifically, we parse the directory structure of a project and encode the hierarchical relationships between files, then prompt an LLM to generate structured natural language descriptions that summarize a file's functionality, role within the project, and typical modification scenarios. This approach aims to produce semantically rich and consistent file-level descriptions, which can facilitate more accurate retrieval of relevant code files for a given user review.

3 Method

In this section, we provide the design details of two-stage approach of user Review-based code localization. Through the first stage of file recall, the search scope can be efficiently narrowed and

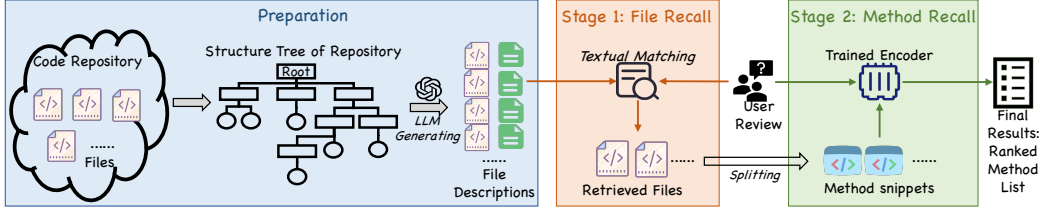


Fig. 2. The framework of two-stage code localization approach ToCoLo

the impact of *similar method code snippets* can be avoided, while in the second stage of ranking, the accuracy of code positioning can be further improved through similarity calculation and ranking.

3.1 Overview

The one-stage approach [15] for user review-based code localization generally follows these steps: (1) The files in a GitHub project are divided into storage units at the method-level granularity and stored in the codebase. (2) When a user review arrives, a fine-tuned encoder is used to obtain vector representations of the review and all code snippets in the codebase. (3) By computing similarity scores, the best-matching code localization result is returned at the method-level. The one-stage approach bypass file-level code localization and directly performs method-level localization. However, this approach introduces several issues. First, directly locating relevant code among all methods results in a high computational cost, as each user review must be compared with every code snippet in the codebase. While methods are the smallest executable units in a codebase, they often lack sufficient context to convey complete semantics. Second, a functionally similar method from a different file may be ranked higher than the actual relevant method that truly matches the user review.

To address the key limitations of existing one-stage methods, particularly the interference caused by functionally similar code snippets across multiple files, we propose a two-stage user review-based code localization approach named ToCoLo (Figure 2). It consists of a file-level recall stage and a method-level ranking stage. Before the first stage, we use a large language model (LLM) to generate natural language descriptions for source code files, summarizing their functionality, role, and modification scenarios. Unlike Retrieval-Augmented Generation (RAG) frameworks [25] that generate answers directly from retrieval, our LLM is only used to produce structured file summaries. This design enforces consistent descriptions, reduces stylistic randomness, and ensures that downstream vector-based ranking remains deterministic and reproducible. In the first stage, we treat user reviews as queries and the generated file descriptions as documents, and perform traditional information retrieval using the BM25 algorithm to retrieve relevant code files. In the second stage, all method-level code snippets from the retrieved files in the first stage are considered as candidates. We fine-tune a model to serve as an encoder using $\langle \text{user review}, \text{method snippet} \rangle$ pairs from the Review2Code [15] training set. Each user review and candidate snippet is encoded into vector representations, and their relevance is determined through similarity computation.

This approach effectively addresses existing issues. The advantages of this approach include: (1) Distinguishing *similar method code snippets* using file-level information. By incorporating file-level context, the model can differentiate between functionally similar methods from different files, reducing misidentifications, and improving localization accuracy. (2) Method-level matching for fine-grained localization. Once the search space is reduced, method-level matching ensures higher accuracy in code localization. (3) File-level matching as a pre-filtering step. It helps narrow the

search space, reducing computational cost and improving efficiency. This combined approach enhances both the efficiency and precision of user review-based code localization.

3.2 Code Files Description Generation

In this step, to provide the large language model with meaningful contextual information about a project, we first parse the project's directory structure and construct hierarchical relationships between code files, including parent-child and module-level associations. This design is inspired by prior works such as ProConSuL [28] and Code-Craft [48], which demonstrate that providing LLMs with project-level context—such as hierarchical structure, module relationships, and call graphs—enables more accurate and context-aware code summarization. Rather than simply generating summaries for individual code snippets, we leverage a structured representation of the entire project or repository to capture each file's role and relationships within the broader codebase. This hierarchical context helps the model understand the relative importance and function of each file within the overall project architecture. For example, in a Model-View-Controller (MVC) architecture, files in the Controller layer are typically associated with user interactions, whereas files in the Service layer implement business logic; capturing these relationships allows the LLM to generate more accurate and task-relevant file descriptions. Specifically, we recursively traverse the root directory of the project to identify all source code files and record their relative paths. We then reconstruct a tree structure in which each directory is represented as a node and its child nodes correspond to subdirectories or contained code files. For instance, the file path `src/controller/UserController.java` is represented as a leaf node under the path `src` → `controller` → `UserController.java`. By providing the LLM with this structured hierarchical information, we ensure that the generated file descriptions are context-aware and reflect both the individual file's purpose and its position within the project.

To produce file-level semantic descriptions for retrieval, we leverage a large language model (ChatGPT, GPT-5¹) to generate structured natural language descriptions for each source code file. The goal is to provide a semantically rich and consistent representation of files. Following insights from prior work on prompt engineering [46], prompt structure and design critically influence output quality, stability, and task alignment. Since prompts can significantly affect LLM outputs, we adopt a structured prompt template to generate file descriptions in order to ensure consistency, comparability, and reproducibility across different files and projects. For a discussion of the effects of different prompt versions and LLM choices, see Section 5.3. This structured template guides the model to produce summaries along three task-relevant dimensions, reducing stylistic variability and aligning the outputs with aspects relevant to user reviews:

- (1) **File function:** What is the primary purpose of this file? This information helps align the file with the functional requirements described in the user review.
- (2) **File role:** What role does this file play within the project? Understanding the file's role in the overall architecture helps the model avoid matching irrelevant files.
- (3) **Modification scenarios:** What circumstances would this file typically be modified? This provides contextual cues about when the file is likely to be changed.

As shown in Figure 3, we use the *Anki* project as an example. First, we input the project name "Anki" and the project structure tree into the prompt for ChatGPT. ChatGPT then outputs its understanding of the various components and modules within the project. Next, we input the project name "Anki", the path to a code file, and the names and parameters of all methods defined in that code file into the prompt for ChatGPT, which then generates a description of the corresponding code file.

¹<https://platform.openai.com/docs/models/gpt-5>

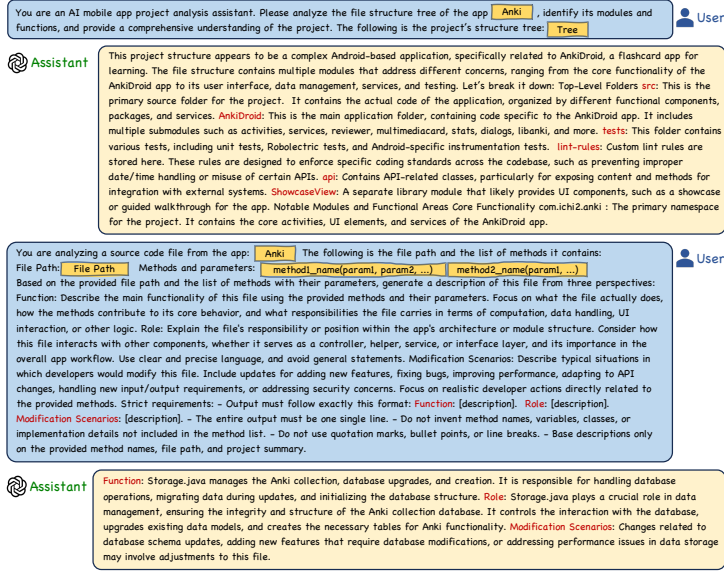


Fig. 3. Using the LLM to generate a description for a code file

We note that, although the LLM may have been pretrained on publicly available code repositories that could partially overlap with our dataset, any potential effects on the results are discussed in the Internal Validity section of Section 6, where we also describe our mitigation strategies.

3.3 Code Files Recall

In this step, we treat the user review as a query and the LLM-generated file descriptions as documents, retrieving candidate files using the BM25 [43] algorithm. BM25 is particularly suitable in our setting for two main reasons. First, efficiency: it allows fast retrieval from potentially large codebases without requiring expensive encoding or attention computations for all files, which is especially important in projects with hundreds or thousands of files. It also avoids the need to fine-tune additional models for retrieval. Second, effectiveness with structured summaries: BM25's term-frequency (TF) and inverse-document-frequency (IDF) based scoring robustly captures lexical overlaps between the review and the LLM-generated file descriptions, even when the phrasing differs. Since our descriptions are structured along the task-relevant dimensions of function, role, and modification scenarios, lexical matching is sufficient to identify files whose content aligns with the review. The most relevant files are recalled from the entire project, forming the candidate file set for subsequent method-level matching.

In the basic BM25 calculation, a document's score typically depends on factors such as the frequency of query terms in the document, the document's length, and the inverse document frequency. Inspired by prior work on refactoring-aware traceability models for bug localization [36], cross-task artifact dependency mining [2], and AmaLgam [51], which leverage version history to improve localization performance, we observe that files frequently modified in historical commits are more likely to be modified again. Based on this insight, we design an additional weighting mechanism to enhance document ranking. Specifically, we assign a weight to each file based on its modification frequency in the dataset, increasing the likelihood of frequently modified files being retrieved. For example, if a file has been submitted multiple times, our weighted algorithm

will make it more likely to be retrieved compared to files with fewer submissions. We analyze the samples in the training set by counting the number of times each file has been committed based on the "commit_id" field. As a result, we obtain a dictionary where the keys are file paths, and the values represent the number of times each file has been committed.

Directly using this weight in the calculation may cause extreme values to disproportionately impact retrieval ranking, as some methods may have significantly more submissions than others. To mitigate this issue, we applied a smoothing process to reduce extreme differences in weights, ensuring that frequently submitted methods do not overly dominate the BM25 score while still maintaining their influence. We experimented with three smoothing approaches: logarithm, square root, and Min-Max normalization. After evaluation, we found that the square root method was the most effective. Compared to the other two methods, applying the square root reduces extreme disparities while still allowing frequently modified files to retain a greater influence. This makes it a better compromise, as it minimizes the impact of extreme values while preserving meaningful weight differentiation.

To control the number of recalled files, we use a hyperparameter $K_inference$ and set it as a scaling factor. This is because using a fixed value, such as $K_inference = 50$, may lead to inconsistent recall rates across projects of different sizes. In a small project with 100 files, recalling 50 files would cover 50% files of the project, which is excessive. However, in a large project with 10,000 files, recalling 50 files would only account for 0.5%, leading to insufficient recall. When $K_inference$ is set as a scaling factor, such as $K_inference = 0.1$, it can adapt to projects of different sizes while maintaining a consistent recall ratio, the number of files recalled is equal to $(K_inference \times Total\ number\ of\ code\ files)$. In small projects, it prevents recalling an excessive number of files, reducing computational costs, while in large projects, it ensures that a sufficient number of candidate files are recalled to maintain coverage. This approach balances computational efficiency and recall sufficiency, avoiding the biases introduced by using a fixed recall value. In our method, we determine the optimal value of $K_inference$ to be 0.1 based on validation set performance. We further analyze how different $K_inference$ values affect performance on the test set in Section 5.5.

3.4 Code Method Recall

In this step, we count the code files recalled in the first stage and collect all the method-level code snippets they contain to form the candidate set for ranking. Then we fine-tune the pre-trained model such as UniXcoder and GraphCodeBERT using the training dataset in the Review2Code [15] dataset, where each training sample is represented as a $\langle \text{user review}, \text{method snippet} \rangle$ pair. The model learns to optimize the similarity between user reviews and relevant method snippets by minimizing the cross-entropy loss. Finally, each user review and candidate method snippet is then encoded into vector representations using the model fine-tuned in the previous step, and their relevance is determined through similarity computation. Finally, the candidate snippets are ranked based on similarity scores, with the top-ranked snippet selected as the final localization result.

4 Experimental Setups

In this section, we describe the research questions, dataset, baselines, metrics, and other settings.

4.1 Research Questions

First of all, to fully evaluate the proposed ToCoLo approach, we design the following five research questions (RQs) to guide the conduct of experiments:

- (1) RQ1: How well does ToCoLo perform?

- (2) RQ2: How much does the ToCoLo reduce the negative impact of *similar method code snippets*?
- (3) RQ3: How do prompt design and LLM choice affect the performance of ToCoLo?
- (4) RQ4: How effective is weakly supervised training based on retrieval candidates compared to fully supervised training for code localization?
- (5) RQ5: What is the impact of different hyperparameters?

4.2 Dataset

Regarding the dataset, we adopt Review2Code [15], a large-scale ground truth dataset of manually-labeled $\langle \text{User Review}, \text{Code} \rangle$ pairs, introduced for the first time. We chose this dataset for evaluation because this dataset is the most recent proposed for the user review-based code localization task. Review2Code provides detailed statistical data from five open-source mobile applications. The user reviews in the dataset are collected from Google's App Store (i.e., Google Play), while the code is sourced from GitHub. In the dataset, user reviews that appear after the corresponding code commit time are selected to ensure that they do not participate in the model's training process, thereby avoiding potential data leakage. In practice, the model is designed to handle new user reviews to predict relevant code, making the reviews selected during testing more reflective of real-life scenarios [15]. In total, the dataset contains 68,359 $\langle \text{User Review}, \text{Method} \rangle$ pairs, and 33,440 $\langle \text{User Review}, \text{File} \rangle$ pairs collected from five popular Android applications. In total, it includes 1,639 code files, 8,169 methods, and 3,563 commits. The dataset is divided into 70-20-10 proportions for training, validation, and testing in the experiments, following the same settings as study [15].

4.3 Baseline

In this work, we define one-stage methods as approaches that rely solely on a single backbone encoder and a direct similarity function to rank code snippets, without additional modeling components or structural enhancements. Under this definition, GraphCodeBERT [10] represents a state-of-the-art one-stage approach [15] for user review-based code localization. This one-stage approach uses GraphCodeBERT [10] as its backbone. To ensure the completeness of our experiments, we also include a one-stage approach with UniXcoder as the backbone as a baseline. This allows us to compare the advantages of our proposed ToCoLo approach over the one-stage approach when using the same backbone.

We also include YOURCoLo as a baseline for comparison. We treat YOURCoLo as a multi-component, backbone-agnostic framework rather than a one-stage method. Unlike one-stage methods, YOURCoLo is designed to address three key characteristics of real-world review-driven code localization: (1) the availability of project-level contextual information, (2) the one-to-many relationship between a user review and multiple relevant code snippets, and (3) the structural and semantic interconnections among code snippets. Specifically, YOURCoLo incorporates project context into the localization process, adopts a training objective that supports multiple relevant code snippets for a single review, and applies a ranking strategy that leverages relationships among candidate code snippets. Under a fixed backbone, we further compare ToCoLo and YOURCoLo by measuring their respective improvements over the corresponding one-stage baseline, which allows us to assess which structural design yields greater performance gains beyond the one-stage approach.

4.4 Metrics

To evaluate the performance of the models, we employ the Mean Reciprocal Rank (MRR), Top- k , and Similar Mislocalization Rate (SMR) as metrics.

(1) **MRR** is a commonly employed metric in information retrieval (IR) to assess the effectiveness of ranking systems. The calculation of MRR involves taking the reciprocal rank of the first relevant

document retrieved for each query and computing the average of these reciprocal ranks across all queries in the evaluation dataset. Concretely, it is calculated as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{Rank_i} \quad (1)$$

where Q is a query set and $Rank_i$ denotes the rank of the correct code snippet associated with the i -th query.

(2) **Top-K** is the proportion of the reviews whose retrieved top- k code snippets contain at least one in the ground truth:

$$Top - K(UR) = \frac{\sum_{ur \in UR} isCorrect(ur, top - k)}{|UR|} \quad (2)$$

where $|UR|$ is the total number of UR and the $isCorrect(ur, top - k)$ function returns 1 if at least one of top- k code snippets actually is relevant to the review ur and returns 0 otherwise.

(3) **MAP@K** is a widely used metric in Information Retrieval (IR) for evaluating the performance of ranking systems, particularly when multiple relevant documents may exist for each query. Unlike MAP, which considers the entire ranked list, MAP@K restricts the evaluation to the top- K retrieved results, making it more suitable for scenarios where users are only interested in the highest-ranked documents.

The calculation of MAP@K involves computing the Average Precision at K (AP@K) for each query and then taking the mean of these values across all queries in the evaluation dataset. Specifically, it is defined as:

$$MAP@K = \frac{1}{|Q|} \sum_{i=1}^{|Q|} AP@K_i \quad (3)$$

where Q denotes the set of queries, and $AP@K_i$ is the Average Precision at K of the i -th query.

The Average Precision at K , $AP@K_i$, measures the ranking quality by averaging the precision scores at the ranks where relevant documents appear within the top- K results. It can be formulated as:

$$AP@K_i = \frac{1}{\min(|R_i|, K)} \sum_{k=1}^K P_i(k) \cdot rel_i(k) \quad (4)$$

where $|R_i|$ is the total number of relevant documents for query i , K is the cutoff rank, $P_i(k)$ denotes the precision at rank k , and $rel_i(k)$ is an indicator function that equals 1 if the document at rank k is relevant to query i , and 0 otherwise.

(4) **NDCG@K** is a widely used metric in IR for evaluating ranking quality, particularly when the relevance of retrieved documents varies in degree. Unlike metrics that consider only binary relevance, NDCG@K accounts for both the position of relevant documents and their graded relevance, emphasizing higher-ranked items.

The calculation of NDCG@K involves computing the Discounted Cumulative Gain at K (DCG@K) for each query and then normalizing it by the ideal DCG at K (IDCG@K), which represents the maximum achievable DCG@K for that query. Formally, it is defined as:

$$NDCG@K = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{DCG@K_i}{IDCG@K_i} \quad (5)$$

where Q denotes the set of queries, $DCG@K_i$ is the Discounted Cumulative Gain at rank K for the i -th query, and $IDCG@K_i$ is the DCG@K computed on the ideal ranking of documents for that query.

The DCG@K for a single query is calculated as:

$$DCG@K_i = \sum_{k=1}^K \frac{2^{rel_i(k)} - 1}{\log_2(k + 1)} \quad (6)$$

where $rel_i(k)$ denotes the graded relevance of the document at rank k for query i .

The ideal DCG@K ($IDCG@K_i$) is computed by first sorting all documents for the query in descending order of relevance, taking the top- K documents, and then applying the same DCG@K formula. This normalization ensures that $0 \leq NDCG@K \leq 1$, with 1 indicating a perfect ranking of the top- K results.

(5) **SMR** quantifies from the user review perspective the situation where the model is misled by different but similar method code snippets (we refer to such cases as *similar method code snippets*). In the user review-based code localization task, the correct code snippet (Ground Truth) should ideally be ranked higher in the result list. However, if the model incorrectly ranks a *similar method code snippets* above the Ground Truth, the review is considered to have been misled by the similar method code, leading to incorrect localization.

$$SMR = \frac{1}{|UR|} \sum_{ur \in UR} isMisled(ur) \times 100\% \quad (7)$$

where $|UR|$ is the total number of UR and the $isMisled(ur)$ function returns 1 if the model incorrectly ranks a *similar method code snippets* above the Ground Truth. SMR reflects the ability of the model to distinguish between the correct code and similar but incorrect code snippets. A high SMR indicates that the model is easily misled, negatively impacting the accuracy of code localization.

4.5 Other Settings

We chose GraphCodeBERT and UniXcoder as the backbone model of ToCoLo, respectively, with the only difference being the backbone model used. GraphCodeBERT and UniXcoder are both widely used pre-training models that perform well in various downstream tasks, such as code completion, code search, and code generation. They have been widely recognized and applied in both academia and industry. We use the hyperparameter value $K_{inference} = 0.1$, which achieves the best performance on the validation set, as the setting for the inference phase. The maximum sequence lengths of the code snippets and user reviews are 256 and 128. The batch size is 32. The learning rate is set to $2e-5$ and the epoch is set to 30. The GPUs used in the experiment are 2×NVIDIA-3090Tis.

For generating file-level code descriptions, we use the OpenAI GPT-5 (*gpt-5-2025-08-07*) model. To ensure reproducibility, all key generation parameters are explicitly fixed. Specifically, we set *temperature*=0 to minimize randomness and produce deterministic outputs. The *top_p* parameter is set to 1 to disable nucleus sampling, and *max tokens* is set to 512 to allow sufficient output length while preventing truncation. Other parameters, including *presence penalty* and *frequency penalty*, are set to 0 to avoid additional stochastic behavior.

5 Results and Discussion

5.1 RQ1: How well does ToCoLo perform

Method. To comprehensively evaluate the effectiveness of our approach, we conduct a unified comparison among the One-Stage method, ToCoLo, and YourCoLo under a backbone-aligned setting. Specifically, we evaluate the performance of ToCoLo from two complementary perspectives: first, the extent to which ToCoLo improves over the One-Stage baseline; and second, whether the improvement achieved by ToCoLo is stronger than that obtained by YourCoLo when both methods are built upon the same baseline. To ensure a fair comparison, all methods adopt the same backbone encoder (GraphCodeBERT or UniXcoder). The One-Stage method directly matches user reviews with method-level code snippets, whereas ToCoLo employs a two-stage localization pipeline by first retrieving relevant files and then ranking method-level snippets within the retrieved files. YourCoLo enhances method-level localization through structure-aware designs, including project-level context modeling, one-to-many supervision, and relation-aware ranking. For each backbone, we use the corresponding One-Stage method as the common reference and report the relative performance gains of ToCoLo and YourCoLo, enabling a clear and controlled comparison of their respective contributions beyond the baseline. In this research question, ToCoLo is evaluated under a fixed prompt and LLM configuration; a systematic analysis of the impact of prompt design and LLM choice on performance is presented in Section 5.3.

Table 1. Performance comparison of One-Stage, YourCoLo, and ToCoLo methods for method-level and file-level code localization using GraphCodeBERT backbone.

APP Name	Model	Method-level										File-level							
		Top-1	Top-3	Top-5	Top-10	Top-20	MAP@3	MAP@5	MAP@10	NDCG@3	NDCG@5	NDCG@10	MRR	Top-1	Top-3	MAP@3	MAP@5	MRR	
K-9 Mail	One-Stage(GraphCodeBERT)	0.29	0.421	0.486	0.579	0.698	0.083	0.106	0.139	0.237	0.243	0.259	0.369	0.336	0.539	0.145	0.174	0.437	
	YourCoLo(GraphCodeBERT)	0.227	0.342	0.404	0.473	0.554	0.068	0.09	0.12	0.22	0.222	0.233	0.313	0.267	0.442	0.09	0.12	0.377	
	ToCoLo(GraphCodeBERT)	0.312	0.399	0.458	0.533	0.632	0.081	0.101	0.13	0.259	0.256	0.256	0.374	0.265	0.486	0.134	0.146	0.408	
AntennaPod	One-Stage(GraphCodeBERT)	0.183	0.333	0.417	0.483	0.55	0.027	0.037	0.05	0.14	0.141	0.135	0.258	0.317	0.45	0.114	0.131	0.394	
	YourCoLo(GraphCodeBERT)	0.166	0.266	0.333	0.383	0.5	0.033	0.046	0.063	0.127	0.14	0.143	0.237	0.293	0.416	0.072	0.102	0.382	
	ToCoLo(GraphCodeBERT)	0.233	0.367	0.417	0.467	0.533	0.07	0.082	0.102	0.207	0.198	0.199	0.326	0.317	0.717	0.179	0.211	0.516	
Cgeo	One-Stage(GraphCodeBERT)	0.152	0.273	0.333	0.439	0.561	0.053	0.064	0.078	0.144	0.137	0.153	0.224	0.212	0.379	0.118	0.129	0.306	
	YourCoLo(GraphCodeBERT)	0.106	0.257	0.257	0.378	0.469	0.052	0.072	0.084	0.156	0.153	0.158	0.203	0.181	0.378	0.062	0.084	0.303	
	ToCoLo(GraphCodeBERT)	0.212	0.318	0.394	0.455	0.485	0.1	0.113	0.127	0.202	0.196	0.201	0.3	0.333	0.53	0.188	0.201	0.466	
Anki	One-Stage(GraphCodeBERT)	0.189	0.385	0.434	0.525	0.598	0.047	0.062	0.076	0.16	0.159	0.154	0.274	0.295	0.541	0.127	0.151	0.396	
	YourCoLo(GraphCodeBERT)	0.147	0.237	0.27	0.36	0.467	0.04	0.052	0.067	0.14	0.138	0.133	0.219	0.237	0.5	0.087	0.103	0.375	
	ToCoLo(GraphCodeBERT)	0.272	0.352	0.402	0.5	0.623	0.056	0.068	0.089	0.174	0.165	0.175	0.321	0.32	0.639	0.162	0.198	0.505	
Termux	One-Stage(GraphCodeBERT)	0.376	0.477	0.56	0.688	0.807	0.155	0.207	0.282	0.331	0.337	0.364	0.433	0.477	0.697	0.286	0.322	0.559	
	YourCoLo(GraphCodeBERT)	0.238	0.339	0.376	0.467	0.614	0.085	0.125	0.176	0.215	0.218	0.247	0.323	0.348	0.559	0.115	0.198	0.48	
	ToCoLo(GraphCodeBERT)	0.459	0.587	0.642	0.734	0.78	0.187	0.213	0.242	0.374	0.353	0.36	0.545	0.523	0.752	0.342	0.393	0.657	
Avg of 5 apps	One-Stage(GraphCodeBERT)	0.238	0.378	0.446	0.543	0.643	0.073	0.095	0.125	0.202	0.203	0.213	0.312	0.327	0.521	0.158	0.181	0.418	
	YourCoLo(GraphCodeBERT)	0.177	0.288	0.328	0.412	0.521	0.056	0.077	0.102	0.172	0.174	0.183	0.259	0.265	0.459	0.085	0.121	0.383	
	ToCoLo(GraphCodeBERT)	0.298	0.405	0.463	0.538	0.611	0.099	0.115	0.138	0.243	0.234	0.238	0.373	0.352	0.625	0.201	0.230	0.510	

Results. With GraphCodeBERT as the backbone, the result of file-level localization is shown in Table 1, ToCoLo(GraphCodeBERT) performs well compared with the One-Stage(GraphCodeBERT) baseline, achieving an average MRR of 0.51 and an average Top-1 of 0.352 across all five application projects (*K-9 Mail*, *AntennaPod*, *Cgeo*, *Anki*, *Termux*). It significantly outperforms the One-Stage(GraphCodeBERT) method in MAP@K and Top-3/Top-5 metrics. In the *K-9 Mail* project, MRR and MAP@K are slightly lower than the baseline, whereas in the remaining four projects, ToCoLo consistently achieves higher scores, reaching the highest MRR of 0.657 on *Termux*. Compared with YourCoLo(GraphCodeBERT), ToCoLo demonstrates superior performance across most metrics, while YourCoLo shows slightly lower scores, falling below the One-Stage baseline. At the method-level, ToCoLo(GraphCodeBERT) consistently outperforms both One-Stage(GraphCodeBERT) and YourCoLo(GraphCodeBERT) in Top-K, MAP@K, NDCG@K, and MRR metrics. Specifically, ToCoLo achieves higher Top-1, Top-3, Top-5, and MRR

Table 2. Performance comparison of One-Stage, YourCoLo, and ToCoLo methods for method-level and file-level code localization using UnixCoder backbone.

APP Name	Model	Method-level											File-level					
		Top-1	Top-3	Top-5	Top-10	Top-20	MAP@3	MAP@5	MAP@10	NDCG@3	NDCG@5	NDCG@10	MRR	Top-1	Top-3	MAP@3	MAP@5	MRR
K-9 Mail	One-Stage(UniXcoder)	0.296	0.411	0.48	0.545	0.642	0.105	0.13	0.165	0.276	0.284	0.292	0.383	0.352	0.495	0.176	0.204	0.456
	YourCoLo(UniXcoder)	0.364	0.445	0.492	0.651	0.785	0.138	0.178	0.243	0.335	0.345	0.389	0.441	0.417	0.535	0.156	0.189	0.5
	ToCoLo(UniXcoder)	0.374	0.452	0.517	0.586	0.648	0.128	0.154	0.184	0.323	0.317	0.313	0.443	0.265	0.486	0.134	0.146	0.408
AntennaPod	One-Stage(UniXcoder)	0.2	0.283	0.317	0.417	0.467	0.051	0.059	0.08	0.178	0.166	0.178	0.269	0.267	0.4	0.123	0.142	0.36
	YourCoLo(UniXcoder)	0.216	0.3	0.4	0.583	0.683	0.068	0.09	0.126	0.182	0.204	0.242	0.311	0.3	0.416	0.132	0.167	0.403
	ToCoLo(UniXcoder)	0.217	0.35	0.383	0.5	0.6	0.067	0.081	0.089	0.179	0.17	0.172	0.312	0.317	0.717	0.179	0.211	0.516
Cgeo	One-Stage(UniXcoder)	0.136	0.212	0.303	0.379	0.47	0.047	0.053	0.058	0.121	0.114	0.112	0.218	0.167	0.318	0.085	0.098	0.3
	YourCoLo(UniXcoder)	0.197	0.333	0.454	0.56	0.667	0.089	0.108	0.133	0.171	0.192	0.223	0.311	0.272	0.5	0.123	0.167	0.419
	ToCoLo(UniXcoder)	0.227	0.303	0.348	0.409	0.5	0.081	0.091	0.1	0.193	0.186	0.175	0.289	0.333	0.53	0.188	0.201	0.466
Anki	One-Stage(UniXcoder)	0.221	0.352	0.426	0.5	0.607	0.06	0.074	0.095	0.186	0.181	0.18	0.319	0.361	0.533	0.151	0.174	0.472
	YourCoLo(UniXcoder)	0.279	0.368	0.434	0.516	0.622	0.067	0.097	0.139	0.239	0.247	0.254	0.359	0.377	0.565	0.158	0.183	0.491
	ToCoLo(UniXcoder)	0.287	0.402	0.443	0.516	0.549	0.078	0.094	0.115	0.232	0.222	0.21	0.364	0.32	0.639	0.162	0.198	0.505
Termux	One-Stage(UniXcoder)	0.294	0.505	0.569	0.679	0.78	0.166	0.208	0.255	0.343	0.343	0.369	0.431	0.394	0.706	0.281	0.321	0.545
	YourCoLo(UniXcoder)	0.422	0.532	0.651	0.752	0.825	0.205	0.287	0.38	0.41	0.454	0.514	0.52	0.532	0.724	0.285	0.365	0.631
	ToCoLo(UniXcoder)	0.459	0.606	0.688	0.752	0.789	0.185	0.237	0.307	0.4	0.37	0.365	0.555	0.523	0.752	0.342	0.393	0.657
Avg of 5 apps	One-Stage(UniXcoder)	0.229	0.353	0.419	0.504	0.593	0.086	0.105	0.131	0.221	0.218	0.226	0.324	0.308	0.490	0.163	0.188	0.427
	YourCoLo(UniXcoder)	0.296	0.396	0.486	0.612	0.716	0.113	0.152	0.204	0.267	0.288	0.324	0.388	0.380	0.548	0.171	0.214	0.489
	ToCoLo(UniXcoder)	0.313	0.423	0.476	0.553	0.617	0.108	0.131	0.159	0.265	0.253	0.247	0.393	0.352	0.625	0.201	0.230	0.510

scores, as well as strong MAP and NDCG performance, indicating its effectiveness in ranking relevant code snippets at the top and covering a larger portion of relevant code.

With UniXcoder as the backbone, the result of file-level localization is shown in Table 2, ToCoLo(UniXcoder) consistently outperforms One-Stage(UniXcoder) and YourCoLo(UniXcoder) in Top-3, MAP@K, and MRR metrics across most projects. In the *K-9 Mail* project, MRR, Top-3, and MAP@3 are slightly lower than both baselines; for the remaining four projects, ToCoLo achieves notable improvements in MRR, Top-3, and MAP@K metrics. At the method-level, ToCoLo(UniXcoder) achieves the highest MRR on the *Termux* project (0.555) and maintains superior Top-1, Top-3, and MRR scores across projects. While YourCoLo(UniXcoder) outperforms ToCoLo in some MAP@K and NDCG@K metrics, ToCoLo generally provides better ranking of top-relevant code snippets.

Discussion. When comparing ToCoLo with the One-Stage method, across all five app projects, ToCoLo consistently outperforms the One-Stage method in Top-1, MAP@3, NDCG@3, and MRR, demonstrating the effectiveness of the two-stage design. The first stage performs file-level pre-filtering, reducing irrelevant candidates and allowing the second-stage encoder to focus on ranking the most promising snippets. As a result, the model is less distracted by unrelated or similar-but-incorrect code, leading to more accurate and reliable localization. The improvements in Top-K, MAP@K, and NDCG@K have practical significance. Developers typically focus on the top results rather than long ranked lists [4, 45], so higher Top-1 and MRR indicate that the first relevant snippet is more likely to appear at the top, enabling faster identification of the correct code. Increased MAP@K and NDCG@K further indicate that multiple relevant snippets are concentrated among the top results, improving both coverage and quality. Overall, these findings show that ToCoLo helps developers quickly locate the first correct snippet while presenting more relevant code in the top ranks.

After establishing its advantage over the One-Stage baseline, we further compare ToCoLo with YourCoLo to assess its benefits over multi-component approach. ToCoLo demonstrates stable performance across different backbone encoders, with consistent Top-1 and MRR improvements, whereas YourCoLo is more backbone-dependent: its performance decreases with GraphCodeBERT but improves with UniXcoder. The two-stage architecture of ToCoLo, combining file-level pre-filtering and method-level ranking, effectively reduces the candidate space and enhances retrieval quality, as reflected in higher file-level metrics. At the method-level, ToCoLo achieves superior Top-1, Top-K, and MRR scores, allowing developers to find the first correct result faster, while YourCoLo

Table 3. Wilcoxon signed-rank test on the MRR metric between ToCoLo and One-Stage approach.

Method	ToCoLo vs. One-Stage	
Number of Test Samples (N)	678	
Backbone model	GraphCodeBERT	UniXcoder
Normality Test	Shapiro-Wilk	Shapiro-Wilk
Normality Result	Not normal	Not normal
Significance Test	Wilcoxon signed-rank	Wilcoxon signed-rank
Wilcoxon Statistic	69239.5	63131.5
p-value	< 0.0001	< 0.0001

occasionally surpasses ToCoLo in MAP@K and NDCG@K. Overall, these results indicate that ToCoLo not only improves ranking accuracy but also ensures robust and efficient code localization in practical scenarios.

In the experimental results, ToCoLo outperforms the one-stage approach. However, numerical improvement alone does not guarantee that the observed gain is statistically meaningful rather than due to randomness. Therefore, we conducted statistical significance testing to evaluate whether the performance difference between ToCoLo and the one-stage approach is statistically significant. For each of the 678 user reviews in the test set, we computed the MRR scores of both methods, resulting in paired samples for comparison, as shown in Table 3. We first applied the Shapiro–Wilk test to examine whether the distribution of per-review MRR differences (ToCoLo MRR minus one-stage MRR) follows normality. The results indicated that the differences are not normally distributed. Consequently, we employed the Wilcoxon signed-rank test, which does not assume normality and is well-suited for paired samples. When using GraphCodeBERT as the backbone, the Wilcoxon W statistic was 69239.5 with a p-value less than 0.0001, indicating that the improvement is highly significant. Similarly, when using UniXcoder as the backbone, the Wilcoxon W statistic was 63131.5 with a p-value less than 0.0001, also confirming a highly significant improvement. These results consistently demonstrate that ToCoLo achieves statistically significant improvements over the one-stage approach.

Summary: ToCoLo consistently outperforms both the One-Stage and YOURCoLo methods on file-level and method-level code localization, demonstrating its superior ability to rank relevant code higher and present more ground-truth snippets early in the results.

5.2 RQ2: How much do the ToCoLo reduce the negative impact of *similar method code snippets*?

Method. To define what we refer to as similar method code snippets, we first conducted a manual analysis to establish classification criteria. Specifically, we randomly sampled 500 method-level code snippets, selecting 100 from each of the five app projects in the ReviewCode dataset to ensure diversity and coverage. By carefully reading and comparing these methods, we identified two categories. The first category consists of methods that are identical but belong to different files (e.g., method E appearing in both file C and file D), as illustrated by the ground truth code snippet and the similar method code snippet in Figure 1(a). The second category consists of methods that are functionally similar but implemented differently and located in different files (e.g., method A in file C and method B in file D), as shown in Figure 1(b).

Next, we performed a statistical analysis of method-level code snippets in the entire codebase to determine how many similar method code snippets exist. We used the BM25 algorithm to identify

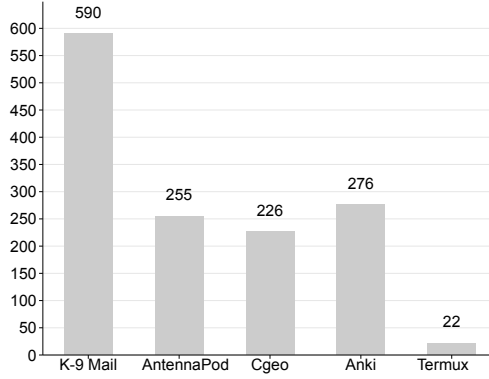


Fig. 4. Number of pairs of *similar method code snippets* in the code base of each App evaluated in this study.

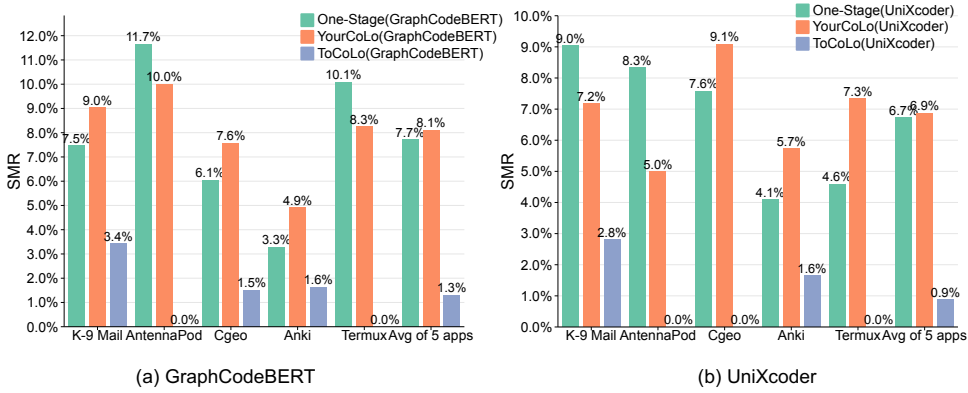


Fig. 5. The ToCoLo reduce the negative impact of the *similar method code snippets*.

similar code snippets by treating each snippet as a query and all method-level snippets from other files as retrieval documents. To reduce false positives, we defined a pair of snippets as similar only if they were each other's most similar counterpart; in other words, snippet A selects snippet B as its most similar match, and snippet B selects snippet A in return. The statistical results are shown in Figure 4. For a given user review, if the ground truth snippet is ranked lower than its similar pair, we consider that the similar method code snippets negatively impacted the code localization for that review.

Finally, we conduct a comparative experiment, recording the SMR metric for the One-Stage method, YourCoLo, and ToCoLo, and analyze to what extent ToCoLo mitigates the impact of *similar method code snippets*.

Results. As shown in Figure 5, compared to the One-Stage and YourCoLo methods using the same backbone, ToCoLo reduces the SMR (Similar Mislocalization Rate) across all projects. For example, with GraphCodeBERT as the backbone, the SMR of *AntennaPod* drops from 11.7% to 0.0%, and

Termux decreases from 10.1% to 0.0%. With UniXcoder as the backbone, the SMR of *AntennaPod*, *Cgeo* and *Termux* all decrease to 0.0%.

Discussion. In the One-Stage method, multiple method snippets that share similar code structure, variable naming, or functionality but come from different files may produce similar vector representations from the encoder, making it difficult to distinguish them during ranking. Consequently, a user review might be matched to a functionally similar but unmapped code snippet, leading to mislocalization.

The ToCoLo method significantly reduces mislocalization caused by *similar method code snippets*, primarily due to its two-stage approach. Compared to the One-Stage method, which directly computes the similarity between user reviews and method-level code snippets, ToCoLo first retrieves relevant file-level code and then performs method-level matching within the retrieved files. This two-stage design effectively minimizes mislocalization caused by *similar method code snippets*, leading to improved ranking metrics at the method-level. Specifically, reductions in SMR translate into higher Top-1, MRR, MAP@K, and NDCG@K scores, demonstrating that mitigating the impact of *similar method code snippets* directly benefits overall localization performance.

Summary: Compared to the One-Stage approach, ToCoLo significantly reduces mislocalization caused by *similar method code snippets*. With GraphCodeBERT as backbone, the average SMR drops from 7.7% to 1.3%, while with UniXcoder, it decreases from 6.7% to 0.9%, thereby directly benefiting localization performance.

5.3 RQ3: How do prompt design and LLM choice affect the performance of ToCoLo?

Method. To better understand the role of code file descriptions in the ToCoLo method, we investigate both prompt design and LLM choice. We generate different file-level descriptions using a combination of three prompt templates and two LLMs. The generation of code file descriptions occurs in a preparatory stage prior to the two-stage process of ToCoLo, using a two-round LLM prompting strategy. In the first round, the LLM analyzes the project's directory structure and produces a global project summary, which serves as stable contextual information for all subsequent file-level analyses. This summary is fixed across all experiments, as it does not directly participate in retrieval and therefore does not affect downstream evaluation. In the second round, we apply the three prompt templates to systematically explore how different types of semantic information and prompt structures influence the quality of LLM-generated file descriptions and their utility for file-level filtering. The three prompt versions are illustrated in Figure 6, with their design rationales summarized as follows:

- **Version 1: Minimal Description** This template asks the LLM to produce only a brief functional summary of the file. It contains no guidance on architectural responsibility or modification scenarios.
- **Version 2: Extended Free-Form Description** This template expands the expected content by asking the LLM to describe the file's functionality, its responsibility within the project, and typical modification scenarios. However, it does not require a structured format, allowing the model to produce a free-form description.
- **Version 3: Structured Three-Field Description** This template enforces a strict structure with three explicit fields: *Function*, *Role*, and *Modification Scenarios*. It provides detailed instructions for each field and applies strict output constraints, forming the highest-guidance version.

Version 1 Prompt You are analyzing a source code file from the app: <code>Anki</code> The following is the file path and the list of methods it contains: File Path: <code>File Path</code> Methods and parameters: <code>method1_name(param1, param2, ...)</code> <code>method2_name(param1, ...)</code> Based on the provided file path and the list of methods with their parameters, summarize the functionality of this file in a description. The entire output must be one single line.	Version 2 Prompt You are analyzing a source code file from the app: <code>Anki</code> The following is the file path and the list of methods it contains: File Path: <code>File Path</code> Methods and parameters: <code>method1_name(param1, param2, ...)</code> <code>method2_name(param1, ...)</code> Based on the provided file path and the list of methods with their parameters, generate a concise description of what this file does, including its main functionality, its responsibility within the project, and the scenarios in which developers may need to modify this file. The entire output must be one single line.
Version 3 Prompt You are analyzing a source code file from the app: <code>Anki</code> The following is the file path and the list of methods it contains: File Path: <code>File Path</code> Methods and parameters: <code>method1_name(param1, param2, ...)</code> <code>method2_name(param1, ...)</code> Based on the provided file path and the list of methods with their parameters, generate a description of this file from three perspectives: Function: Describe the main functionality of this file using the provided methods and their parameters. Focus on what the file actually does, how the methods contribute to its core behavior, and what responsibilities the file carries in terms of computation, data handling, UI interaction, or other logic. Role: Explain the file's responsibility or position within the app's architecture or module structure. Consider how this file interacts with other components, whether it serves as a controller, helper, service, or interface layer, and its importance in the overall app workflow. Use clear and precise language, and avoid general statements. Modification Scenarios: Describe typical situations in which developers would modify this file. Include updates for adding new features, fixing bugs, improving performance, adapting to API changes, handling new input/output requirements, or addressing security concerns. Focus on realistic developer actions directly related to the provided methods. Strict requirements: - Output must follow exactly this format: Function: [description]. Role: [description]. Modification Scenarios: [description]. - The entire output must be one single line. - Do not invent method names, variables, classes, or implementation details not included in the method list. - Do not use quotation marks, bullet points, or line breaks. - Base descriptions only on the provided method names, file path, and project summary.	

Fig. 6. Comparison of prompt versions used for code file description generation.

We then evaluate the impact of LLM choice by introducing another LLM, GPT-4.1² (gpt-4.1-2025-04-14), and comparing it with GPT-5 (gpt-5-2025-08-07). In this experiment, the prompt template, the ToCoLo backbone (fixed to UniXcoder), and all other experimental settings, including the project datasets, are kept constant to ensure a fair and controlled comparison. We also include the One-Stage method with the same UniXcoder backbone as a baseline to quantify how much performance improvement is achieved by ToCoLo when using different LLMs and prompt versions to generate code file descriptions.

Results. The experimental results are as shown in Table 4. First, we compare ToCoLo with the One-Stage method. At the file level, ToCoLo underperforms One-Stage when using Version 1 and Version 2 prompts, while it outperforms One-Stage when using the Version 3 prompt. However, the differences in file-level performance have little impact on method-level localization, as ToCoLo consistently achieves better method-level results than One-Stage, regardless of the prompt or LLM used.

Next, considering the prompt versions used within ToCoLo, the method achieves superior localization performance at both the file and method levels when using the Version 3 prompt, compared to Version 1 and Version 2. Regarding the two LLMs employed in ToCoLo, using the same prompt version results in similar performance, indicating that the choice of LLM for generating code file descriptions has minimal impact on overall performance. Minor fluctuations can be observed in file-level metrics; however, method-level Top-K and MRR metrics remain stable, and MAP@K and NDCG@K at the method-level do not show significant changes.

Discussion. First, although the performance of ToCoLo at the file-level retrieval stage is influenced by prompt design, ToCoLo consistently outperforms the One-Stage method in the final method-level code localization task, regardless of the prompt version or the LLM used. This indicates that file-level recall is not the sole determining factor for method-level localization performance. The

²<https://platform.openai.com/docs/models/gpt-4.1>

Table 4. Impact of prompt design and LLM choice on file-level and method-level code localization performance.

APP Name	Method	Prompt	LLM	Method-level										File-level								
				Top-1	Top-3	Top-5	Top-10	Top-20	MAP@3	MAP@5	5 MAP@10	NDCG@3	NDCG@5	5 NDCG@10	MRR	Top-1	Top-3	MAP@3	MAP@5	5 MRR		
K-9 Mail	One-Stage	N/A	Version1	N/A	0.296	0.411	0.48	0.545	0.642	0.105	0.13	0.165	0.276	0.284	0.292	0.383	0.352	0.495	0.176	0.204	0.456	
				GPT-4.1	0.321	0.408	0.464	0.508	0.567	0.089	0.112	0.14	0.279	0.273	0.265	0.384	0.115	0.277	0.065	0.073	0.23	
				GPT-5	0.315	0.408	0.461	0.526	0.576	0.093	0.113	0.139	0.273	0.262	0.256	0.383	0.128	0.29	0.068	0.078	0.251	
	ToCoLo	Version2	GPT-4.1	0.318	0.414	0.47	0.533	0.601	0.085	0.109	0.14	0.277	0.273	0.269	0.39	0.121	0.296	0.062	0.074	0.247		
			GPT-5	0.327	0.427	0.474	0.551	0.601	0.094	0.117	0.148	0.288	0.28	0.278	0.4	0.118	0.274	0.076	0.088	0.243		
			GPT-4.1	0.324	0.414	0.477	0.545	0.617	0.104	0.131	0.162	0.297	0.29	0.29	0.4	0.259	0.449	0.124	0.139	0.391		
	Version3	GPT-5	0.374	0.452	0.517	0.586	0.648	0.128	0.154	0.184	0.323	0.317	0.313	0.443	0.265	0.486	0.134	0.146	0.408			
		AntennaPod	One-Stage	N/A	N/A	0.2	0.283	0.317	0.417	0.467	0.051	0.059	0.08	0.178	0.166	0.178	0.269	0.267	0.4	0.123	0.142	0.36
					GPT-4.1	0.217	0.3	0.3	0.417	0.5	0.026	0.036	0.052	0.16	0.149	0.145	0.277	0.217	0.367	0.066	0.077	0.346
GPT-5	0.2				0.233	0.3	0.383	0.5	0.025	0.039	0.048	0.14	0.146	0.137	0.253	0.217	0.35	0.074	0.094	0.335		
ToCoLo	Version2		GPT-4.1	0.2	0.217	0.25	0.383	0.45	0.023	0.03	0.047	0.142	0.128	0.137	0.239	0.333	0.45	0.127	0.144	0.431		
			GPT-5	0.217	0.25	0.25	0.417	0.517	0.027	0.034	0.052	0.15	0.127	0.138	0.263	0.3	0.467	0.125	0.146	0.414		
			GPT-4.1	0.233	0.3	0.317	0.433	0.567	0.053	0.064	0.072	0.16	0.156	0.165	0.29	0.35	0.633	0.186	0.203	0.513		
	Version3	GPT-5	0.217	0.35	0.383	0.5	0.6	0.067	0.081	0.089	0.179	0.17	0.172	0.312	0.317	0.717	0.179	0.211	0.516			
		Cgeo	One-Stage	N/A	N/A	0.136	0.212	0.303	0.379	0.47	0.047	0.053	0.058	0.121	0.114	0.112	0.218	0.167	0.318	0.085	0.098	0.3
					GPT-4.1	0.167	0.273	0.303	0.364	0.5	0.071	0.082	0.092	0.163	0.164	0.158	0.244	0.182	0.455	0.096	0.107	0.335
GPT-5	0.152				0.273	0.318	0.379	0.485	0.05	0.062	0.071	0.145	0.151	0.144	0.239	0.212	0.394	0.096	0.111	0.34		
ToCoLo	Version2		GPT-4.1	0.189	0.232	0.322	0.36	0.465	0.07	0.078	0.082	0.151	0.16	0.15	0.262	0.212	0.364	0.089	0.098	0.33		
			GPT-5	0.197	0.242	0.303	0.348	0.47	0.066	0.076	0.086	0.152	0.158	0.149	0.254	0.227	0.409	0.101	0.116	0.347		
			GPT-4.1	0.212	0.298	0.364	0.394	0.53	0.074	0.087	0.095	0.179	0.184	0.169	0.281	0.288	0.47	0.157	0.178	0.414		
	Version3	GPT-5	0.227	0.303	0.348	0.409	0.5	0.081	0.091	0.1	0.193	0.186	0.175	0.289	0.333	0.53	0.188	0.201	0.466			
		Anki	One-Stage	N/A	N/A	0.221	0.352	0.426	0.5	0.607	0.06	0.074	0.095	0.186	0.181	0.18	0.319	0.361	0.533	0.151	0.174	0.472
					GPT-4.1	0.262	0.369	0.418	0.443	0.549	0.068	0.082	0.099	0.213	0.201	0.184	0.336	0.279	0.475	0.143	0.158	0.415
GPT-5	0.246				0.369	0.418	0.467	0.566	0.066	0.082	0.102	0.22	0.209	0.196	0.332	0.27	0.475	0.128	0.149	0.418		
ToCoLo	Version2		GPT-4.1	0.254	0.402	0.434	0.492	0.566	0.069	0.085	0.103	0.224	0.212	0.194	0.343	0.344	0.541	0.132	0.16	0.481		
			GPT-5	0.27	0.385	0.443	0.492	0.566	0.068	0.085	0.105	0.221	0.213	0.198	0.351	0.336	0.615	0.147	0.174	0.487		
			GPT-4.1	0.295	0.41	0.451	0.492	0.579	0.076	0.093	0.115	0.233	0.222	0.207	0.371	0.303	0.656	0.178	0.201	0.496		
	Version3	GPT-5	0.287	0.402	0.443	0.516	0.549	0.078	0.094	0.115	0.232	0.222	0.21	0.364	0.32	0.639	0.162	0.198	0.505			
		Termux	One-Stage	N/A	N/A	0.294	0.505	0.569	0.679	0.78	0.166	0.208	0.255	0.343	0.343	0.369	0.431	0.394	0.706	0.281	0.321	0.545
					GPT-4.1	0.404	0.56	0.587	0.615	0.642	0.148	0.169	0.186	0.342	0.306	0.287	0.483	0.284	0.578	0.211	0.264	0.469
GPT-5	0.385				0.495	0.532	0.587	0.624	0.125	0.144	0.17	0.304	0.27	0.269	0.454	0.257	0.56	0.186	0.232	0.444		
ToCoLo	Version2		GPT-4.1	0.431	0.578	0.642	0.697	0.706	0.165	0.192	0.215	0.367	0.342	0.33	0.519	0.349	0.615	0.216	0.288	0.521		
			GPT-5	0.394	0.55	0.587	0.642	0.661	0.164	0.194	0.223	0.367	0.336	0.324	0.486	0.349	0.578	0.241	0.299	0.497		
			GPT-4.1	0.486	0.652	0.688	0.725	0.78	0.193	0.221	0.248	0.408	0.374	0.362	0.572	0.477	0.688	0.279	0.344	0.616		
	Version3	GPT-5	0.459	0.606	0.688	0.752	0.789	0.185	0.237	0.307	0.4	0.37	0.365	0.555	0.523	0.752	0.342	0.393	0.657			
		Avg of 5 apps	One-Stage	N/A	N/A	0.229	0.353	0.419	0.504	0.593	0.086	0.105	0.131	0.221	0.218	0.226	0.324	0.308	0.490	0.163	0.188	0.427
					GPT-4.1	0.274	0.382	0.414	0.469	0.552	0.080	0.096	0.114	0.231	0.219	0.208	0.345	0.215	0.430	0.116	0.136	0.359
GPT-5	0.260				0.356	0.406	0.468	0.550	0.072	0.088	0.106	0.216	0.208	0.200	0.332	0.217	0.414	0.110	0.133	0.358		
ToCoLo	Version2		GPT-4.1	0.278	0.369	0.424	0.493	0.558	0.082	0.099	0.117	0.232	0.223	0.216	0.351	0.272	0.453	0.125	0.153	0.402		
			GPT-5	0.281	0.371	0.411	0.490	0.563	0.084	0.101	0.123	0.236	0.223	0.217	0.351	0.266	0.469	0.138	0.165	0.398		
			GPT-4.1	0.310	0.415	0.459	0.518	0.615	0.100	0.119	0.138	0.255	0.245	0.239	0.383	0.335	0.579	0.185	0.213	0.486		
	Version3	GPT-5	0.313	0.423	0.476	0.553	0.617	0.108	0.131	0.159	0.265	0.253	0.247	0.393	0.352	0.625	0.201	0.230	0.510			

second-stage ranking module of ToCoLo, which re-ranks candidate methods using an encoder-based similarity model, can effectively mitigate the noise introduced during the file-level retrieval stage. This result validates the necessity and robustness of the two-stage localization design adopted by ToCoLo.

Second, while the two-stage design ensures overall robustness, the quality of LLM-generated file descriptions still plays a critical role in achieving optimal performance. In particular, the Version 3 prompt leads to significantly better results than Version 1 and Version 2. This demonstrates that the quality of LLM-generated code file descriptions is a critical factor affecting ToCoLo's performance. The Version 3 prompt better captures key aspects such as file functionality, architectural role, and modification scenarios, which are more semantically aligned with user reviews, thereby improving overall localization effectiveness. In other words, ToCoLo's design guarantees stable performance, and higher-quality prompts provide additional improvements by supplying richer, more review-aligned file descriptions.

Finally, when using the same prompt, the choice of LLM for generating code file descriptions (GPT-4.1 vs. GPT-5) has a limited impact on the overall performance of ToCoLo. This suggests that ToCoLo does not heavily rely on larger or more recent LLMs; instead, performance gains primarily stem from its two-stage design and well-structured prompt design.

We analyze the differences in code file descriptions generated by the three prompt versions through two concrete examples using the ToCoLo method with GPT-5.

The first example is from the *AntennaPod* project, shown in Figure 7. The user review used for retrieval in this example is: *it's very good. however the widget should be better designed. also the widget sometimes doesn't respond to press (play)*. The review indicates issues related to the

User Review: it's very good. however the widget should be better designed. also the widget sometimes doesn't respond to press (play) Ground Truth: app/src/main/java/de/danoch/antennapod/service/PlayerWidgetService.java core/src/main/java/de/danoch/antennapod/core/service/PlayerWidgetJobService.java app/src/main/java/de/danoch/antennapod/receiver/PlayerWidget.java	File Path: app/src/main/java/de/danoch/antennapod/service/PlayerWidgetService.java Code file descriptions generated using the Version 1 Prompt : The PlayerWidgetService.java file manages the lifecycle and connection of a media player widget service, handling service creation, start commands, service binding and unbinding, updating the widget views, providing playback progress information, and cleaning up resources on destruction. Code file descriptions generated using the Version 2 Prompt : PlayerWidgetService.java manages the lifecycle and UI updates of the podcast playback home screen widget by connecting to the playback service, updating widget views with current playback progress, and handling service binding, making it essential for widget functionality and requiring modification when changing widget behavior, playback display, or service interaction. Code file descriptions generated using the Version 3 Prompt : Function: PlayerWidgetService manages the widget interface for audio playback, providing updates on playback status, progress, and controls directly from the home screen. It ensures that the widget reflects the current state of the audio player and allows users to interact with playback without opening the app. Role: PlayerWidgetService serves as the bridge between the audio playback service and the home screen widget , ensuring that widget updates are timely and accurate, thus enhancing user engagement with the app's playback features. Modification Scenarios: Issues like widget not updating, playback controls failing to respond , incorrect progress display, or crashes when interacting with the widget may require reviewing this file.
Recall Results: Using code file descriptions generated by the Version 1 Prompt : app/src/main/java/de/danoch/antennapod/fragment/SubscriptionFragment.java core/src/main/java/de/danoch/antennapod/core/service/playback/PlaybackService.java core/src/main/java/de/danoch/antennapod/core/service/playback/LocalPSMP.java app/src/main/java/de/danoch/antennapod/dialog/SleepTimerDialog.java core/src/main/java/de/danoch/antennapod/core/service/download/DownloadService.java Using code file descriptions generated by the Version 2 Prompt : app/src/main/java/de/danoch/antennapod/activity/MediaPlayerActivity.java core/src/main/java/de/danoch/antennapod/core/receiver/MediaButtonReceiver.java app/src/main/java/de/danoch/antennapod/receiver/PlayerWidget.java core/src/main/java/de/danoch/antennapod/core/service/PlayerWidgetJobService.java core/src/main/java/de/danoch/antennapod/core/service/playback/PlaybackService.java Using code file descriptions generated by the Version 3 Prompt : core/src/main/java/de/danoch/antennapod/core/service/playback/PlaybackService.java core/src/main/java/de/danoch/antennapod/core/service/PlayerWidgetJobService.java app/src/main/java/de/danoch/antennapod/core/service/PlayerWidgetJobService.java app/src/main/java/de/danoch/antennapod/receiver/PlayerWidget.java core/src/main/java/de/danoch/antennapod/core/service/playback/PlaybackServiceTaskManager.java	

Fig. 7. A case study: When using ToCoLo(GraphCodeBERT), better alignment between the natural language description of the file containing the *ground truth code snippet* and the user review helps filter out *similar method code snippets*, thereby reducing their negative impact on code localization.

widget's design and its responsiveness to playback controls. Key terms include *widget*, *better designed*, *doesn't respond*, *press*, and *play*. All three prompt versions are used to generate code file descriptions for the same file, *PlayerWidgetService.java*. However, only the description generated using the Version 3 prompt enables this file to be successfully retrieved when the review is used as the query. Specifically, the descriptions generated by the Version 1 and Version 2 prompts fail to retrieve this file. The description generated by the Version 1 prompt focuses on service lifecycle management and widget view updates, providing a low-level, implementation-oriented summary that lacks user-facing interaction semantics. Although the description generated by the Version 2 prompt contains the keyword *widget* and offers richer application-level context about widget behavior, it does not mention interaction failures or unresponsive playback controls. Consequently, the lexical overlap between the review and the file description remains insufficient for effective BM25-based retrieval. In contrast, the description generated by the Version 3 prompt successfully leads to the retrieval of *PlayerWidgetService.java*. By structuring the description around function, architectural role, and modification scenarios, the description generated by the Version 3 prompt captures user-facing problems such as widget interaction failures and playback controls that fail to respond. Notably, the resulting description explicitly includes terms such as *widget* and *respond*, which closely align with the keywords appearing in the user review. This increased lexical overlap substantially improves BM25 matching, enabling the file to be successfully retrieved.

The second example is from the *K9mail* project, shown in Figure 8. The user review used for retrieval in this example is: *Slide right to select, slide left to unselect conversations for bulk actions*, which clearly describes a gesture-based interaction for selecting and unselecting conversations to support bulk operations. Key concepts include *sliding gestures*, *select* and *unselect*, *conversations*, *bulk*, and *actions*. All three prompt versions generate code file descriptions for the same file, *MessageViewFragment.java*. However, only the description generated by the Version 3 prompt enables this file to be successfully retrieved when using the review as the query. In contrast, the descriptions generated by the Version 1 and Version 2 prompts fail to retrieve this file. The description produced by the Version 1 prompt primarily focuses on message display logic, fragment

User Review: Slide right to select , slide left to unselect conversations for bulk actions . Ground Truth: app/ui/legacy/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java	File Path: app/ui/legacy/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java
Recall Results: Using code file descriptions generated by the Version 1 Prompt : k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java app/core/src/main/java/com/fsck/k9/controller/MessagingController.java k9mail/src/main/java/com/fsck/k9/controller/MessagingController.java k9mail/src/main/java/com/fsck/k9/activity/FolderList.java k9mail/src/main/java/com/fsck/k9/fragment/MessageListFragment.java Using code file descriptions generated by the Version 2 Prompt : k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java k9mail/src/main/java/com/fsck/k9/ui/crypto/MessageCryptoHelper.java app/ui/src/main/java/com/fsck/k9/fragment/MessageListFragment.java k9mail/src/main/java/com/fsck/k9/fragment/MessageListFragment.java src/com/fsck/k9/activity/setup/Prefs.java Using code file descriptions generated by the Version 3 Prompt : app/ui/legacy/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java app/core/src/main/java/com/fsck/k9/controller/MessagingController.java k9mail/src/main/java/com/fsck/k9/fragment/MessageListFragment.java src/com/fsck/k9/activity/MessageList.java app/ui/src/main/java/com/fsck/k9/activity/MessageList.java	Code file descriptions generated using the Version 1 Prompt : The MessageViewFragment.java file in the app module's UI legacy package manages the display and interaction of email message views, handling header toggling, message loading states, read status checks, refile actions, and alternate sending options within the fragment lifecycle. Code file descriptions generated using the Version 2 Prompt : MessageViewFragment.java manages the UI and logic for displaying email message content within the apps legacy message view, handling header toggling, message loading states, read status, and user actions like refile or sending alternate messages, and developers modify it when updating message display behavior, header visibility, or message interaction workflows. Code file descriptions generated using the Version 3 Prompt : Function: MessageViewFragment manages the selection and action interface for email conversations , enabling users to easily select or unselect messages for bulk actions . It facilitates user interactions by allowing sliding gestures to toggle message selection. Role: MessageViewFragment serves as the UI component that displays email messages and supports bulk operations, ensuring users can effectively manage conversations with simple gestures. Modification Scenarios: Issues such as inability to select messages, incorrect gesture recognition, or failure to execute bulk actions may necessitate examining and modifying this file.

Fig. 8. A case study: When using ToCoLo(GraphCodeBERT), better alignment between the natural language description of the file containing the *ground truth code snippet* and the user review helps filter out *similar method code snippets*, thereby reducing their negative impact on code localization.

lifecycle management, and auxiliary actions such as header toggling and message refile, reflecting a reading-oriented view of email messages rather than selection-oriented interactions. Although the description generated by the Version 2 prompt provides a richer, application-level summary and mentions generic message interaction workflows, it still lacks explicit references to gesture-based selection, conversation selection states, or bulk actions. As a result, the lexical overlap between the review and the file descriptions generated by the description generated by the Version 1 and Version 2 remains minimal, limiting their effectiveness under BM25-based retrieval. In contrast, the description generated by the Version 3 prompt explicitly incorporates key terms that overlap with the user review, including *select* and *unselect*, *conversations*, and *bulk actions*. By framing the file's functionality around gesture-based selection and bulk operations on conversations, the description generated by the Version 3 prompt achieves a substantially higher degree of lexical overlap with the review. In addition, its modification scenarios further reinforce this alignment by explicitly mentioning selection- and gesture-related failures. This combination of explicit keyword matching and interaction-oriented description significantly improves BM25-based retrieval, allowing *MessageViewFragment.java* to be successfully retrieved.

The two case studies consistently show that descriptions generated by Version 3 Prompt are more specific, detailed, and informative than those produced by Version 1 Prompt and Version 2 Prompt, leading to more reliable retrieval of ground-truth files using BM25. Version 3 Prompt explicitly structures descriptions around function, architectural role, and modification scenarios. This richer and more structured representation better captures user-facing issues and incorporates review-aligned keywords (e.g., *respond*, *select*, *unselect*, *bulk*, *actions*), substantially increasing lexical overlap with reviews and improving retrieval performance.

Summary: The performance of ToCoLo is influenced by prompt design, where structuring file descriptions around function, architectural role, and modification scenarios better captures review-aligned semantics and enables reliable retrieval of ground-truth files, whereas the choice of LLM has only a minor impact.

5.4 RQ4: How effective is weakly supervised training based on retrieval candidates compared to fully supervised training for code localization?

Method. When fine-tuning the encoder in our second-stage model, we initially train the method-level code localization model using the full training set. However, this setup does not match the data distribution encountered during inference, where file-level retrieval is first performed, and method-level ranking is applied only within the retrieved files. To better align the training distribution with that of inference, we explore a new data construction strategy that simulates the inference pipeline during training. Specifically, for each user review, we retrieve a fixed number of candidate files, extract all method snippets from these files, and construct ⟨user review, code snippet⟩ pairs from this constrained candidate space. This design ensures that the model learns to make predictions within the same limited context it will encounter at inference time. However, since some ground-truth method snippets may not be included in the recalled files, the resulting training set is only partially supervised. We refer to this setup as weak supervision, where positive and negative samples are constructed based on retrieval results guided by pre-trained models. This enables us to investigate whether training under a more inference-aligned supervision signal can improve code localization performance compared to conventional full-data training.

To align the training process with inference phase file recall behavior, we propose a method to construct weakly supervised training samples that reflect the actual retrieval distribution used at inference. We use the file retrieval strategy employed during the first stage to retrieve ($K_{inference} \times \text{Total number of code files}$) candidate files for each user review in the training set.

Next, we construct the ⟨user review, code snippet⟩ pairs used for fine-tuning. By analyzing the original training set, we observe that each user review is on average associated with 12.29 matched code snippets. For each user review, we first collect all method snippets from its corresponding retrieved files as candidate methods. Then, we rank these candidate methods using two state-of-the-art pretrained models from the code search task: UniXcoder[9] and CoCoSoDa[47]. We take the intersection of the top-12 ranked methods from both models. If the intersection contains fewer than 12 methods, we supplement it with the highest-ranked methods from both models that are not in the intersection, until we collect 12 method snippets in total.

These method snippets serve as the basis for constructing training samples. In cases where ground-truth method snippets from the original training set are not present in the selected 12 candidates, we iteratively replace candidate methods starting from the end of the list with ground-truth snippets until all ground-truth methods have been incorporated. Finally, we treat ⟨user review, ground-truth method snippet⟩ pairs as positive samples, and ⟨user review, non-ground-truth method snippet⟩ pairs as negative samples.

To investigate the impact of the number of candidate methods (i.e., candidate set size) associated with each user review in the training set on model performance, we design a comparative study based on the observation that each user review in the original training set is matched with 12 code snippets on average. We introduce a hyperparameter K_{train} , and construct different versions of the training set by setting the candidate size to $K_{train} \in (9, 10, 11, 12, 13, 14, 15)$. The number of positive and negative samples in the constructed different training sets is shown in Table 5. Using the same backbone model and training strategy, we take GraphCodeBERT and UniXcoder as the backbone, respectively, and the loss function used in the fine-tuning process of the model is Binary Cross Entropy. We fine-tune the model on each variant and compare its performance on the test dataset to analyze how the candidate set size affects the final code localization capability.

Results. In the experiments using GraphCodeBERT as the backbone, as K_{train} increases from 9 to 15, the overall MRR changes slightly, but the variation is relatively small (Figure 9(a)). In the

Table 5. The number of training dataset samples constructed using the file recall strategy employed in the inference stage

App Name	Samples Count	K_train							
		9	10	11	12	13	14	15	
K-9 Mail	Positive Samples	21,803	21,567	21,269	21,033	20,645	20,251	20,023	
	Negative Samples	9,660	11,084	12,519	14,005	15,517	17,042	18,545	
AntennaPod	Positive Samples	6,112	6,077	5,691	5,411	5,356	5,334	5,169	
	Negative Samples	1,720	1,983	2,259	2,525	2,796	3,086	3,370	
Cgeo	Positive Samples	4,606	4,468	4,461	4,432	4,380	4,346	4,273	
	Negative Samples	2,125	2,421	2,736	3,038	3,336	3,637	3,965	
Anki	Positive Samples	9,949	9,917	9,856	9,798	9,793	9,718	9,652	
	Negative Samples	3,687	4,262	4,833	5,442	6,060	6,685	7,287	
Termux	Positive samples	2,645	2,550	2,490	2,399	2,291	2,276	2,221	
	Negative samples	2,597	2,866	3,160	3,422	3,641	3,851	4,048	

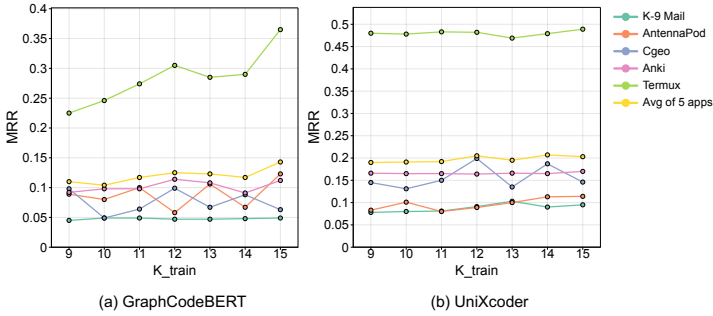


Fig. 9. The effect of K_{train} value on the result of method-level code localization

experiments using UniXcoder as the backbone, as K_{train} increases from 9 to 15, the MRR remains almost stable with even smaller changes (Figure 9(b)). However, regardless of the value of K_{train} , the performance of the fine-tuned models (in terms of MRR) is consistently lower than that of the models fine-tuned on the original full training dataset. Adjusting the hyperparameter K_{train} has little effect on improving the model performance.

Discussion. Experimental results show that although constructing the candidate set based on the retrieval stage aligns better with the inference-time data distribution, the overall model performance is slightly lower compared to training with the full candidate set. This suggests that while distributional alignment between training and inference is important, maintaining a stronger supervisory signal during training may be more critical when high-quality annotated data is available.

One contributing factor to the performance drop is the decline in the quality of training data in the retrieval-based setting. Not all code snippets relevant to the user reviews were successfully retrieved, and some positive samples present in the original training set were missing in the newly constructed version. The retrieval process relied on pre-trained models such as UniXcoder and CoCoSoDa, which introduce inherent biases and lead to a distribution shift between the new training data and the actual relevance distribution.

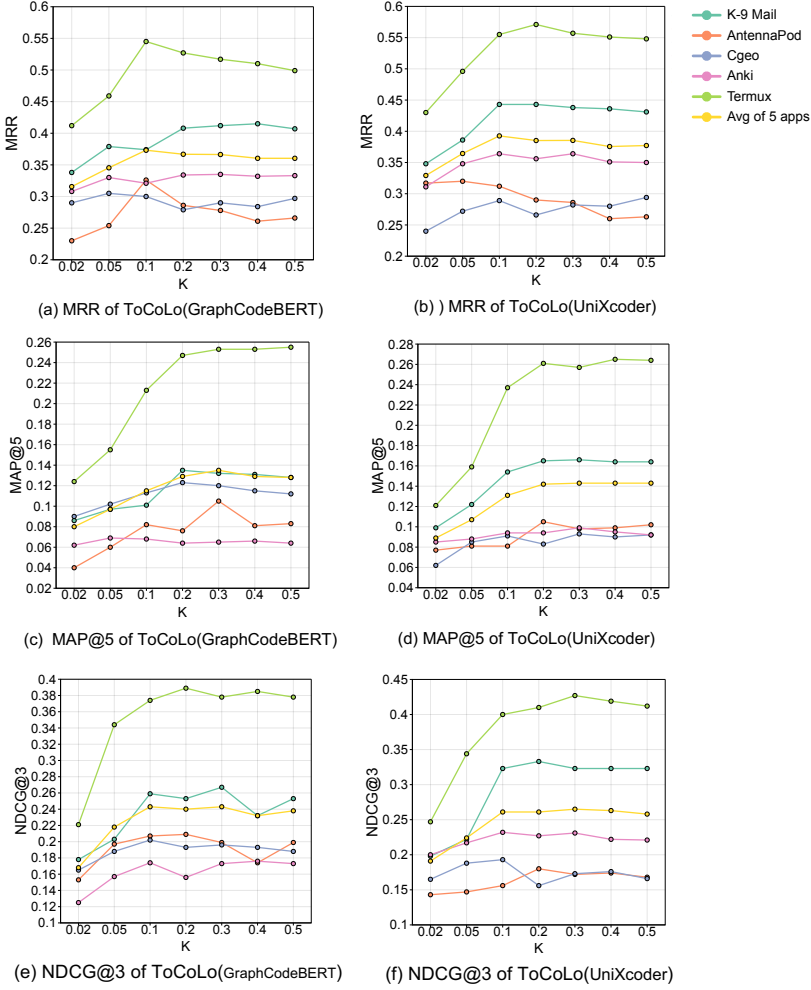


Fig. 10. The effect of $K_{inference}$ value on the result of method-level code localization.

Furthermore, varying the candidate set size K_{train} yields only marginal performance changes. Since the underlying issue of data quality remains unresolved, simply increasing or decreasing K_{train} does not substantially affect model performance.

Summary: The performance gap suggests that aligning training with inference phase distributions is insufficient when training data quality is compromised.

5.5 RQ5: What Is the Impact of Different Hyperparameters?

Method. In this section, we study the impact of the recall proportion on the code localization performance of the ToCoLo method. One parameter is involved: $K_{inference}$. For file-level code localization, BM25 is used to retrieve code files from the repository based on user reviews, without involving model fine-tuning. In the inference phase, file retrieval is performed based on a proportion of the code repository. Changing the file retrieval proportion does not affect the final ranking

of file-level code localization results. As a result, adjusting the inference phase hyperparameter $K_inference$ does not influence file-level code localization performance.

Therefore, we focus on the impact of $K_inference$ on method-level code localization. We conduct experiments with $K_inference$ set to 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, and 0.5 to observe how the number of recalled files affects performance. Evaluation is performed using MRR, MAP@5 and NDCG@3. We use MAP@5 as a representative of MAP@K because it reflects the ranking quality within a reasonably small candidate set, which aligns with practical scenarios where developers typically inspect only the top-ranked results. Similarly, we select NDCG@3 as a representative of NDCG@K to capture the quality of ranking at the very top of the list.

Results. As shown in Figure 10, the MRR of both backbones increases noticeably as $K_inference$ grows from 0.02 to around 0.1–0.2. Beyond 0.2, the MRR curves flatten out and, in some cases, show a slight decline (e.g., *AntennaPod*). ToCoLo_(UniXcoder) follows a similar trend, with MRR increasing up to 0.1–0.2 and stabilizing thereafter. In contrast, the MAP@5 and NDCG@3 results steadily increase for both backbones as larger candidate sets are recalled, eventually reaching their maximum when $K_inference$ is around 0.1–0.2. These results indicate that moderate values of $K_inference$ effectively balance retrieval depth and ranking accuracy, whereas excessively large values provide limited additional benefit.

Discussion. For MRR, moderate increases in $K_inference$ (0.1–0.2) are beneficial as they balance the trade-off between retrieving enough candidates and avoiding excessive noise. Larger values introduce more irrelevant files, which increases computational cost and can slightly harm ranking precision. In contrast, MAP@5 benefits more directly from larger candidate sets, since recalling more files naturally increases the chance that relevant methods appear within the top-5 results; NDCG@3 exhibits a similar tendency. However, the improvements taper off once the recall set becomes sufficiently large, leading to a saturation effect. From an efficiency perspective, larger $K_inference$ values require significantly more similarity computations at the method-level. Therefore, the choice of hyperparameter should aim to balance effectiveness (accuracy gains in MRR, MAP@5 and NDCG@3) with efficiency (computational overhead). Empirically, setting $K_inference$ within the range of 0.1–0.2 achieves near-optimal accuracy while keeping the computational cost manageable. The consistently superior performance of UniXcoder further suggests that stronger backbones mitigate the sensitivity to hyperparameter choices.

Summary: Setting $K_inference$ around 0.1–0.2 balances accuracy and efficiency. MRR benefits from moderate candidate retrieval, while MAP@K and NDCG@K improve until saturation. Larger values increase computation with limited gains.

5.6 Case Study and Error Analysis

As shown in Figure 11, the *ground truth method snippet* is located in the file `main/src/cgeo/geocaching/maps/CGeoMap.java`, while the *similar method code snippet* comes from `main/src/cgeo/geocaching/maps/mapsforge/v6/caches/CachesOverlay.java`. Although they belong to different files, their functionalities are quite similar, both are primarily designed to trigger data or display updates when the `MapView` changes, ensuring that users are presented with the most up-to-date cache information or map content. Using the one-stage approach, we can see that the *ground truth method snippet* is ranked 17th, whereas the *similar method code snippet* is ranked 10th. This indicates that without incorporating file-level information, the code localization model may struggle to make the correct judgment. By using our proposed ToCoLo method, in the first stage, by incorporating the file description information, we can see that the *Function* section in the description of `CGeoMap.java`

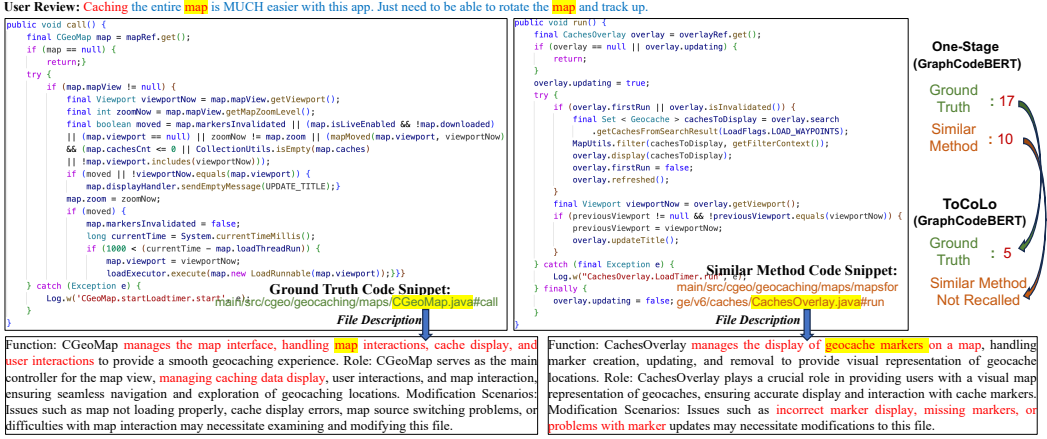


Fig. 11. A case study: When using ToCoLo(GraphCodeBERT), better alignment between the natural language description of the file containing the *ground truth code snippet* and the user review helps filter out *similar method code snippets*, thereby reducing their negative impact on code localization.

aligns well with the user review, containing more keywords relevant to the user review. Additionally, the *Role* section also includes statements that match the user review, which allows the file to be successfully retrieved. In contrast, the *Function* and *Modification Scenarios* sections in the description of *CachesOverlay.java* focus more on the management and display of cache markers, and contain fewer keywords relevant to the user review. Therefore, it scored lower during the recall stage and was not recalled. As a result, the *similar method code snippet* is excluded from the similarity computation with the user review in the second stage. This avoids the influence of the *similar method code snippet* on the ranking of the *ground truth method snippet*. We can observe that the *ground truth method snippet*'s rank improves to 5th.

As shown in Figure 12, the *ground truth method snippet* is located in the file *app/src/main/java/de/danoeh/antennapod/fragment/QueueFragment.java*, while the *similar method code snippet* comes from *app/src/main/java/de/danoeh/antennapod/fragment/NewEpisodesFragment.java*. Although they are from different files, the method names and parameters are identical. These two code snippets exhibit a high degree of similarity in terms of structure and coding pattern: both begin by checking whether the superclass has handled the event and use a *switch* statement to execute specific operations based on the menu item ID. Functionally, they both involve UI interactions and data operations, such as refreshing content, updating display elements, showing confirmation dialogs, or modifying state. Using the one-stage approach, we observe that the *ground truth method snippet* is ranked 369th, while the *similar method code snippet* is ranked 25th. This indicates that the code localization model struggles to make accurate judgments without incorporating file-level information. In the first stage, by incorporating file description information, we can observe that the *Function*, *Role*, and *Modification Scenarios* sections in the file description of *QueueFragment.java* contain more keywords relevant to the user review, allowing this file to be successfully recalled. In contrast, the *Function* and *Modification Scenarios* in the file description of *NewEpisodesFragment.java* focus more on the display and management of new podcast episodes and provide fewer descriptions related to navigation functionality. Due to the lack of relevant keywords, it received a lower score during the recall stage and was not retrieved. As a result, the *similar method code snippet* is excluded from the similarity computation with the user review in the second stage, thereby eliminating its

User Review: Love it, although the UI navigation could be improved a bit.

```
public boolean onOptionsItemSelected(MenuItem item) {
    if (isUser.onOptionsItemSelected(item)) {
        switch (item.getItemId()) {
            case R.id.queue_lock:
                toggleQueueLock();
                return true;
            case R.id.refresh_item:
                AutoDataManager.runImmediate(requireContext());return true;
            case R.id.clear_queue:
                ConfirmationDialog conDialog = new ConfirmationDialog(getActivity(),
                    R.string.clear_queue_label, R.string.clear_queue_confirmation_msg) {
                    public void onConfirmPressed(DialogInterface dialog) {
                        dialog.dismiss();
                        QDWriter.clearQueue();
                        conDialog.createNewDialog().show();return true;
                    }
                };
                (MainActivity).requireActivity().loadChiliFragment(EpisodesApplyActionFragment.newInstance(queue, ACTION_DELETE | ACTION_REMOVE_FROM_QUEUE | ACTION_DOWNLOAD));return true;
            case R.id.queue_sort_episode_title_asc:
                setSortOrder(SortOrder.EPISODE_TITLE_A_Z);return true;
            case R.id.queue_sort_episode_title_desc:
                setSortOrder(SortOrder.EPISODE_TITLE_Z_A);return true;
            case R.id.queue_sort_date_asc:
                app/src/main/java/de/danoch/antennapod/fragme
                setSortOrder(SortOrder.DATE_OLD_NEW);return true;
            case R.id.queue_sort_date_desc:
                app/src/main/java/de/danoch/antennapod/fragme
                setSortOrder(SortOrder.DATE_NEW_OLD);return true;
            case R.id.queue_sort_duration_asc:
                setSortOrder(SortOrder.DURATION);return true;
        }
    }
}
```

Ground Truth Code Snippet:

app/src/main/java/de/danoch/antennapod/fragme

QueueFragment.java#onOptionsItemSelected

File Description

```
public boolean onOptionsItemSelected(MenuItem item) {
    if (isUser.onOptionsItemSelected(item)) {
        switch (item.getItemId()) {
            case R.id.refresh_item:
                List < Feed > feeds = ((MainActivity) getActivity()).getFeeds();
                if (feeds != null) {
                    DBTasks.refreshAllFeeds(getActivity(), feeds);
                }
                return true;
            case R.id.mark_all_read_item:
                DBWriter.markAllItemsRead(getActivity());
                Toast.makeText(getActivity(), R.string.mark_all_read_msg,
                    Toast.LENGTH_SHORT).show();
                return true;
            case R.id.episode_filter_item:
                boolean newWal = item.isChecked();
                setShowNewEpisodes(newWal);
                item.setChecked(newWal);
                return true;
            default:
                return false;
        }
    }
    app/src/main/java/de/danoch/antennapod/fragme/
    NewEpisodesFragment.java#onOptionsItemSelected
    return true;
}
```

Similar Method Code Snippet:

app/src/main/java/de/danoch/antennapod/fragme/

NewEpisodesFragment.java#onOptionsItemSelected

File Description

One-Stage
(UniXcoder)

Ground
Truth : 369

Similar
Method : 25

ToCoLo
(UniXcoder)

Ground
Truth : 20

Similar Method
Not Recalled

Function: QueueFragment oversees the queue of podcast episodes, handling item download status, context menu interactions, navigation between items, view creation, toolbar and info bar updates, sorting, queue locking, item loading, and menu creation. Role: QueueFragment plays a crucial role in managing the queue of podcast episodes, ensuring smooth UI navigation and user interactions, including navigating between different podcast episodes, downloading progress tracking, context menu handling, UI updates, and sorting functionalities to enhance the user experience. Modification Scenarios: QueueFragment may need modifications in scenarios involving issues with download progress tracking, context menu functionality, navigation problems, UI updates, sorting options, queue locking, and item loading to address potential problems and ensure smooth operation

Function: NewEpisodesFragment displays and manages new episodes, handling UI updates, data loading, and event handling. Role: NewEpisodesFragment acts as a user interface component, showing new episodes, managing data loading, and responding to user interactions to provide a seamless experience. Modification Scenarios: Issues like new episodes not displaying correctly, data loading errors, or UI not updating properly may require reviewing and modifying this file.

Fig. 12. A case study: When using ToCoLo(UniXcoder), better alignment between the natural language description of the file containing the *ground truth code snippet* and the user review helps filter out *similar method code snippets*, thereby reducing their negative impact on code localization.

User Review: There is a BUG in which my cards got flipped. On the front side there were the answers, that i wrote on the back Ground Truth: AnkiDroid/src/main/java/com/ichi2/anki/cardviewer/TypedAnswer.java#cleanCorrectAnswer src/com/ichi2/anki/AbstractFlashcardViewer.java#onClick	
Method Location Results: 1. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#cleanCorrectAnswer 2. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#updateForNewCard 3. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#initControls 4. AnkiDroid/src/main/java/com/ichi2/anki/Previewer.java#onClick 5. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#typeAnsQuestionFilter 6. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#typeAnsAnswerFilter 7. AnkiDroid/src/main/java/com/ichi2/anki/Previewer.java#displayCardQuestion 8. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#typeAnswer 9. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java#restorePreferences 10. AnkiDroid/src/main/java/com/ichi2/anki/Preferences.java#removeUnnecessaryAdvancedPrefs 	Ground Truth Method Not in the Result List
File Recall Results: 1. AnkiDroid/src/main/java/com/ichi2/anki/AbstractFlashcardViewer.java 2. AnkiDroid/src/main/java/com/ichi2/anki/CardBrowser.java 3. AnkiDroid/src/main/java/com/ichi2/anki/DeckPicker.java 4. AnkiDroid/src/main/java/com/ichi2/anki/CardTemplateEditor.java 5. src/main/java/com/ichi2/anki/Reviewer.java 	Ground Truth File Not Recalled
67. AnkiDroid/src/main/java/com/ichi2/anki/cardviewer/TypedAnswer.java 87. src/com/ichi2/anki/AbstractFlashcardViewer.java	Ground Truth File

Fig. 13. A case study: When using ToCoLo(UniXcoder), method-level localization fails because the ground-truth file was not recalled.

influence on the ranking of the *ground truth method snippet*. We can observe an improvement in the ranking of the *ground truth method snippet*, which rises to 20th.

As shown in Figure 13, we present a challenging example where ToCoLo does not successfully localize the ground-truth method, which helps reveal an inherent limitation of the two-stage localization framework. In this example, the ground-truth method is not retrieved at the method level. A closer inspection indicates that the underlying cause originates from the file-level recall

stage: the ground-truth file is ranked 67th and 87th, respectively, and thus falls outside the file-level recall scope. As a result, methods within the ground-truth file are excluded from the subsequent ranking process. This suggests that the file-level descriptions generated for these ground-truth files are not sufficiently aligned with the semantics of the user review, preventing them from being recalled. Consequently, although the method-level ranking stage of ToCoLo can effectively distinguish relevant methods when adequate candidates are available, its overall performance remains constrained by the recall quality of the file-level recall stage. This observation motivates future work on enhancing file-level representations or developing hybrid retrieval strategies that jointly leverage file-level and method-level signals to further improve robustness.

6 Threats To Validity

Construct Validity. When using LLMs to generate code file descriptions, the generated summaries may suffer from several limitations: they can be overly abstract or generic, omit important implementation details, or fail to fully reflect realistic modification scenarios. Such issues may lead to an inaccurate modeling of file-level semantics, thereby affecting the reliability of file-level recall. To mitigate this threat, we adopt a structured prompt template that explicitly constrains the generation process along three predefined dimensions: function, role, and modification scenarios. This design reduces stylistic variability and guides the model to focus on the file’s function, architectural role, and typical modification scenarios, rather than low-level implementation details. Moreover, our analysis of multiple prompt versions (Section 5.3) reveals consistent performance trends, indicating that the structured representation used in our evaluation is both stable and meaningful. Nevertheless, while the structured prompt mitigates many of these issues, the quality of file-level recall remains inherently dependent on the fidelity of the generated summaries. As discussed in Section 5.6, which presents a case study of ToCoLo localization failures, mismatches between file-level descriptions and user reviews can still prevent ground-truth files from being recalled, thereby limiting the effectiveness of subsequent method-level ranking. To address this limitation, future work may explore richer file-level representations or hybrid retrieval strategies that jointly leverage file-level and method-level signals, thereby reducing the reliance on a single form of generated summary.

Internal Validity. A potential threat to internal validity is that the LLM used in our approach might have been pretrained on publicly available code repositories that overlap with our dataset, which could in principle expose review-to-code associations. To mitigate this threat, we restrict the LLM’s role to generating natural language descriptions of code files using only the source code as input; no user reviews are provided. Consequently, the LLM cannot directly encode or reveal any review-to-code mappings from the dataset. Furthermore, these generated summaries are only used in the BM25-based file recall stage, which relies purely on textual similarity and does not access any internal knowledge of the LLM. Method-level ranking is then performed by a small model that has never seen the dataset, ensuring that the final localization step is independent of any potential prior LLM exposure. Therefore, the only possible effect of LLM pretraining is an indirect benefit: more informative file descriptions may slightly improve BM25 recall, but this does not constitute data leakage, as neither BM25 retrieval nor method-level ranking uses the review-to-code ground truth. In this way, although the LLM may have prior exposure to external code repositories, our design ensures that no confidential review-to-code associations are leaked, mitigating the threat to internal validity.

Moreover, our conclusions are drawn based on multiple metrics (MRR, Top-K, MAP@K, NDCG@K) across diverse projects and prompt configurations, ensuring that our findings are statistically robust and not dependent on a single evaluation scenario.

External Validity. Although our proposed ToCoLo approach achieves good results on multiple projects, its generalizability to other environments remains uncertain due to dataset limitations. Specifically, it has only been evaluated on a dataset comprising Java projects, and its performance on projects written in other programming languages or in different domains has not been assessed. To address this limitation, we plan to extend our evaluation to datasets from a broader range of programming languages and application contexts in future work. This will help further validate the effectiveness of our approach and enhance its applicability across diverse environments.

7 Conclusion and Future Work

In this paper, we propose a two-stage approach named ToCoLo to address one key challenge in user review-based code localization. To reduce the negative impact of *similar method code snippets*, we use an LLM to generate relevant descriptions for each code file, and then apply file recall in the first stage of ToCoLo to eliminate interference between similar method code snippets. The experimental evaluation demonstrates that ToCoLo consistently improves user review-based code localization across different backbone models. With UNIXCODER as the backbone, ToCoLo achieves absolute improvements of 8.3% in Top-1 accuracy, 7.0% in Top-3, 4.5% in NDCG@3, and 6.9% in Mean Reciprocal Rank (MRR) compared to the one-stage baseline. Using GRAPHCODEBERT, ToCoLo attains a Top-1 gain of 6.0%, along with 4.1% in NDCG@3 and 6.2% in MRR. These results indicate that ToCoLo is more effective at ranking truly relevant code snippets at the very top of the localization results. The impact of similar method code snippets on the SMR evaluation metric was reduced to 0.9%. Finally, we conduct some case studies to demonstrate the superior performance of ToCoLo.

In future studies, we will explore integrating more contextual information, such as function call graphs, into file recall and propose an improved user review-based code localization method to further enhance localization accuracy. The replication package is available at the following link: <https://github.com/Ch1Ku0/tocolo>.

Acknowledgments

This research / project is supported by the National Natural Science Foundation of China (62572235), CCF-Huawei Populus Grove Fund, and Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming.

References

- [1] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. 2018. Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 100–105.
- [2] Usman Ashraf, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Mining cross-task artifact dependencies from developer interactions. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 186–196.
- [3] Aakash Bansal, Chia-Yi Su, and Collin McMillan. 2023. Revisiting file context for source code summarization. *arXiv preprint arXiv:2309.02326* (2023).
- [4] Chris Barry and Mark Lardner. 2011. A study of first click behaviour and user interaction on the Google SERP. In *Information Systems Development: business systems and services: modeling and development*. Springer, 89–99.
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [6] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C Gall. 2017. Analyzing reviews and code of mobile apps for better release planning. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 91–102.
- [7] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. 2013. Development and deployment at facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

- [8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
- [10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
- [11] Marlo Haering, Christoph Stanik, and Walid Maalej. 2021. Automatically matching bug reports with related app reviews. In *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE, 970–981.
- [12] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.
- [13] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2012. App store mining and analysis: MSR for app stores. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 108–111.
- [14] Mark Harman and Peter O’Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 1–23.
- [15] Xinwen Hu, Yu Guo, Jianjie Lu, Zheling Zhu, Chuanyi Li, Jidong Ge, Liguang Huang, and Bin Luo. 2022. Lighting up supervised learning in user review-based code localization: dataset and benchmark. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–545.
- [16] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2020. Control flow graph embedding based on multi-instance decomposition for bug localization. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 4223–4230.
- [17] Claudia Iacob and Rachel Harrison. 2013. Retrieving and analyzing mobile apps feature requests from online reviews. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 41–44.
- [18] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *EMNLP (1)*. 6769–6781.
- [19] Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–6.
- [20] Sayed Hassan Khatonabadi, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. 2023. On wasted contributions: Understanding the dynamics of contributor-abandoned pull requests—a mixed-methods study of 10 large open-source projects. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–39.
- [21] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.
- [22] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.
- [23] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 579–590.
- [24] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A survey of DevOps concepts and challenges. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–35.
- [25] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.
- [26] Zhengliang Li, Zhiwei Jiang, Xiang Chen, Kaibo Cao, and Qing Gu. 2021. Laprob: a label propagation-based software bug localization method. *Information and Software Technology* 130 (2021), 106410.
- [27] Mario Linares-Vásquez, Andrew Holtzauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 242–251.
- [28] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. 2024. Proconsul: Project context for code summarization with llms. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 866–880.
- [29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
- [30] Walid Maalej, Zijad Kurtanović, Hadeer Nabil, and Christoph Stanik. 2016. On the automatic classification of app reviews. *Requirements Engineering* 21 (2016), 311–331.

- [31] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE transactions on software engineering* 43, 9 (2016), 817–847.
- [32] Stuart Mcilroy, Weiye Shang, Nasir Ali, and Ahmed E Hassan. 2017. User reviews of top mobile apps in Apple and Google app stores. *Commun. ACM* 60, 11 (2017), 62–67.
- [33] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 188–197.
- [34] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 263–272.
- [35] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
- [36] Feifei Niu, Wesley KG Assunção, LiGuo Huang, Christoph Mayr-Dorn, Jidong Ge, Bin Luo, and Alexander Egyed. 2023. Rat: A refactoring-aware traceability model for bug localization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 196–207.
- [37] Ehsan Noei, Feng Zhang, and Ying Zou. 2019. Too many user-reviews! what should app developers look at first? *IEEE Transactions on Software Engineering* 47, 2 (2019), 367–378.
- [38] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *arXiv preprint arXiv:1901.04085* (2019).
- [39] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and localizing change requests for mobile apps based on user reviews. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 106–117.
- [40] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 281–290.
- [41] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2016. Ardoc: App reviews development oriented classifier. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 1023–1027.
- [42] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 33, 6 (2007), 420–432.
- [43] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
- [44] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [45] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 191–201.
- [46] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).
- [47] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. CoCoSoDa: Effective Contrastive Learning for Code Search. *arXiv preprint arXiv:2204.03293* (2022).
- [48] David Sounthiraraj, Jared Hancock, Yassin Kortam, Ashok Javvaji, Prabhat Singh, and Shaila Shankar. 2025. Code-Craft: Hierarchical Graph-Based Code Summarization for Enhanced Context Retrieval. *arXiv preprint arXiv:2504.08975* (2025).
- [49] Statista. 2024. Number of Mobile Cellular Subscriptions Worldwide from 2010 to 2028 (in millions). <https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/> Accessed: 2024-12-08.
- [50] Statista. 2025. Frequency of mobile app updates on the Google Play Store as of January 2025. <https://www.statista.com/statistics/1404434/google-play-store-app-updates-by-frequency/> Accessed: 2025-1-12.
- [51] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd international conference on program comprehension*. 53–63.
- [52] Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–625.
- [53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

- [54] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 262–273.
- [55] Grant Williams and Anas Mahmoud. 2017. Mining twitter feeds for software user requirements. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*. IEEE, 1–10.
- [56] Huayao Wu, Wenjun Deng, Xintao Niu, and Changhai Nie. 2021. Identifying key features from app user reviews. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 922–932.
- [57] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate nearest neighbor negative contrastive learning for dense text retrieval. *arXiv preprint arXiv:2007.00808* (2020).
- [58] Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, et al. 2023. Lemur: Harmonizing natural language and code for language agents. *arXiv preprint arXiv:2310.06830* (2023).
- [59] Le Yu, Jiachi Chen, Hao Zhou, Xiapu Luo, and Kang Liu. 2018. Localizing function errors in mobile apps with user reviews. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 418–429.
- [60] Jinglei Zhang, Rui Xie, Wei Ye, Yuhang Zhang, and Shikun Zhang. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*. 219–229.
- [61] Tao Zhang, Jiachi Chen, Xian Zhan, Xiapu Luo, David Lo, and He Jiang. 2019. Where2Change: Change request localization for app reviews. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2590–2616.
- [62] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 14–24.
- [63] Yu Zhou, Yanqi Su, Taolue Chen, Zhiqiu Huang, Harald Gall, and Sebastiano Panichella. 2020. User review-based change file localization for mobile applications. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2755–2770.
- [64] Liming Zhu, Len Bass, and George Champlin-Scharff. 2016. DevOps and its practices. *IEEE software* 33, 3 (2016), 32–34.
- [65] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. 2020. Cooba: Cross-project bug localization via adversarial transfer learning. In *IJCAI*.