YourCoLo: Leveraging One-to-Many Relationships and Inter-Code Connections for User Review-Based Code Localization

KUO CHI and CHANGAN NIU, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
ZHOU YANG, Singapore Management University, Singapore, Singapore
CHUANYI LI, YI FENG, JIDONG GE, and BIN LUO, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
DAVID LO, Singapore Management University, Singapore, Singapore
VINCENT NG, Human Language Technology Research Institute, University of Texas at Dallas, Richardson, Texas, USA

In an era where mobile devices are ubiquitous, digital distribution platforms such as the Google Play Store have become integral to our daily lives, hosting millions of applications and serving billions of users. Users can leave reviews to provide developers with valuable feedback, including requests for new features and reports of issues. These user reviews play a crucial role in software development, testing, and maintenance by informing developers about user needs and potential problems, which motivates us to revisit a key problem: *given user reviews, how can we automatically identify the relevant code snippets from software codebases to assist developers in addressing the reviews?*

Existing practices to address this problem typically involve calculating the similarity between user reviews and code snippets. However, we identify three key limitations. First, although existing methods show promising results on individual projects, their high performance cannot be generalized across projects. Second, the state-of-the-art approach models the problem as a one-to-one relationship between a user review and code snippets, ignoring the one-to-many relationship that often exists. Third, the state-of-the-art approach focuses solely on the direct relationship between reviews and code snippets, overlooking the interconnections among code snippets themselves, which contain valuable information that can aid in accurately identifying relevant code. To address these limitations and advance the state of the art, we propose YourCoLo, a novel approach that fully leverages contextual information, one-to-many relationships, and inter-code connections.

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.
Authors' Contact Information: Kuo Chi, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: chikuo@smail.nju.edu.cn; Changan Niu, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: niu.ca@outlook.com; Zhou Yang, Singapore Management University, Singapore, Singapore; e-mail: zy25@ualberta.ca; Chuanyi Li (corresponding author), State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: lcy@nju.edu.cn; Yi Feng, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: fy@nju.edu.cn; Jidong Ge, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: gjd@nju.edu.cn; Bin Luo, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; e-mail: luobin@nju.edu.cn; David Lo, Singapore Management University, Singapore, Singapore; e-mail: davidlo@smu.edu.sg; Vincent Ng, Human Language Technology Research Institute, University of Texas at Dallas, Richardson, Texas, USA; e-mail: vince@hlt.utdallas.edu.
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.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/5-ART155
https://doi.org/10.1145/3769010

Specifically, YourCoLo is powered by three novel designs: (1) a prompt-enhanced mechanism to incorporate rich project-level context into code localization, (2) a new loss function designed to handle the one-to-many relationships between user reviews and multiple relevant code snippets, and (3) a ranking strategy that considers interconnections among related code snippets. Our experimental evaluation shows that YourCoLo substantially outperforms state-of-the-art models, surpassing CodeBERT, CodeLlama, and GraphCodeBERT by 18.3, 9.3, and 7.7 percentage points at the method level and by 18.4, 7.7, and 7.0 percentage points at the file level (in terms of mean reciprocal rank). In addition, YourCoLo also achieves improvements of 8.8 percentage points and 6.8 percentage points in mean average precision (MAP) at the method and file levels, respectively, compared to the state-of-the-art method. These results underscore YourCoLo's effectiveness and its potential to guide developers more accurately toward the code snippets most pertinent to user feedback.

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

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

ACM Reference format:
Kuo Chi, Changan Niu, Zhou Yang, Chuanyi Li, Yi Feng, Jidong Ge, Bin Luo, David Lo, and Vincent Ng. 2026. YourCoLo: Leveraging One-to-Many Relationships and Inter-Code Connections for User Review-Based Code Localization. *ACM Trans. Softw. Eng. Methodol.* 35, 6, Article 155 (May 2026), 36 pages.
https://doi.org/10.1145/3769010

1 Introduction

Recent reports [74] predict that by 2025, more than 7.49 billion people worldwide (that is, more than 93% of the whole population) will use mobile devices, fueling the rapid growth of platforms such as the App Store and Google Play. These platforms serve billions of users and host millions of applications, where users can provide valuable feedback to the applications [31, 49, 58, 61, 81]. This feedback, which includes issue discovery and requests for new features, is often expressed in the form of ***user reviews*** **(URs)** and plays a critical role in shaping software development, testing, and maintenance practices. Prior studies have categorized URs into several types, most notably **problem discovery (PD)**, which reports bugs or unexpected behaviors, and **feature request (FR)**, which suggests new features or enhancements [47]. For instance, a user mentioned in a review: "*On of my decks has a lot of sub decks. In last version it is not possible to expand it and see sub decks. Please fix it*," which is a typical PD highlighting a UI issue. For instance, a user mentioned in a review: "*Nice... But can you please also add the color feature? It's easy for us to remember words in color format*," which is an FR suggesting a new rendering feature. Developers can implement this feature by connecting it to relevant rendering or display components in the code. This approach reduces the cost of requirements analysis and improves the traceability of requirement changes [15, 70].

As mentioned by Harman and O'Hearn [24], developers are not short on bug reports or user feedback; they are, in fact, overwhelmed by the sheer volume of URs [48]. In real-world development scenarios, the process of selecting valuable URs is primarily performed through manual review. Developers typically identify useful reviews based on practical criteria such as clear problem descriptions, reproducibility, impact severity, and relevance to recent software versions. To manage this flood of information and better support software development, we focus on a key challenge: *Given URs, how can we automatically identify relevant code snippets within software codebases to assist developers in addressing these reviews?* By pinpointing the exact code segments that need attention, developers can save valuable time (which they would otherwise be spent searching for what to revise) and direct their efforts more effectively toward improving the code. To address this challenge, the *UR-based code localization task* [8] has been proposed. This task can be performed at different levels of granularity, typically categorized into method-level and file-level localization, where URs are

used to identify either specific methods or relevant files. In this task, we directly links URs to the relevant code elements rather than going through issues and **pull requests (PRs)**, for two main reasons. The first is the coverage limitations of issue and PR-based linkage. In fact, according to the report by Haering et al. [21], approximately 47% of URs appear before any related issue is created. This means that using issues as an intermediary would result in the omission of nearly half of the genuine and valuable user feedback, as these reviews never enter the GitHub issue or PR system. Developers do not necessarily create an issue for every review, especially for those of lower priority or those whose reproducibility has not yet been confirmed. Furthermore, issue/PR maintenance practices vary greatly between projects, and many projects lack any systematic tracking of URs. The second reason is to avoid process bottlenecks and ensure timely response. The study by Khatoonabadi et al. [33], which analyzed 265,325 PRs from 10 popular and mature GitHub projects, reported that 4,450 valid PRs were abandoned by contributors due to a lack of timely review and feedback, directly resulting in the loss of intended code changes. This finding highlights that when feedback must pass through intermediate stages (issues and PRs) before reaching the code, the entire feedback pipeline is vulnerable to disruption if any stage fails to respond promptly. By directly linking URs to code methods, we bypass potential bottlenecks such as issues and PRs and allow review information to flow immediately into the code localization process. This approach ensures that user feedback is mapped to specific methods or classes via the shortest possible path, thereby improving automation and timeliness in feedback handling, which is particularly crucial in fast-iteration, rapid-decision mobile application development environments.

Although UR-based code localization is related to two established tasks, bug localization [36] and code search [5, 10, 75], it presents distinct characteristics and challenges that merit independent investigation. Figure 1 compares these three tasks. Generally, all three tasks aim to return relevant code snippets given the input, but the characteristics of the inputs are different. All three tasks aim to return relevant code snippets based on given inputs. However, the nature of these inputs varies significantly. Bug localization uses bug reports, often authored by developers or technical support staff, which typically include rich details such as stack traces, error logs, screenshots, and even suggested fixes [29]. In contrast, code search queries are concise natural language descriptions, usually written by developers, that directly specify the desired functionality using technical jargon. However, URs are fundamentally different. They rarely contain technical terms, program-specific details, or additional diagnostic information such as execution logs or test cases. These differences mean that we cannot directly apply approaches tailored for bug localization and code search to UR-based code localization. Instead, methods must be designed specifically to interpret and connect user feedback with the relevant code snippets.

Currently, Hu et al. [27] proposed the **state-of-the-art (SOTA)** UR-based code localization approach: they use GraphCodeBERT to encode URs and code snippets into vectors. Then, they computed the similarity between the UR vector and the code snippet vectors. The code snippet with the highest similarity to the given UR is selected as the localized code. Although this approach has achieved promising results, we identify its three key limitations. The first limitation is *the lack of cross-project generalization*. Hu et al. show that, at the method level, the model fine-tuned to be applicable across all projects showed a 6.7% lower **mean reciprocal rank (MRR)** performance. Additionally, the accuracy of the top 5 and top 10 localization results decreased by 10% and 7.3%, respectively, compared to the models individually fine-tuned for each project. This suggests that current models still rely heavily on project-specific fine-tuning to achieve favorable results. However, such fine-tuning is often impractical in real-world scenarios due to the high cost of collecting labeled data for each project. This hinders the adaptability and scalability of current solutions. Therefore, establishing a cross-project general model is essential. A general model can eliminate the need for per-project fine-tuning, significantly reducing time and resource overhead. It

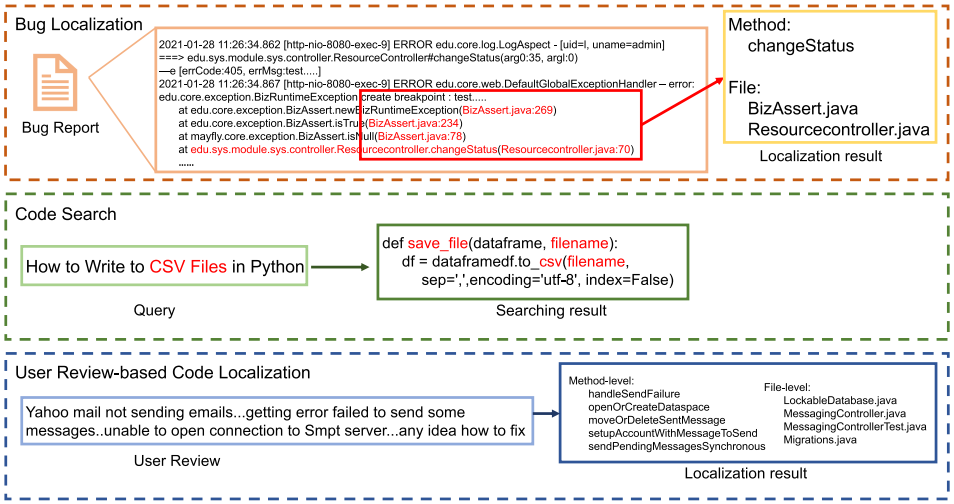


Fig. 1. Comparing the objectives, inputs, and outputs of bug localization, code search, and UR-based code localization.

also simplifies deployment and maintenance across multiple projects while maintaining efficiency as the number of projects grows. This direction aligns with the broader trend in **large language models (LLMs)**, where general-purpose models are increasingly favored over domain-specific fine-tuned models to strike a balance between performance, cost, and flexibility. Second, *the one-to-many relationship between a UR and code snippets is overlooked*. It is common for developers to update multiple source code snippets to address a UR, leading to a one-to-many relationship between URs and code snippets. However, the current SOTA [27] uses cross-entropy [67] as the loss function, which can only handle one-to-one relationships. Third, Hu et al. [27] *ignore the interconnection among code snippets* that correspond to the same UR. The code snippets that developers edit to address a UR are usually related; for example, the corresponding function calls shall be modified when updating the function parameters. Unfortunately, current methods fail to capture such interconnection between multiple code snippets, thereby reducing the model's effectiveness.

In this article, we propose **yet another user review-based code localization (YourCoLo)** approach. The proposed tool has three novel designs to address each aforementioned limitation of the baseline method [27]. *First*, we propose **cross-project prompting fine-tuning (CPFT)** to improve the cross-project generalization ability. When fine-tuning the model on some pairs of reviews and code snippets, CPFT adds the project information like project names (e.g., called *prompt* in our paper) to the model input. Introducing project information such as project names allows the model to explicitly recognize differences between projects, thereby reducing the negative impact of cross-project data variability on model performance and enabling the model to better handle data from different projects. *Then*, to account for the one-to-many relationship between the UR and code snippets, we extend the cross-entropy loss to propose a **cross-entropy loss based on the probability distribution of all positive samples (CEAPS)**. It allows a model to be trained to learn a UR's connection with multiple relevant code snippets. *Finally*, we propose the **locally coupled priority sorting (LCPS)** method to utilize the interconnection between code snippets that are relevant to the same UR. To retrieve such interconnection, we analyze the Review2Code dataset and perform statistical and frequency analysis on the code involved in each commit of each project to identify code snippets that frequently change together and have strong interconnections.

Intuitively, if a UR is highly relevant to code snippet A and the code snippets A and B tend to change together, this review is also likely to be related to B. Specifically, we construct a coupling dictionary by counting the frequency of co-modifications of code snippets within each commit. This dictionary records how often each pair of code snippets appears together in the same commit. After obtaining the interconnection, LCPS combines similarity and interconnection measures to re-rank code snippets associated with URs, ultimately enhancing the accuracy of the code localization task. We apply LCPS only in the inference phase, where it serves as a post-processing step to refine the ranking without interfering with the semantic learning process during training. To more clearly illustrate how our proposed methods address the identified challenges, we provide the following mapping summary: To improve cross-project generalization, we introduce CPFT, which incorporates project-level prompts into fine-tuning to help the model distinguish between different projects. To capture the one-to-many relationship between URs and code snippets, we propose CEAPS, a loss function that jointly optimizes all relevant code snippets for each review. To leverage the interconnection among code snippets, we design LCPS, a re-ranking strategy that utilizes co-change patterns from historical commits to enhance localization accuracy.

We evaluated the effectiveness of YourCoLo in the Review2Code dataset and compared YourCoLo with SOTA approaches [27]. Experimental results show that: (1) YourCoLo significantly outperforms the baseline, achieving average MRR scores that surpass CodeBERT, CodeLlama, and GraphCodeBERT by 18.3, 9.3, and 7.7 percentage points at the method level and by 18.4, 7.7, and 7 percentage points at the file level, respectively. YourCoLo achieves improvements of 8.8 percentage points and 6.8 percentage points in **mean average precision (MAP)** at the method and file levels, respectively, compared to the SOTA method. (2) All components, including CPFT, CEAPS, and LCPS, of YourCoLo are effective. Removing each of these three components separately results in mean MRR score decrease of 6, 5.1, and 1.3 percentage points at the method level, meanwhile, at the file level, the decreases are 5.3, 6.1, and 0.5 percentage points, respectively.

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

- We propose a novel UR-based code localization approach named YourCoLo to solve three limitations of the SOTA approach.
- We analyze the Review2code dataset and perform statistical and frequency analysis on the code involved in each commit of each project. These analyses provide valuable data support for future research.
- We evaluated the effectiveness of YourCoLo and compared it with the SOTA approaches with comprehensive experiments, and the results show that YourCoLo significantly outperforms the SOTA approach.

The remainder of the article is organized as follows. We introduce related work in Section 2. In Section 3, we provide details of each strategy in the YourCoLo approach. In Section 4, we describe the dataset, baselines, metrics, and other settings. Section 5 reports and discusses the performance of the proposed YourCoLo approach, analyzes the validity of each component, and evaluates the influence of hyperparameters on the experimental results. Threats to validity are discussed in Section 6, and Section 7 concludes the article and mentions future work.[1]

2 Related Work

2.1 UR

A UR [20, 26] is the user's feedback and evaluation of a product, service, or system, usually based on their experience with it. URs can be in the form of text, ratings, reviews, and so on, and usually

[1]Source code and data are available at the following link: https://github.com/Ch1Ku0/yourcolo.

appear on product pages, app stores, or other platforms. In the field of software engineering, UR plays an important role [58, 61, 80, 81], especially in code development for project maintenance and evolution [37, 88]. Specifically, during the release planning phase and development activities, URs can expose specific problems encountered by users in the use process, such as delays, inconvenience, crashes, and so on, which provide the most direct feedback to the development team and help to solve problems with a wide range of impacts [11]. 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 URs provide the test team with real usage scenarios, from which the importance of the UR can be seen [23, 60].

Consequently, we can see that URs serve as a direct source of user feedback and are widely applied throughout the development process, providing both theoretical foundations and practical demand for the proposed code localization method.

2.2 Code Localization

UR-Based Code Localization. Many approaches [8, 27, 59, 83, 85, 87] have been proposed to address the UR-based code localization problem. URR [8] classifies URs and source code files into predefined categories and then uses a retrieval tool to search and return the highest-scoring source files. ChangeAdvisor [59] groups URs that express similar needs and then uses cosine similarity to measure the similarity between source code snippets and these clusters. Where2Change [85] is similar to ChangeAdvisor [59] in clustering and similarity calculation, but it additionally measures the similarity between UR clusters and problem reports, which are detailed descriptions of software issues submitted by users or developers to help fix bugs. ReviewSolver [83] leverages contextual information in URs and correlates reviews with the bytecode of the app to locate functional errors in mobile apps. RISING [87] is similar to ChangeAdvisor [59] and Where2Change [85], with two main steps: clustering and similarity calculation. Additionally, it uses commit messages to bridge the lexical gap between URs and source code snippets. Most recently, Hu et al. [27] propose a large-scale, manually labeled ⟨UR, Code⟩ dataset named Review2Code, consisting of 68,359 pairs of URs 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.

Bug Localization. Bug localization [29, 34, 35, 40, 45, 46, 52, 63, 84] aims to identify the buggy code based on a given bug report, facilitating effective bug fix. In the realm of **information retrieval (IR)** techniques for bug localization, the **vector space model (VSM)** has been widely utilized due to its simplicity and effectiveness, particularly in the early stages of research. For example, BugLocator [86] employs the revised VSM to assess the textual similarity between bug reports and the source code, ranking source code files based on this similarity. Similarly, Locus [79] represents one of the earliest approaches to changeset-based bug localization by matching bug reports to specific hunks. VSM has also been popular for vectorizing URs and source code [16, 82]. Some other studies use topic models to measure the similarity between topic distributions in source code snippets and bug reports [51, 62]. Despite these advancements, the mismatch between bug reports and source code snippets continues to limit the performance of IR techniques. RAT [57] has shown that files modified together in previous bug-fixing commits are more likely to be involved in subsequent bugs. This insight has been used in various IR-based bug localization approaches. With the success of **deep learning (DL)** in natural language processing, DL-based methods have been increasingly applied to bug localization, including TRANP-CNN [36], which combines cross-project transfer learning with convolutional neural networks to address the cold start problem, COOBA [89], which uses adversarial transfer learning to transfer knowledge between projects while filtering out noise, and Bug2Commit [50], an unsupervised model that utilizes multiple data dimensions associated with bug reports and commits.

Early research on bug localization proposes a basic framework for text-code matching, which can be adapted to scenarios involving code localization based on URs. The success of DL in bug localization serves as a direct inspiration to address UR-based tasks. However, locating code based on URs is more open-ended, with textual expressions closer to conversational language, requiring methodological innovation to address these challenges.

Code Search. The UR-based code localization task requires solving the semantic mapping between natural language and code, similar to the core challenge of the code search task. The development of code search has evolved from traditional IR techniques [25, 69] to DL techniques and can be divided into four stages. Initially, IR-based code search methods were primarily based on keyword matching. With the advancement of DL, research began to focus on constructing neural networks [5, 42, 73], where queries and code were encoded into a shared embedding space to capture the semantic relationship between queries and code snippets [5, 17]. This progress led to the emergence of code pre-trained models, such as CodeBERT [13], GraphCodeBERT [19], UniXcoder [18], and SPT-code [54], which significantly improved code search effectiveness. Recent studies have further enhanced the performance and fine-tuning efficiency of pre-trained models through techniques like contrastive learning [38, 71], layer freezing [72], and adapter fine-tuning [77], making code search technology more efficient and precise. As large-scale models gain prominence, GPT-based [65] models such as Codex [7] and GPT-4 [1] have also been applied in the field of code search, demonstrating remarkable performance. These models have further advanced the development of code search technology.

Research on code search provides important theoretical support for solving semantic mapping between natural language and code. However, the task of UR-based code localization faces additional challenges due to the non-technical nature and semantic ambiguity of URs.

2.3 Pre-Trained Models of Code

The development of code pre-trained models provides a stronger foundation for the advancement of UR-based code localization. Several pre-trained models for programming languages [9, 13, 18, 19, 32, 32, 54] have been proposed, significantly advancing the development of code representation. Similarly, the evaluation of pre-trained models has evolved rapidly. Niu et al. [53, 55] evaluate and compare pre-trained models of source code on a wide range of code-related downstream tasks. Ahmed and Devanbu [2] study the value of multilingual fine-tuning for pre-trained models on a variety of tasks. CrossCodeBench [56] is proposed to evaluate the cross-task and in-context learning capacity of code models. Among them, there are several pre-trained models widely studied, used, and discussed by researchers. CodeBERT [13] is a pre-trained dual-modal encoder designed for both code and natural language text. It not only captures the semantic relationship between code and natural language but also effectively supports code localization tasks. GraphCodeBERT [19] incorporates information about the structural and semantic relationships within the code, improving the representation of the code by capturing complex dependencies. UniXcoder [18] is designed for code understanding and generation tasks, which can effectively understand the structure, syntax, and semantics of the code and effectively integrate the syntax structure of the code with the semantic information. CodeLlama [66] is a specialized LLM designed for programming tasks, focusing on code generation, comprehension, and analysis. It is trained on extensive datasets of code and natural language, enabling it to handle natural language queries about code and produce or understand code snippets effectively. CodeLlama aims to assist developers by providing relevant code suggestions and insights based on their programming needs. Due to the architectural characteristics of GPT [14], we cannot directly obtain a single global vector representation from its output. Therefore, in this article, we generate a vector representing the entire input sequence by performing a weighted average of the hidden states of the unfilled tokens. The advantage of this

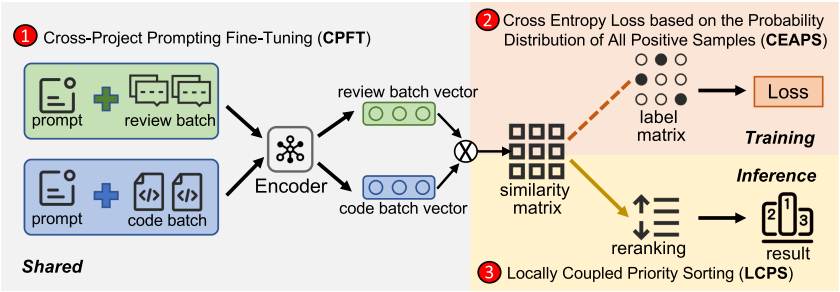


Fig. 2. The overview of YourCoLo's technical design.

method is that it effectively aggregates the token vectors output by the model to generate a unified vector representation of the entire sequence.

3 Proposed Approach

In this section, we provide the details of each strategy in the YourCoLo approach. The overall process of the proposed approach YourCoLo is shown in Figure 2, including the training phase and the inference phase. The first strategy is CPFT. During the training phase, we fine-tune the model to obtain an encoder that generates UR vectors and code vectors. Prompts are added during the fine-tuning process to provide the model with contextual information about the project, enhancing its adaptability to fine-tuning across multiple projects. The second strategy is CEAPS in the training phase so that the model can better understand the complex association between reviews and multiple code snippets. The third strategy is LCPS in the inference phase, we analyze the historical commit frequency of code, identify code snippets that are frequently co-modified, and use these coupling relationships to optimize the ranking of initially matched code snippets.

3.1 CPFT

Considering the limitation that current approaches, such as UNIF [5], CAT [22], CodeBERT [13], and GraphCodeBERT [19], which can only be fine-tuned for individual projects, the process of creating separate fine-tuned models for each project is both costly and inefficient, as fine-tuning a universal model across multiple projects leads to degraded performance. For example, Hu et al. [27] show that: compared to models fine-tuned separately for each project, fine-tuning GraphCodeBERT across all project data resulted in a 6.7% decrease in MRR performance, while fine-tuning CodeBERT under the same conditions led to a 22.3% decrease. Furthermore, for GraphCodeBERT, the precision of the Top-1 and Top-3 localization results decreased by 5% and 11.9%, respectively, while the accuracy of the Top-5 and Top-10 results decreased by 10% and 7.3%, respectively. To address this issue, we propose a strategy named CPFT. We design prompts that incorporate project-specific information, enabling the model to generalize across multiple projects without the need for individual fine-tuning.

Prompts are structured inputs that are used to guide the behavior of the model or improve its understanding by providing additional contextual information [44]. Prompts often include instructions, placeholders, or specific tokens that frame the input in a way that highlights the task objectives. For example, a prompt can indicate the type of task, the domain of the input, or specific constraints that the model should follow.

As shown in Figure 3, CPFT's prompt consists of two parts: the first placeholder is filled with the project name to which the sample belongs, and the second placeholder is filled with a UR or code sequence. This structure provides the model with an explicit context about the project at the beginning of the prompt, allowing it to better distinguish between projects.

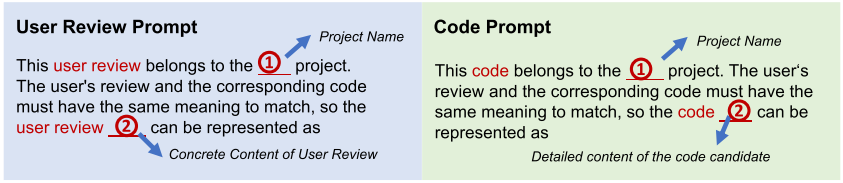


Fig. 3. The templates of UR prompt and code prompt.

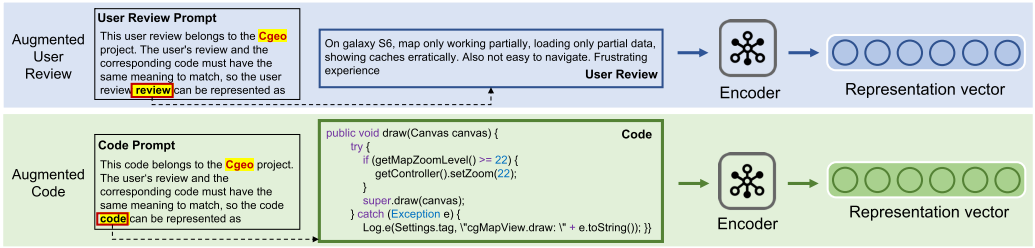


Fig. 4. Prompt components for enhancing model understanding of project context, semantic matching, and code representation.

The motivation for this design is twofold. First, by explicitly identifying the project, the model gains a clear context about differences and associations across projects, reducing the negative impact of cross-project data variability on model performance, and improving its understanding of cross-project relationships. Second, the inclusion of a UR or code sequence in the second half helps the model focus on semantic matching, emphasizing that URs and code snippets should align semantically. This dual-purpose prompt structure enhances the model's ability to learn project-specific nuances while maintaining semantic consistency.

As shown in Figure 4, we chose manually designed and fixed prompts that are inserted directly into the input of the model without further optimization. The prompt and UR snippet is concatenated and input into the model, and then a vector for similarity calculation is generated through the model encoding. The same process is applied to the prompt and the code snippet. This approach enables quick application in UR-based code localization tasks. Since these prompts rely on domain-specific knowledge, such as project names, or simple heuristics, such as the mapping description between URs and code snippets, they do not require large amounts of training data.

By adding prompt information specific to each project to the training data, we provide a clear context that helps the model better understand and differentiate between projects, guiding it to prioritize relevant information during inference. As a result, this approach is able to strengthen the model's cross-project generalization.

3.2 CEAPS

A key feature of UR-based code localization is the one-to-many relationship, where a single UR often corresponds to multiple code changes. As shown in Figure 5, there are multiple code snippets related to the issues mentioned in the UR. However, the SOTA approach [27] uses cross-entropy [67] as the loss function, which overlooks this one-to-many relationship. Figure 6 illustrates a practical example, where URs and code snippets are presented along with their corresponding pairs. Notably, there is a one-to-many relationship between URs and code snippets. For instance, review r_1 corresponds to both code snippets c_1 and c_4, and the same applies to review r_4, which corresponds to c_1 and c_4 as well. The cross-entropy loss function only considers a match between r_1 and c_1,



Fig. 5. An example illustrating the one-to-many relationship between a UR and code snippets.

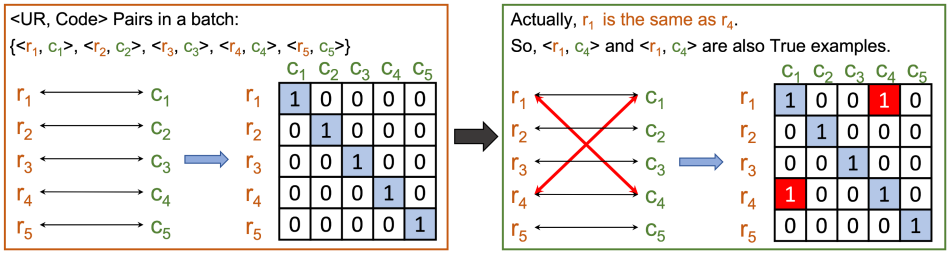


Fig. 6. The actual relationships between the review and code set within a batch during model training.

ignoring that r_1 and r_4 are the same UR. As a result, the matching relationship between r_1 and c_4, and r_4 and c_1 is neglected in the loss calculation, which can lead to a suboptimal performance of the model.

To address this issue, we propose a method called CEAPS. CEAPS modifies the cross-entropy loss function to ensure that all relevant code snippets (positive samples) are considered in the loss calculation, thereby preventing the loss of important positive samples. This modification enables the model to properly account for the one-to-many relationship between URs and code snippets during loss computation. The following are the details of the CEAPS method.

Similarity Calculation and Probability Distribution Transformation. For each review r_i and code c_j, their similarity score $s(r_i, c_j)$ is calculated using the inner product of their vector representations. This results in a similarity matrix:

$$P(c_j, r_i) = \frac{e^{s(r_i, c_j)}}{\sum_{k=1}^{M} e^{s(r_i, c_k)}}, \tag{1}$$

where $P(c_j, r_i)$ is the probability of matching between the code snippet c_j and the review r_i, $s(r_i, c_j)$ is the similarity score and M is the number of code snippets.

Tag Definition. Define a binary label matrix Y, if the reviews r_i are related to code snippet c_j, we set $Y_{ij} = 1$, otherwise, $Y_{ij} = 0$, for all samples in a batch. A matrix is generated as shown in Figure 6.

Loss Calculation. The loss is calculated by the probability distribution $P(c_j, r_i)$ and the label matrix Y. The formula is as follows.

$$L(r_i) = -\sum_{j=1}^{M} Y_{ij} \log(P(c_j, r_i)). \tag{2}$$

The following takes review r_1, as an example to illustrate the loss calculation process. For the review r_1, which is matched by the codes c_1 and c_4, the loss should be evaluated as follows:

$$\begin{aligned} L_{r_1} = -\big[& Y_{r1,c1} \log(P(c1, r1)) + Y_{r1,c2} \log(P(c2, r1)) \\ & + Y_{r1,c3} \log(P(c3, r1)) + Y_{r1,c4} \log(P(c4, r1)) \\ & + Y_{r1,c5} \log(P(c5, c1)) \big]. \end{aligned} \tag{3}$$

Both $Y_{r1,c1}$ and $Y_{r1,c4}$ are equal to 1, indicating that the review r_1 is matched to the codes c_1 and c_4. In contrast, $Y_{r1,c2}$, $Y_{r1,c3}$, and $Y_{r1,c5}$ are all 0, indicating that there is no relationship between review r_1 and codes c_2, c_3, c_5.

Total Loss. Finally, the loss of all reviews in the batch is averaged to obtain the overall loss of the batch. N indicates the batch size.

$$L_{\text{batch}} = \frac{1}{N} \sum_{i=1}^{N} L(r_i). \tag{4}$$

The final loss is calculated using the CEAPS method, which considers all code snippets associated with each UR. The modified loss function better reflects the actual one-to-many relationship between URs and code snippets and avoids the problem of information loss in one-to-one matching.

3.3 LCPS

Existing models used by the SOTA approach [27], such as CodeBERT [13] and GraphCodeBERT [19], also fail to capture the interconnection between code snippets that correspond to the same UR, resulting in suboptimal performance. This limitation arises because code snippets that are frequently modified together or share a strong contextual relationship may not be prioritized appropriately in the results.

Inspired by the work on refactoring-aware traceability models for bug localization [57] and cross-task artifact dependency mining [3], we recognize the potential of incorporating historical commit data to measure and enhance the coupling strength between code snippets. The former demonstrates that accounting for co-modification patterns and maintaining accurate code history through refactoring-aware techniques can significantly improve the effectiveness of localization approaches. Meanwhile, the latter highlights the importance of leveraging historical developer interactions to uncover dependencies across different tasks, which inspires us to consider frequent co-occurrences of code modifications as an indicator of strong semantic relationships. Building on these insights, we also draw inspiration from AmaLgam [78], which leverages version history to improve bug localization. Like AmaLgam, we emphasize the significance of historical commit data in assessing code relationships. Our approach enhances code snippet ranking by analyzing co-modification patterns, and we evaluate it on real-world datasets, following AmaLgam's empirical validation methodology, to measure the impact of historical data on ranking effectiveness. By integrating these ideas, we refine code ranking by prioritizing historically coupled code snippets, ultimately improving the accuracy of our localization method.

Therefore, we aim to incorporate historical commit data to measure and enhance the coupling strength between code snippets, thereby improving the accuracy of localization results. We propose a LCPS method to improve the correlation ordering of code snippets through coupling optimization in a local scope. To better target more snippets of code associated with reviews, this method introduces the code's historical commit frequency as a cofactor, which serves as a measure of how often code snippets have been modified together in the past, as a basis for re-ranking the results of code localization. By analyzing the cooperative modification of the code in the history submission and using the historical commit frequency between code snippets as a measure of code coupling, the initially matched code snippets can be reordered to optimize the localization results.

The LCPS method is divided into two parts: the first involves statistics and analysis of the dataset and the second focuses on re-ranking.

3.3.1 Statistical Analysis of the Dataset. Given that there often exist multiple code snippets corresponding to a single UR, it is important for a UR-based code localization method to identify and capture such an interconnection between multiple code snippets.

However, the URs come from Google's App Store (Google Play), while the code is sourced from GitHub, making them from two separate platforms. Furthermore, reviews are selected to ensure they are posted after the corresponding code commits, preventing any data leakage into the model training process [27].

Consequently, there is no direct connection between code commits and URs. This makes it challenging to infer the coupling relationships between code snippets directly from the reviews. URs are often subjective descriptions expressed in natural language, lacking the technical detail or relevance needed for this task. For example, a review might describe a user experience issue (e.g., "mail does not sync") without addressing the underlying code implementation. Reviews typically focus on symptoms rather than the root cause, further complicating the direct mapping to specific code snippets.

In contrast, historical commit data represents an objective accumulation of information recorded during the development process. Commit records to actual collaborative code modifications, capturing the true logical or functional relationships between code reviews. By analyzing the frequency of these co-modifications, it becomes possible to identify patterns that reflect developer intent and technical dependencies. This objectivity makes historical commit data a reliable source for extracting coupling relationships between code snippets. While the UR-based code localization task's primary goal remains driven by URs, the frequency of collaborative changes in historical commits can serve as an added feature to enhance the prioritization of code snippets.

Figure 8 shows the statistical analysis of our dataset. The first step is to count the code snippets involved in each commit based on commit information from the samples in the training dataset. Figure 9 shows a sample from *K-9 Mail* project, this sample includes the fields "commit_id," "method_path," and "method_name." The "commit_id" field represents the unique commit hash for each commit, while "method_path" and "method_name" indicate the filename and method name involved in the sample, respectively.

We perform statistics based on the "commit_id" field in each sample. Code snippets with the same "commit_id" are grouped together, creating a dictionary where each "commit_id" serves as the key, and the value is the list of method names associated with that commit, thereby obtaining the *Function List*.

We analyze the coupling relationships between the code snippets in each commit in the *Function List*. For example, as shown in Figure 7, the method *synchronizeMailboxSynchronous* calls the *moveToSpam* method, indicating a calling relationship between them. Additionally, there may be different types of coupling relationships between these simultaneously committed code snippets, including but not limited to:

— *Calling Relation*: One snippet of code may call another, so they are logically closely related.
— *Data Dependency*: Different pieces of code may share the same data structure or global variables, and modifying one piece may require tweaking another.
— *Co-Change Frequency*: Multiple code snippets are often modified in the same commit, reflecting their coupling in maintenance and evolution.
— *Location Proximity*: Snippets of code in the same class or file, even though they have no calling relationship, are often modified at the same time because of the requirements of the same functional module.

The next step is to count the number of historical co-modifications between code snippets in each commit. We perform statistical analysis on the code methods involved in each commit. We first define a coupling dictionary *Frequent_dict*, where the key of the dictionary is a tuple of the paths (file path + method name) of the two code snippets, and the value is the number of times they occur in the same commit. For each commit, if two pieces of code appear together in the

Function name: synchronizeMailboxSynchronous

```
977  976                 for (Message localMessage : allMessages) {
978  977                     if (localMessage.getSpamFlag().equalsIgnoreCase("YES")) {
979       -                         Log.d("K9-Mail", "Message " +  Address.toString(localMessage.getFrom()) + " is SPAM");
980       -                         moveToSpam(localMessage, remoteFolder);
     978  +                         // Log.d("K9-Mail", "Message " +  Address.toString(localMessage.getFrom()) + " is SPAM");
     979  +                         String spamfolder = account.getSpamFolderName();
     980  +                         String inbox = account.getInboxFolderName();
     981  +                         moveToSpam(localMessage, account, inbox, spamfolder);
981  982                           continue;
982  983                   } else {
983       -                         Log.d("K9-Mail", "Message " +  Address.toString(localMessage.getFrom()) + " is not SPAM");
     984  +                         // Log.d("K9-Mail", "Message " +  Address.toString(localMessage.getFrom()) + " is not SPAM");
984  985                   }
985  986           }
```

Changed code

Function name: moveToSpam

```
1052       -    private void moveToSpam(Message message, Folder localFolder) {
      1053 +    private void moveToSpam(Message message, Account account, String inbox, String spamFolder) {
1053 1054           // Move the message to the user specified spam folder.
1054 1055       Log.d("K9-Mail", "moving messages to SPAM folder, from: " + Address.toString(message.getFrom()));
1055       -           Folder spamFolder = null;
1056       -           try {
1057       -               localFolder.moveMessages(new Message[] { message }, spamFolder);
1058       -           } catch (MessagingException e) {
1059       -               // TODO Auto-generated catch block
1060       -               e.printStackTrace();
1061       -           }
      1056 +           moveMessage(account, inbox, message, spamFolder, null);
1062 1057       }
1063 1058
1064 1059       private void closeFolder(Folder f) {
```

Changed code

Fig. 7. Example of a coupling relationship between two code snippets, in which one snippet calls the other.

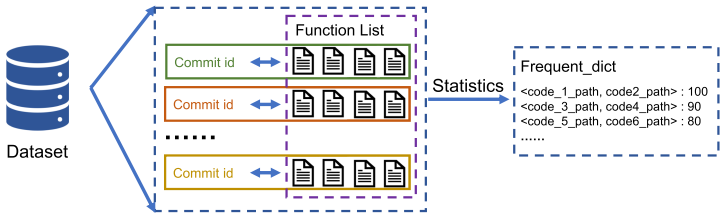


Fig. 8. The process of creating coupling dictionary *Frequent_dict*.


```
{
  "review_id": 1129,
  "review_raw": "Won't sync to my active sync server. Tried every combination I can think of.
The web suggestions are way out-of-date, some of the options don't exist.",
  "commit_id": "3e1a8da32a7e2bbb55441acf6e5de26e707ac878",
  "method_path": "app/core/src/main/java/com/fsck/k9/service/CoreService.java",
  "method_name": "isMailSyncDisabled",
  "method_content":public static boolean isMailSyncDisabled(Context context) {
                    final boolean hasConnectivity = Utility.hasConnectivity(context);
                    final boolean autoSync = ContentResolver.getMasterSyncAutomatically();
                    boolean doBackground = true;
                    K9.BACKGROUND_OPS bOps = K9.getBackgroundOps();
                    switch (bOps) {
                      case NEVER:
                        doBackground = false;
                        break;
                      case ALWAYS:
                        doBackground = true;
                        break;
                      case WHEN_CHECKED_AUTO_SYNC:
                        doBackground = autoSync;
                        break;
                    }
}
```


Fig. 9. One sample from *K-9 Mail* project.

Table 1. Review2code Dataset Statistics

Name	Reviews	Methods	Files	Commits	Frequent_dict Length	Pairs	
						<UR., M.>	<UR., F.>
K-9 Mail	3,209	2,741	371	913	46,817	33,351	16,502
AntennaPod	595	1,498	378	570	23,467	9,279	4,777
Cgeo	658	1,623	477	853	12,635	5,420	2,952
Anki	1,211	2,118	361	1,076	57,830	13,276	5,624
Termux	1,085	189	52	151	999	7,033	3,585
Total	6,758	8,169	1,639	3,563	141,748	68,359	33,440

Pair of Code Snippets	Frequency
< k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onActivityResul, k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onPendingIntentResult >	25
< k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onCryptoClic, k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#startExtractingTextAndAttachments >	24
< k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onCryptoClick, k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onPendingIntentResult >	24
< k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onActivityResult, k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onCryptoClick >	24
< k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#onPendingIntentResult, k9mail/src/main/java/com/fsck/k9/ui/messageview/MessageViewFragment.java#startExtractingTextAndAttachments >	24

Fig. 10. Top five entries from the *Frequent_dict* of app *K-9 Mail*.

same commit, we assume that there is some coupling between them. Each pair of code snippets ⟨code1_path, code2_path⟩ that appear in the same commit is counted. Finally, the dictionary stores the number of common changes for each pair of code snippets in the format ⟨(code1_path, code2_path): frequency⟩, where code1 and code2 represent the two code snippets, and frequency represents the number of times they occur together in the commit.

The statistics of the total commits and the length of the *Frequent_dict* dictionary involved in each app project can be seen in Table 1. We show the top five entries from the *Frequent_dict* of the app *K-9 Mail* in Figure 10. On the left side of the figure is the dictionary key, representing each pair of code snippets. Each code path consists of a file path and function name, joined with a "#." The number on the right indicates how many times a pair of code snippets appears together.

3.3.2 Re-Ranking. After constructing the coupling dictionary *Frequent_dict*, the core idea of this method is to insert the K methods that are most coupled to the current K code methods into subsequent positions (i.e., K+1 to 2K positions); there may be duplicates in the newly inserted K code snippets because the strongest coupling of different code snippets may be the same one, so the newly inserted K code snippets are de-processed on the basis of retaining the original order. Thus, forming a strongly coupled code snippet on a local scale. This value K is called a coupling window, reflecting the role of the value K in controlling the number of coupling snippets in the sequencing window.

4 Experimental Setups

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

4.1 Dataset

Regarding the dataset, we adopt Review2Code [27], a large-scale human-labeled ⟨UR, Code⟩ ground-truth dataset, introduced for the first time. We chose this dataset for evaluation because this dataset is the most recent proposed for the UR-based code localization task.

Review2Code provides detailed statistical data from five open source mobile applications. The URs in the dataset are collected from Google's App Store (i.e., Google Play), while the code is sourced from GitHub. In the dataset, URs 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 URs to predict relevant code, making the reviews selected during testing more reflective of real-life scenarios [27].

We analyze the samples in this dataset, counting the number of commits in each mobile application project and the historical frequency of commits between code snippets. The dataset includes information on URs (Reviews), Code Methods (Methods), Files, and Pairings. In total, the dataset contains 68,359 ⟨UR, Method⟩ pairs and 33,440 ⟨UR, File⟩ 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 [27]. Detailed information is presented in Table 1.

4.2 Baseline

We first adopt *BM25* and *TF-IDF*, which are classical IR methods for ranking code snippets based on their lexical relevance to URs. BM25 is a probabilistic ranking function, while TF-IDF is the core weighting scheme used in the VSM with cosine similarity as the similarity metric. Both methods serve as strong traditional baselines in IR-based retrieval tasks. Beyond these, we further include several representative approaches from related domains. First, we incorporate *CodeMatcher* [43], a recent IR-based code search method that combines traditional keyword-based retrieval with deep semantic understanding. To bridge the semantic gap between natural language queries and code snippets, CodeMatcher employs a fuzzy matching strategy based on important keywords, followed by a sequential matching-based re-ranking mechanism. Its effectiveness has been validated on a large-scale dataset covering approximately 41k repositories, where it demonstrated SOTA performance. We also include *BLUiR* [68], a classic and structured IR-based bug localization method. BLUiR extracts program constructs by building **abstract syntax trees (ASTs)** from source code and applies the Okapi BM25 model to compute the similarity between bug reports and candidate code changes. BLUiR is widely recognized in the software engineering community for its effectiveness in integrating structural information into IR and has laid the foundation for a line of research in IR-based bug localization. Since its original implementation is not publicly available, we carefully re-implemented it based on the technical descriptions in the seminal paper to ensure fair and valid comparisons. We also include *FLIM* [41], a SOTA bug localization method that leverages pre-trained language models, such as CodeBERT, to capture deep semantic representations of both bug reports and code elements. FLIM integrates these representations into a learning-to-rank framework, which takes the semantic embeddings of bug reports and code elements as input features and trains a ranking model to prioritize code elements that are most likely relevant to a given bug report. In the bug localization task, FLIM outperforms traditional IR-based methods such as BugLocator [86] and BLUiR [68], IR combined with machine learning methods such as AdaptiveBL [12], and DL approaches such as DreamLoc [64], TRANP-CNN [30], and LS-CNN [28], achieving SOTA performance. Finally, we select *CodeBERT* [13] and *GraphCodeBERT* [19] as baselines, given their strong performance reported by Hu et al. [27]. Notably, GraphCodeBERT has achieved SOTA performance on the user-based code localization task.

Additionally, we select *CodeLlama* [66] as another baseline. As pre-train research continues to progress, LLMs have demonstrated strong capabilities across a wide range of tasks. CodeLlama, in particular, has shown remarkable performance on code-related tasks such as code generation, code review, and code completion [4, 6, 39]. However, no studies to date have applied CodeLlama to UR-based code localization tasks. By including it as a baseline, we aim to evaluate its performance in

this context and provide a more comprehensive comparative analysis. The version of CodeLlama we select is CodeLlama-13B, which has 13 billion parameters. It strikes a balance between performance and computational cost, capturing complex semantics better than smaller versions (e.g., CodeLlama-7B) while requiring fewer resources than larger ones (e.g., CodeLlama-34B), making it suitable for most research environments.

In our experiments, for models with encoder architecture, that is, CodeBERT and GraphCodeBERT, the fine-tuning scheme and dataset follow the approach of Hu et al. [27] and then we use them to encode URs and code snippets into vectors. Specifically, both models encode the input (URs and code) into a shared semantic space, separately. We obtain the representation vector of the [CLS] token in the output vector as the representation vector of the entire input sequence. For CodeLlama, which uses a decoder architecture, we generate a vector for the entire input sequence by performing an average of the hidden states of the unfilled tokens. After obtaining the vectors for URs and code snippets, we calculate their similarity, and the code snippet most similar to a given UR is selected as a result of code localization. CodeLlama adopts the same fine-tuning scheme and dataset as CodeBERT and GraphCodeBERT.

4.3 Metrics

To evaluate the performance of the models, we use the MRR and Top-k as metrics.

(1) *MRR* is a commonly employed metric in IR to assess the effectiveness of ranking systems. Calculating 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:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank_i}, \tag{5}$$

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

(2) *MAP* is a widely used metric in IR for evaluating the performance of ranking systems, particularly when multiple relevant documents may exist for each query. The calculation of MAP involves computing the **average precision (AP)** for each query and then taking the mean of these values across all queries in the evaluation dataset. Specifically, it is defined as:

$$MAP = \frac{1}{|Q|}\sum_{i=1}^{|Q|}AP_i, \tag{6}$$

where Q denotes the set of queries, and AP_i is the Average Precision of the ith query.

The Average Precision AP_i measures the ranking quality by averaging the precision scores at the ranks where relevant documents appear. It can be formulated as:

$$AP_i = \frac{1}{|R_i|}\sum_{k=1}^{N}P_i(k)\cdot rel_i(k), \tag{7}$$

where $|R_i|$ is the total number of relevant documents for query i, N is the number of retrieved documents, $P_i(k)$ 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.

In summary, MAP takes into account both the precision and the rank positions of relevant documents, offering a comprehensive evaluation of the overall ranking performance across multiple queries.

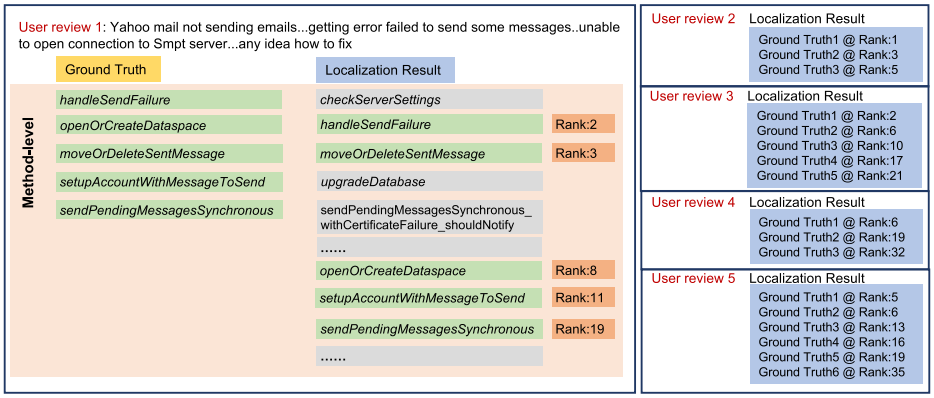


Fig. 11. A concrete code localization example excerpted from the experiments, showing the localization results for a query set of five URs.

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

$$Top\text{-}k(UR) = \frac{\sum_{ur \in UR} isCorrect(ur, top\text{-}k)}{|UR|}, \tag{8}$$

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

Next, we use a concrete example to illustrate how a single result of code localization through a UR in the test dataset contributes to the overall metric calculation. Assume that the test dataset contains $Q = 5$ URs. Figure 11 presents a concrete example of UR-based code localization, where the correct code snippet ranks second in the localization results. So the *Rank* is 1/2, the $isCorrect(ur, top\text{-}1) = 0$, the $isCorrect(ur, top\text{-}3) = 1$, the $isCorrect(ur, top\text{-}5) = 1$. Suppose the ranks of the correct results in the remaining four code location results for the URs are 1, 2, 6, and 5, respectively. The MRR is calculated as follows:

$$MRR = \frac{1}{5}\left(\frac{1}{2} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{5}\right). \tag{9}$$

The calculation process of *AP* of *UR 1* is as follows:

$$AP = \frac{1}{5}\left(\frac{1}{2} + \frac{2}{3} + \frac{3}{8} + \frac{4}{11} + \frac{5}{19}\right) \approx 0.4337. \tag{10}$$

The *AP* of the remaining *UR* is 0.333, 0.625, 0.125, and 0.25, and the calculation process for *MAP* is as follows:

$$MAP = \frac{1}{5}(0.4337 + 0.7556 + 0.3213 + 0.1219 + 0.2227). \tag{11}$$

The calculation process of *Top*-5 is as follows:

$$Top\text{-}5 = \frac{1}{5}(1 + 1 + 1 + 0 + 1). \tag{12}$$

4.4 Other Settings

We chose GraphCodeBERT and UniXCoder as the backbone model of YourCoLo, 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.

For baselines, the code and hyper-parameters setup are completely derived from the source code in the GitHub repository of Hu et al. [27].

We set the coupling window size K to 3. The maximum sequence lengths of the code snippets and URs 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.

5 Results and Discussion

In this section, we report and discuss the performance of our proposed YourCoLo approach, analyze the validity of each component of the YourCoLo approach, and evaluate the influence of the hyperparameters involved on the experimental results. This is done by answering the following three **research questions (RQs)**.

(1) RQ1: How well does YourCoLo perform?
(2) RQ2: How much do different components contribute to the code localization performance?
(3) RQ3: What is the impact of different hyperparameters?

5.1 RQ1: How Well Does YourCoLo Perform?

Method. The BM25 algorithm does not require any model fine-tuning. In this setup, URs serve as queries, while code snippets serve as documents, enabling the retrieval of relevant code based on URs. For both CodeBERT and GraphCodeBERT, we adopt the training method described in previous studies [27], where the model is fine-tuned on data from a single project. This training scheme is referred to as **train on single (ToS)**. This is because previous research has shown that fine-tuning using data from all projects does not yield the same results as fine-tuning using data from a single project. Therefore, we chose the method that performs best for code localization in our previous work: fine-tuning a model individually for each project using data from a single project. In contrast, for both CodeLlama and our proposed YourCoLo approach, the model is fine-tuned using prompt-enhanced data from all projects. This training scheme is referred to as **train on all (ToA)**. This approach aims to demonstrate that our proposed method is applicable to all projects and does not need to be fine-tuned individually for each project. Finally, we test the models obtained through the different methods outlined above on each project.

Results. As shown in Table 2, for the overall performance of code localization, YourCoLo(UniXcoder) outperforms other benchmark models in most cases, especially in the performance of high Top-K values (Top-10, Top-20), MRR, and MAP. At the method level, YourCoLo(UniXcoder) significantly outperforms the other models in all applications, especially on *Termux* application, where YourCoLo(UniXcoder)'s MRR reached 0.752, the Top-20 reached 0.825, and YourCoLo(UniXcoder)'s MAP achieved 0.443. At the file level, YourCoLo(UniXcoder) also achieved the highest MRR and MAP value for most applications. In *K-9 Mail*, *AntennaPod*, YourCoLo(UniXcoder) has significantly improved in the accuracy of the Top-20 and maintains a lead in MRR.

Discussion. Classical IR techniques such as BM25 and TF-IDF yield relatively low performance, highlighting their limitations in bridging the semantic gap between URs and code snippets. Although both methods are robust in traditional lexical matching, they fall short in capturing the contextual or structural of code. In contrast, CodeMatcher, a recent IR-based code search method, demonstrates superior performance over BM25 and TF-IDF. Despite not employing DL, CodeMatcher effectively incorporates keyword extraction and fuzzy matching strategies, followed by a sequential re-ranking mechanism. This hybrid yet lightweight design enables it to better align user queries with relevant code segments. Interestingly, BLUiR, a classic IR-based bug localization method that integrates

Table 2. The Performance of Different Methods on Locating Code at Method Level and File Level

APP Name	Model	Scheme	Method Level							File Level						
			Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
K-9 Mail	BM25	None	0.012	0.024	0.046	0.077	0.127	0.036	0.01	0.052	0.14	0.196	0.314	0.454	0.13	0.031
	TF-IDF	None	0.006	0.018	0.031	0.04	0.105	0.026	0.007	0.046	0.096	0.155	0.255	0.445	0.107	0.047
	CodeMatcher	None	0.062	0.1	0.115	0.15	0.23	0.092	0.051	0.096	0.174	0.227	0.37	0.517	0.159	0.069
	BLUiR	None	0.006	0.009	0.015	0.046	0.077	0.02	0.006	0.062	0.199	0.249	0.373	0.504	0.16	0.072
	FLIM	ToS	0.082	0.156	0.238	0.302	0.348	0.144	0.127	0.128	0.321	0.401	0.507	0.622	0.231	0.15
	CodeBERT	ToS	0.118	0.246	0.299	0.411	0.508	0.205	0.181	0.178	0.355	0.421	0.583	0.698	0.28	0.187
	GraphCodeBERT	ToS	0.29	0.421	0.486	0.579	0.698	0.369	0.232	0.336	**0.539**	0.614	0.732	0.832	0.437	0.269
	CodeLlama	ToS	0.28	0.401	0.49	0.576	0.67	0.34	0.22	0.32	0.502	0.623	0.75	0.852	0.423	0.236
	YourCoLo(GraphCodeBERT)	ToA	0.227	0.342	0.404	0.473	0.554	0.313	0.21	0.267	0.442	0.523	0.641	0.76	0.377	0.218
	YourCoLo(UniXcoder)	ToA	**0.364**	**0.445**	**0.492**	**0.651**	**0.785**	**0.441**	**0.344**	**0.417**	0.535	**0.632**	**0.791**	**0.866**	**0.5**	**0.355**
AntennaPod	BM25	None	0.033	0.033	0.067	0.083	0.117	0.054	0.013	0.05	0.117	0.183	0.367	0.5	0.132	0.036
	TF-IDF	None	0.016	0.066	0.066	0.083	0.1	0.045	0.014	0.033	0.116	0.233	0.3	0.45	0.111	0.023
	CodeMatcher	None	0.033	0.1	0.15	0.183	0.25	0.062	0.005	0.05	0.15	0.2	0.35	0.567	0.156	0.061
	BLUiR	None	0.016	0.033	0.05	0.1	0.133	0.038	0.009	0.083	0.183	0.3	0.383	0.516	0.181	0.076
	FLIM	ToS	0.009	0.027	0.137	0.201	0.431	0.067	0.063	0.036	0.091	0.284	0.467	0.66	0.14	0.073
	CodeBERT	ToS	0.083	0.2	0.217	0.317	0.4	0.162	0.08	0.133	0.35	0.433	0.483	0.617	0.257	0.136
	GraphCodeBERT	ToS	0.183	**0.333**	**0.417**	0.483	0.55	0.258	0.111	**0.317**	**0.45**	**0.617**	0.683	0.75	0.394	0.204
	CodeLlama	ToS	0.172	0.321	0.403	0.49	0.523	0.237	0.137	0.312	0.436	0.602	0.687	0.722	0.392	0.198
	YourCoLo(GraphCodeBERT)	ToA	0.166	0.266	0.333	0.383	0.5	0.237	0.123	0.293	0.416	0.516	0.653	0.716	0.382	0.175
	YourCoLo(UniXcoder)	ToA	**0.216**	0.3	0.4	**0.583**	**0.683**	**0.311**	**0.192**	0.3	0.416	0.516	**0.716**	**0.816**	**0.403**	**0.255**
Cgeo	BM25	None	0.015	0.06	0.09	0.09	0.181	0.056	0.018	0.045	0.121	0.167	0.242	0.409	0.112	0.02
	TF-IDF	None	0.015	0.045	0.075	0.121	0.151	0.048	0.016	0.09	0.136	0.196	0.303	0.424	0.15	0.06
	CodeMatcher	None	0.045	0.06	0.121	0.151	0.257	0.076	0.011	0.121	0.212	0.333	0.393	0.454	0.174	0.081
	BLUiR	None	0.015	0.045	0.045	0.06	0.075	0.036	0.01	0.075	0.136	0.212	0.287	0.53	0.156	0.073
	FLIM	ToS	0.03	0.03	0.03	0.045	0.106	0.036	0.029	0.03	0.06	0.166	0.363	0.56	0.106	0.098
	CodeBERT	ToS	0.076	0.182	0.227	0.228	0.379	0.125	0.09	0.091	0.197	0.242	0.379	0.606	0.15	0.115
	GraphCodeBERT	ToS	0.152	0.273	0.333	0.439	0.561	0.224	0.129	0.212	0.379	0.485	0.682	0.788	0.306	0.192
	CodeLlama	ToS	0.149	0.273	0.301	0.43	0.589	0.215	0.132	0.21	0.385	0.458	0.681	0.789	0.299	0.187
	YourCoLo(GraphCodeBERT)	ToA	0.106	0.257	0.257	0.378	0.469	0.203	0.124	0.181	0.378	0.439	0.56	0.742	0.303	0.136
	YourCoLo(UniXcoder)	ToA	**0.197**	**0.333**	**0.454**	**0.56**	**0.667**	**0.311**	**0.179**	**0.272**	**0.5**	**0.651**	**0.727**	**0.863**	**0.419**	**0.263**
Anki	BM25	None	0.024	0.032	0.04	0.065	0.098	0.042	0.014	0.057	0.106	0.163	0.303	0.459	0.122	0.058
	TF-IDF	None	0.016	0.032	0.049	0.065	0.106	0.039	0.014	0.049	0.081	0.147	0.327	0.557	0.118	0.074
	CodeMatcher	None	0.032	0.122	0.147	0.213	0.237	0.082	0.016	0.065	0.27	0.385	0.557	0.59	0.18	0.093
	BLUiR	None	0.016	0.024	0.024	0.024	0.049	0.026	0.008	0.057	0.155	0.229	0.368	0.573	0.155	0.084
	FLIM	ToS	0.036	0.064	0.128	0.192	0.348	0.078	0.068	0.073	0.211	0.293	0.431	0.651	0.186	0.101
	CodeBERT	ToS	0.107	0.27	0.344	0.433	0.533	0.193	0.102	0.287	0.508	0.631	**0.787**	**0.869**	0.392	0.256
	GraphCodeBERT	ToS	0.189	**0.385**	**0.434**	**0.525**	0.598	0.274	0.121	0.295	0.541	0.607	0.77	0.861	0.396	0.213
	CodeLlama	ToS	0.179	0.381	0.421	0.502	0.588	0.257	0.156	0.285	0.523	0.602	0.732	0.865	0.396	0.197
	YourCoLo(GraphCodeBERT)	ToA	0.147	0.237	0.27	0.36	0.467	0.219	0.106	0.237	0.5	0.565	0.68	0.79	0.375	0.198
	YourCoLo(UniXcoder)	ToA	**0.279**	0.368	**0.434**	0.516	**0.622**	**0.359**	**0.196**	**0.377**	**0.565**	**0.663**	0.778	0.86	**0.491**	**0.277**
Termux	BM25	None	0.045	0.11	0.201	0.293	0.513	0.131	0.102	0.09	0.284	0.44	0.66	0.89	0.242	0.124
	TF-IDF	None	0.036	0.110	0.174	0.311	0.513	0.121	0.083	0.174	0.302	0.44	0.706	0.899	0.302	0.203
	CodeMatcher	None	0.201	0.339	0.422	0.522	0.541	0.28	0.09	0.247	0.495	0.651	0.743	0.743	0.362	0.175
	BLUiR	None	0.027	0.082	0.119	0.174	0.247	0.085	0.063	0.128	0.266	0.422	0.651	0.834	0.269	0.157
	FLIM	ToS	0.146	0.247	0.293	0.385	0.495	0.217	0.17	0.165	0.33	0.422	0.541	0.67	0.278	0.183
	CodeBERT	ToS	0.284	0.367	0.505	0.67	0.817	0.344	0.263	0.385	0.624	0.752	0.881	**0.982**	0.446	0.29
	GraphCodeBERT	ToS	0.376	0.477	0.56	0.688	0.807	0.433	0.32	0.477	0.697	0.789	0.908	0.963	0.559	0.422
	CodeLlama	ToS	0.368	0.472	0.558	0.682	0.808	0.429	0.31	0.473	0.689	0.787	0.906	0.958	0.551	0.397
	YourCoLo(GraphCodeBERT)	ToA	0.238	0.339	0.376	0.467	0.614	0.323	0.253	0.348	0.559	0.651	0.816	0.926	0.48	0.294
	YourCoLo(UniXcoder)	ToA	**0.422**	**0.532**	**0.651**	**0.752**	**0.825**	**0.52**	**0.443**	**0.532**	**0.724**	**0.798**	**0.926**	0.981	**0.631**	**0.49**

Bold numbers indicate the best performance for each metric in the table.

AST-based structural information with BM25 scoring, performs worse than BM25 and TF-IDF in our experiments. Also originating from the field of bug localization, the FLIM method achieves better performance than BLUiR in the UR based code localization task, indicating that combining DL techniques with traditional IR methods is more effective than relying solely on traditional IR. Although FLIM uses CodeBERT as a feature extractor, its performance is inferior to directly fine-tuning CodeBERT. This is because FLIM's learning-to-rank model requires a lot of high-quality relevance labels, and the amount of data available for UR-based code localization is relatively limited compared to the datasets typically employed by FLIM in bug localization tasks, which may hinder the model's ability to learn stable ranking patterns. In contrast, end-to-end fine-tuning can better leverage pre-trained weights to effectively adapt to limited data. Our proposed approach, YourCoLo, achieves the best performance across all methods and outperforms the previous SOTA approach, in which GraphCodeBERT was fine-tuned on a single project in earlier benchmarks. This solves the performance

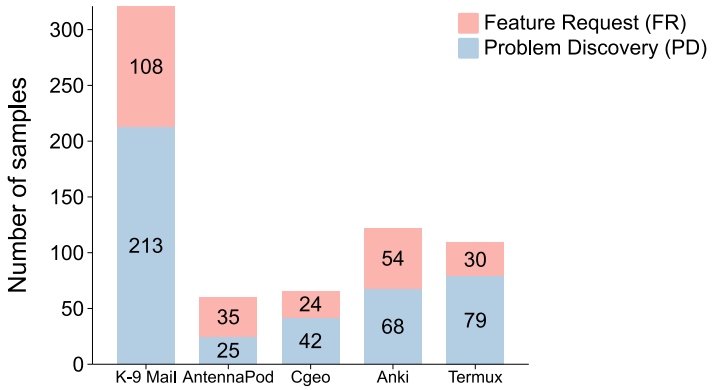


Fig. 12. The number of PD-type and FR-type reviews in its test dataset for each app project.

reduction problem on multiple projects seen in previous models. Compared to earlier fine-tuning methods, YourCoLo can fine-tune a universal model that is applicable to multiple projects.

In order to further analyze the model's performance, we evaluated its performance separately on the two major types of URs in the Review2Code dataset: PD and FR. Specifically, the reviews in the test sets of the five app projects were manually classified into two sub-test sets according to their type (PD or FR). We then compiled a summary chart, as shown in Figure 12, presenting for each app project the number of PD-type and FR-type reviews contained in its test set. Then our method was tested on the PD and FR sub-test sets for each app project, and the corresponding experimental results were recorded.

As shown in Table 3, at the method level, our model achieves higher MRR and MAP on PD-type reviews in projects such as *Cgeo* and *Anki*, whereas in projects like *K-9 Mail*, *AntennaPod*, and *Termux*, FR-type reviews achieve higher MRR. To summarize the overall trend across projects, we computed the weighted average of the metrics across the five app projects, using the number of reviews in each project as the weight. At the method level, the weighted averages show that our model performs slightly better on PD-type reviews (MRR = 0.425, MAP = 0.319) than on FR-type reviews (MRR = 0.401, MAP = 0.277), likely because PD-type reviews include concrete problem descriptions and error reproduction details. At the file level, the performance gap is smaller, with PD-type reviews achieving MRR = 0.512 and MAP = 0.358, and FR-type reviews achieving MRR = 0.491 and MAP = 0.322. These results indicate that our model is robust at coarser granularity and can effectively localize code files based on both review types. Overall, these results demonstrate that our method can handle both PD and FR reviews. PD-type reviews provide slightly more useful information for method-level localization, while FR-type reviews are still effectively leveraged, particularly at the file level.

In the experimental results, YourCoLo outperforms the SOTA approach. However, numerical improvement alone does not guarantee that the observed gain is statistically meaningful rather than due to randomness. As shown in Table 4, to assess whether the improvement of YourCoLo over the SOTA method in terms of MRR is statistically significant, we conducted a non-parametric significance test based on the 678 URs in the test set. For each review, we computed the MRR scores produced by YourCoLo and the SOTA method, resulting in paired samples for comparison. We formulated the null hypothesis (H0) that there is no significant difference in MRR scores between the two methods, against the alternative hypothesis (H1) that a significant difference exists. We first applied the Shapiro–Wilk test on the distribution of per-review MRR differences (i.e., YourCoLo MRR minus SOTA MRR) to check for normality; the test indicated that the differences are not normally distributed. Therefore, we employed the Wilcoxon signed-rank test, which does not assume

Table 3. Performance of YourCoLo on Two Types of URs: PD and FR

APP Name	Review Type	Method Level							File Level						
		Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
K-9 Mail	PD	0.352	0.436	**0.497**	**0.657**	**0.802**	0.433	**0.348**	0.408	0.535	**0.638**	**0.807**	**0.877**	0.49	**0.358**
	FR	**0.388**	**0.462**	0.481	0.638	0.75	**0.455**	0.336	**0.435**	**0.537**	0.62	0.759	0.842	**0.517**	0.35
AntennaPod	PD	0.16	0.24	0.36	0.4	0.52	0.258	0.177	0.28	**0.44**	0.44	0.56	**0.76**	0.385	0.25
	FR	0.257	**0.342**	**0.457**	**0.514**	**0.6**	**0.35**	**0.2**	**0.314**	0.4	**0.514**	**0.714**	0.743	**0.41**	**0.258**
Cgeo	PD	0.261	**0.38**	**0.5**	**0.642**	**0.809**	**0.377**	**0.213**	**0.357**	**0.523**	**0.667**	**0.785**	**0.904**	**0.482**	**0.314**
	FR	0.083	0.25	0.375	0.416	0.416	0.196	0.122	0.125	0.458	0.625	0.625	0.792	0.306	0.173
Anki	PD	0.294	**0.441**	**0.529**	**0.573**	**0.676**	**0.395**	**0.201**	**0.382**	**0.573**	**0.691**	**0.779**	0.838	**0.52**	**0.296**
	FR	0.222	0.333	0.389	0.518	0.592	0.315	0.187	0.351	0.518	0.592	0.759	**0.907**	0.457	0.252
Termux	PD	0.379	0.569	**0.696**	**0.797**	**0.848**	0.509	**0.444**	0.518	0.746	**0.784**	**0.898**	0.962	0.621	0.47
	FR	0.5	**0.6**	0.667	0.7	0.8	**0.581**	0.442	**0.633**	**0.767**	0.767	0.833	**1**	**0.7**	**0.543**
Weighted Average	PD	0.328	**0.444**	**0.531**	**0.653**	**0.775**	**0.425**	**0.319**	**0.412**	**0.573**	**0.665**	**0.803**	**0.882**	**0.512**	**0.358**
	FR	0.318	0.414	0.470	0.581	0.669	0.401	0.277	0.394	0.534	0.617	0.749	0.856	0.491	0.322

Bold numbers indicate the best performance for each metric in the table.

Table 4. Statistical Test Results between YourCoLo and the SOTA Approach

Evaluation Metric	**MRR**	**MAP**
Test Sample Size	678	678
Normality Test	Shapiro-Wilk	Shapiro-Wilk
Normality Result	Not normal	Not normal
Significance Test	Wilcoxon signed-rank	Wilcoxon signed-rank
Wilcoxon Statistic	51,059.5	60,653.5
p-value	<0.0001	<0.0001

normality and is suitable for paired samples. The test results demonstrate that the improvement of YourCoLo over the SOTA method is statistically significant, with a Wilcoxon W statistic of 51,059.5 and a p-value less than 0.0001. Similarly, to examine whether the improvement in terms of MAP is statistically significant, we repeated the same procedure. We computed the per-review MAP scores for both methods, formulated the same null and alternative hypotheses, and applied the Shapiro–Wilk test on the distribution of MAP differences. The test similarly indicated that the differences are not normally distributed. Consequently, we also used the Wilcoxon signed-rank test for MAP, which showed that the improvement of YourCoLo over the SOTA method is statistically significant, with a Wilcoxon W statistic of 60,653.5 and a p-value less than 0.0001.

The functions of the YourCoLo approach differ between the UniXcoder and GraphCodeBERT models. Using UniXcoder as the backbone, YourCoLo(UniXcoder) achieves the best performance for code localization. In contrast, YourCoLo(GraphCodeBERT), which uses GraphCodeBERT as the backbone, does not perform as effectively. This suggests that our proposed YourCoLo approach is particularly well suited for the UniXcoder model. UniXcoder emphasizes joint modeling of code and natural language, effectively handling both types of data during the pre-training phase. This capability allows it to better utilize contextual information when understanding code [18]. On the other hand, GraphCodeBERT focuses mainly on representing code with graph structures, integrating syntactic and semantic information to capture structured code features [19].

We use prompts to tag projects so that the model can recognize the features of different projects as they fine-tune, allowing for more efficient training on cross-project datasets. In doing this, the prompt acts as a project identifier to help the model recognize the relationship between URs and code snippets across projects. The model needs to focus on the prompt to some extent to differentiate

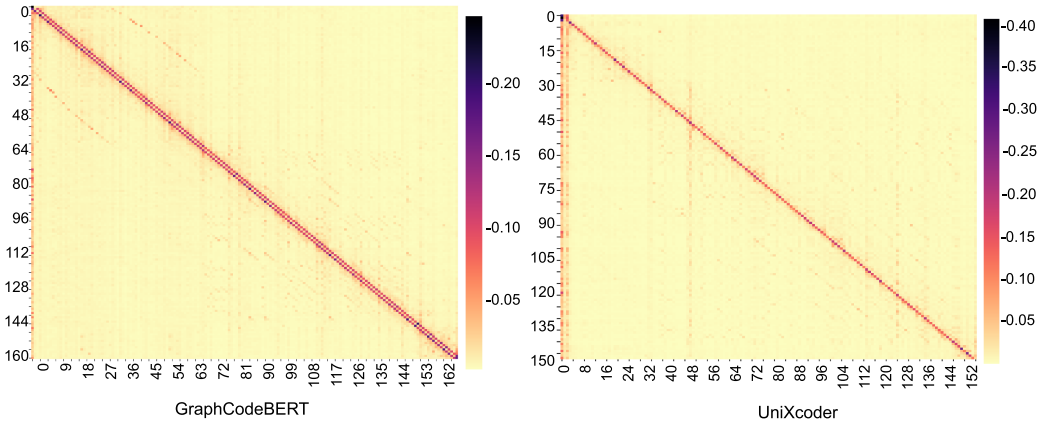


Fig. 13. The attention heat distribution of GraphCodeBERT and UniXcoder for the code snippet with the added prompt.

between the features of different projects. Both models are based on the Transformer architecture and rely on the attention mechanism [76], with their primary task still being to match URs with code content. Therefore, most of the attention should be focused on the association between the review and code content.

In order to further explore the different focus and use of the prompt by GraphCodeBERT and UniXcoder, we show in Figure 13 the attention heat distribution of GraphCodeBERT and UniXcoder for the code snippet with the added prompt. For GraphCodeBERT, there are two adjacent diagonal lines of attention, indicating that GraphCodeBERT is primarily focused on the relationship between each token and its neighboring tokens. In the upper left corner, we can see that on each side of the main diagonal, there is a shorter diagonal line of attention, which reflects GraphCodeBERT's focus on the added prompt. This shows that GraphCodeBERT allocates some of its attention to the prompt information, suggesting that adding the prompt affects the original distribution of the attention weights. This interference disrupts the main task of aligning URs with code snippets, resulting in less effective code localization. In contrast, UniXcoder integrates the prompt more seamlessly into the model's input without altering the original attention distribution. As a project-specific context, the prompt aids the model in identifying the relationship between URs and code snippets across different projects, enhancing the model's cross-project generalization ability.

Summary. Our proposed approach YourCoLo (with UniXcoder as the backbone) significantly outperforms current baselines, especially in achieving superior localization results across multiple application projects without having to fine-tune each project individually. Specifically, YourCoLo outperforms the current SOTA method by 7.7% and 7.0% in MRR, and by 8.8% and 6.8% in MAP, at the method and file levels, respectively.

5.2 RQ2: How Much Do Different Components Contribute to the Code Localization Performance?

In this section, we design experiments to verify the improvement in task performance of each part of YourCoLo. In this experiment, we use the UniXcoder model as the backbone of YourCoLo.

5.2.1 *How Much Does CPFT Contribute to the Code Localization Performance?*

Method. In this section, we conduct comparative experiments involving five different training schemes to evaluate the performance of our proposed CPFT method, which are shown in Table 5.

Table 5. Comparison of Five Different Training Schemes to Investigate the Impact of CPFT on Method-Level and File-Level Code Localization Performance

Method	Fine-Tuning Scope	Prompt Augmentation	Training Method
ToS_GraphCodeBERT	Single project	No	Individual model for each project
ToA_GraphCodeBERT	All projects	No	Universal model for all projects
ToS_CPFT_w/o_prompt	Single project	No	Individual model for each project
ToA_CPFT_w/o_prompt	All projects	No	Universal model for all projects
ToA_CPFT	All projects	Yes	Universal model for all projects

Table 6. Decomposition of the Proposed Prompt Template and Its Design Rationale, Detailing Each Part's Content and Purpose

Part	Content	Purpose
partial_1	This code belongs to the + project_name + project.	Provide project background information as context for the model
partial_2	The user's review and the corresponding code must have the same meaning to match.	Explicitly inform the model that the goal is semantic matching
partial_3	so the code + code_content + can be represented as	Guide the model to perform vectorization/representation learning

For the column "Prompt Augmentation," "No" means that the training sample is not augmented using a prompt, while "Yes" indicates that the training sample is augmented using a prompt, as illustrated in Figure 4. "Partial prompt" means that only the initial part of the prompt in Figure 4 is used, which indicates the project to which the review or code snippet belongs. For example, in the case of the *Cgeo* project, the prompt is: "*This user review/code belongs to the Cgeo project.*" It should be noted that the first two training schemes in Table 5 are designed to enrich the experimental results and to demonstrate that using CPFT is more suitable for fine-tuning a universal model that works for most projects. The latter three training schemes in Table 5 aim to explore how much does CPFT contribute to the code localization performance. *ToS_CPFT_w/o_prompt* and *ToA_CPFT_w/o_prompt* are designed to explore the differences between fine-tuning using data from a single project and fine-tuning using data from all projects. *ToA_CPFT_w/o_prompt* and *ToA_CPFT* are designed to investigate the effects of fine-tuning using all project data, with a focus on the influence of prompt template. In the first two schemes listed in Table 5, the same Cross-Entropy loss function and hyperparameter settings are applied, and the LCPS method is not applied during the inference process. In the last three schemes listed in Table 5, the same CEAPS loss function and hyperparameter settings are used, and the LCPS method is not applied during the inference process.

To investigate the impact of different components of our designed prompt template on model performance, we decomposed the complete prompt template into three distinct parts as Table 6. We then conducted an ablation study by systematically removing each part to observe the effect on retrieval performance. Specifically, we compared the full prompt template (partial_1 + partial_2 + partial_3) against several ablated versions (e.g., removing partial_1, partial_2, or partial_3 individually). The same CEAPS loss function and hyperparameter settings are used, and the LCPS method is not applied during the inference process. The performance was measured using standard evaluation metrics, including MRR, MAP, and Top-K. This design allows us to analyze the contribution of each prompt component to the final results.

Results. The evaluation results of the five methods on the test datasets of the five projects are shown in Table 7. At the method level, *ToA_CPFT* performed best overall, outperforming other fine-tuning methods with respect to Top-1, Top-3, MRR, and MAP. *ToS_CPFT_w/o_prompt*

Table 7. Code Localization Performance on Five Applications under Different Training Schemes at the Method and File Levels

Training Scheme	Method Level							File Level						
	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
ToS_GraphCodeBERT	0.238	0.378	0.446	0.543	0.643	0.312	0.183	0.327	0.521	0.615	**0.755**	0.839	0.418	0.279
ToA_GraphCodeBERT	0.226	0.333	0.401	0.503	0.605	0.291	0.167	0.334	0.502	0.609	0.737	0.833	0.418	0.278
ToS_CPFT_w/o_prompt	0.256	0.392	**0.470**	**0.555**	**0.648**	0.338	0.223	0.339	**0.543**	**0.640**	0.737	0.836	0.439	0.274
ToA_CPFT_w/o_prompt	0.221	0.368	0.430	0.517	0.599	0.312	0.201	0.318	0.535	0.624	0.727	0.827	0.431	0.256
ToA_CPFT	**0.296**	**0.396**	0.448	0.540	0.623	**0.376**	**0.251**	**0.380**	0.542	0.632	0.740	**0.840**	**0.484**	**0.312**

Bold numbers indicate the best performance for each metric in the table.

Table 8. Ablation Experiments to Assess the Impact of Individual Prompt Components

Training Scheme	Method Level							File Level						
	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
ToA_CPFT	**0.296**	**0.396**	**0.448**	**0.540**	**0.623**	**0.376**	**0.251**	**0.380**	**0.542**	**0.632**	**0.740**	**0.840**	**0.484**	**0.312**
ToA_CPFT_w/o_prompt	0.221	0.368	0.430	0.517	0.599	0.312	0.201	0.318	0.535	0.624	0.727	0.827	0.431	0.256
ToA_CPFT_w/o_prompt_partial_1	0.230	0.310	0.450	0.470	0.590	0.328	0.218	0.292	0.519	0.613	0.717	0.835	0.438	0.244
ToA_CPFT_w/o_prompt_partial_2	0.247	0.353	0.425	0.503	0.608	0.347	0.235	0.321	0.524	0.619	0.737	0.832	0.444	0.253
ToA_CPFT_w/o_prompt_partial_3	0.275	0.373	0.442	0.520	0.650	0.364	0.243	0.373	0.531	0.620	0.738	0.840	0.473	0.278

Bold numbers indicate the best performance for each metric in the table.

performed better than *ToS_GraphCodeBERT* and *ToA_GraphCodeBERT* without prompt, but worse than *ToA_CPFT* with prompt. *ToA_CPFT* performed better than *ToS_CPFT_w/o_prompt*. At the file level, *ToA_CPFT* has significant advantages over other methods at Top-1, Top-3, Top-5, MRR, and MAP, indicating that prompt-enhanced cross-project models are effective not only at the method level but also at larger granularity (file level).

As shown in Table 8, at the method level, removing either partial_1 or partial_2 leads to a significant decrease in both MRR and MAP metrics, whereas removing partial_3 results in a relatively small decline. This indicates that the third part has a comparatively minor impact. Similarly, at the file level, removing partial_1 or partial_2 also causes a notable drop in MRR and MAP scores. These results suggest that different parts of the prompt contribute differently, with partial_1 and partial_2 being the most critical components.

Discussion. When fine-tuning a universal model on data from all projects without using prompt, the model's performance decreases compared to fine-tuning a separate model for each project using data from a single project (also without using prompt). We also find that utilizing the full prompt template yields the best performance, whereas using only a partial prompt template leads to a performance decline. These results indicate that different parts of the prompt contribute unevenly, with partial_1 and partial_2 being the most critical components. This suggests that our designed prompt template is well-structured and complete. These findings demonstrate the effectiveness of the CPFT method, as fine-tuning using prompt augmentation data from all projects leads to improved performance.

5.2.2 *How Much Does CEAPS Contribute to the Code Localization Performance?*

Method. In this section, we compare *CESPS* and *CEAPS*. *CESPS* refers to the original cross-entropy loss function, which calculates the loss based on a single positive sample. *CEAPS* is our newly designed loss function that leverages the probability distribution of all positive samples. For these experiments, prompts are added to enhance the data and fine-tune a model using all project data.

Table 9. Code Localization Results on Five Apps Comparing Two Loss Calculation Methods, CEAPS and CESPS, Evaluated at Method Level and File Level

APP Name	Loss Type	Method Level							File Level						
		Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
K-9 Mail	CESPS	0.345	0.442	**0.492**	**0.582**	0.654	0.421	0.298	0.383	**0.538**	**0.623**	**0.728**	0.797	0.477	0.327
	CEAPS	**0.364**	**0.445**	0.485	0.579	**0.66**	**0.431**	**0.316**	**0.417**	0.532	0.607	0.722	**0.8**	**0.494**	**0.343**
AntennaPod	CESPS	0.186	**0.321**	**0.38**	**0.452**	0.537	0.263	0.156	**0.31**	0.402	0.512	0.647	0.728	0.381	0.215
	CEAPS	**0.216**	0.3	0.333	0.433	**0.55**	**0.29**	**0.179**	0.3	**0.416**	**0.533**	**0.65**	**0.733**	**0.396**	**0.234**
Cgeo	CESPS	0.136	0.227	0.287	0.348	0.439	0.212	0.135	0.181	0.348	0.454	0.545	0.727	0.3	0.198
	CEAPS	**0.197**	**0.333**	**0.409**	**0.485**	**0.545**	**0.295**	**0.171**	**0.272**	**0.5**	**0.636**	**0.697**	**0.848**	**0.417**	**0.254**
Anki	CESPS	0.195	0.375	**0.45**	**0.535**	**0.597**	0.279	0.162	0.316	0.538	0.601	0.71	**0.856**	0.375	0.219
	CEAPS	**0.279**	0.369	0.434	0.508	0.59	**0.357**	**0.172**	**0.377**	**0.549**	**0.622**	**0.713**	0.836	**0.483**	**0.263**
Termux	CESPS	0.377	0.487	0.55	0.67	**0.82**	0.442	0.376	0.487	0.7	0.752	0.91	0.961	0.578	0.456
	CEAPS	**0.422**	**0.532**	**0.577**	**0.697**	0.77	**0.505**	**0.419**	**0.532**	**0.715**	**0.761**	**0.917**	**0.972**	**0.628**	**0.467**

Bold numbers indicate the best performance for each metric in the table.

All other experimental settings are kept constant, with the only difference being the loss calculation method. The LCPS method is not used during the inference process.

Results. The experimental results are shown in Table 9. At the method level, MRR improves with the use of *CEAPS*. For most applications, *CEAPS's* Top-1, Top-3, and Top-5 accuracies are better than or equal to those of *CESPS* in most cases. Especially in *K-9 Mail*, *Cgeo*, and *Termux*, *CEAPS* showed significant improvements in Top-1 and Top-3. At the file level, MRR improved with the use of *CEAPS*, *K-9 Mail* from 0.477 to 0.494, and *Termux* from 0.578 to 0.628. MAP improved with the use of *CEAPS*, *Termux* from 0.376 to 0.419. Overall, *CEAPS* achieved the best results for almost all apps in MRR, MAP, and Top-1. *CEAPS* outperforms *CESPS* in most cases, reflecting its advantage in handling one-to-many relationships between URs and code snippets.

Discussion. By calculating loss based on the probability distribution of all positive samples, *CEAPS* can better capture general matching information, thereby enhancing the model's localization capability.

5.2.3 How Much Does LCPS Contribute to the Code Localization Performance?

Method. In this section, we re-rank the matched code snippets with and without the LCPS method, where LCPS leverages coupling relationships to optimize the re-ranking of initially matched code snippets. In the experiment, the loss function used in the fine-tuning stage, the training data, and other experimental settings are the same for both approaches; the only difference is that one uses re-ranking, while the other does not.

— *LCPS*: Re-rank the matched code snippets with the LCPS method.
— *w/o_LCPS*: No re-ranking method is used.

Results. The experimental results are shown in Table 10. At the method level, the results show that the *LCPS* method has a certain degree of improvement compared with *w/o_LCPS* in the Top-5, Top-10, and Top-20 indexes, while *K-9 Mail* and *Termux* show a relatively significant improvement: *LCPS's* performance on the Top-10 and Top-20 improved from 0.579 to 0.651 and from 0.66 to 0.785, respectively. At the file level, *LCPS's* performance is significantly improved compared with *w/o_LCPS*, especially in Top-10 and Top-20, and *Termux* and *K-9 Mail*'s re-ranking effect is particularly prominent. *K-9 Mail* increased from 0.722 to 0.791 in Top-10. The re-ranking method *LCPS* has limited improvement in Top-1 and Top-3, and according to the characteristics of the method, the local coupling is improved while keeping the original sorting order basically

Table 10. Code Localization Performance on Five Apps Comparing Results with LCPS Re-Ranking and without Re-Ranking

APP Name	Re-Rank	Method Level							File Level						
		Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP	Top-1	Top-3	Top-5	Top-10	Top-20	MRR	MAP
K-9 Mail	LCPS	0.364	0.445	**0.492**	**0.651**	**0.785**	**0.441**	**0.344**	0.417	**0.535**	**0.632**	**0.791**	**0.866**	**0.5**	**0.355**
	w/o_LCPS	0.364	0.445	0.485	0.579	0.66	0.431	0.316	0.417	0.532	0.607	0.722	0.8	0.494	0.343
AntennaPod	LCPS	0.216	0.3	**0.4**	**0.583**	**0.683**	**0.311**	**0.192**	0.3	0.416	0.516	**0.716**	**0.816**	**0.403**	**0.255**
	w/o_LCPS	0.216	0.3	0.333	0.433	0.55	0.29	0.179	0.3	0.416	**0.533**	0.65	0.733	0.396	0.234
Cgeo	LCPS	0.197	0.333	**0.454**	**0.56**	**0.667**	**0.311**	**0.179**	0.272	0.5	**0.651**	**0.727**	**0.863**	**0.419**	**0.263**
	w/o_LCPS	0.197	0.333	0.409	0.485	0.545	0.295	0.171	0.272	0.5	0.636	0.697	0.848	0.417	0.254
Anki	LCPS	0.279	0.368	0.434	**0.516**	**0.622**	**0.359**	**0.196**	0.377	**0.565**	**0.663**	**0.778**	**0.86**	**0.491**	**0.277**
	w/o_LCPS	0.279	**0.369**	0.434	0.508	0.59	0.357	0.172	0.377	0.549	0.622	0.713	0.836	0.483	0.263
Termux	LCPS	0.422	0.532	**0.651**	**0.752**	**0.825**	**0.52**	**0.443**	0.532	**0.724**	**0.798**	**0.926**	**0.981**	**0.631**	**0.49**
	w/o_LCPS	0.422	0.532	0.577	0.697	0.77	0.505	0.419	0.532	0.715	0.761	0.917	0.972	0.628	0.467

Bold numbers indicate the best performance for each metric in the table.

unchanged. Second, *LCPS* can improve the overall MRR, especially at the file level. *LCPS* can improve the MAP at the method level.

Discussion. The re-ranking method LCPS has limited improvement in Top-1 and Top-3, and according to the characteristics of the method, the local coupling is improved while keeping the original sorting order basically unchanged. Second, LCPS can improve the overall MRR, especially at the file level. LCPS demonstrates that using the historical commit frequency between code snippets as a measure of code coupling to assist in re-ranking can improve code localization results.

Summary. Experimental results demonstrate that the CPFT method addresses cross-project generalization issues, the CEAPS method excels in handling one-to-many mapping relationships, and the LCPS method enhances code localization performance. Each method proves to be effective in its respective aspect.

5.3 RQ3: What Is the Impact of Different Hyperparameters?

Method. In this section, we study the impact of the hyperparameter: coupling window K. In the experimental process, we use the UniXcoder model as the backbone of YourCoLo, as it yielded the best results, and set different K values for the experimental analysis to observe changes in various indicators of code localization under different K values.

Results. As shown in Figures 14 and 15, the MRR and MAP metrics show that the optimal values of K for most projects are between 3 and 5 at the method and file levels. The value of K has a greater impact on *Termux* and *K-9 Mail*, but less so on *Anki*. For Top-1, the value of K has little effect, as the code snippet ranked first remains unchanged in the final localization results, regardless of the value of K. For Top-3, *K-9 Mail* and *Anki* perform stably across the values of K, while other projects peak at $K = 2$. As K increases, the effect decreases for Top-3 due to the fixed first three elements. For Top-5, performance varies by project, with *K-9 Mail* and *Termux* peaking at $K = 2$ and *AntennaPod* at $K = 3$. At the file level, most projects perform best at $K = 4$ or 5. For Top-10, the K value significantly impacts performance at the method level, with a more stable improvement at the file level. For Top-20, the performance of the method level improves with K up to 5, then stabilizes or decreases, while the performance of the file level shows continuous improvement with higher values of K.

Discussion. The coupling information of some app projects is concentrated in a small range of values K, which shows the effectiveness of local coupling. The file level, on the other hand, is more dependent on a larger K value, indicating that connections between files usually require a

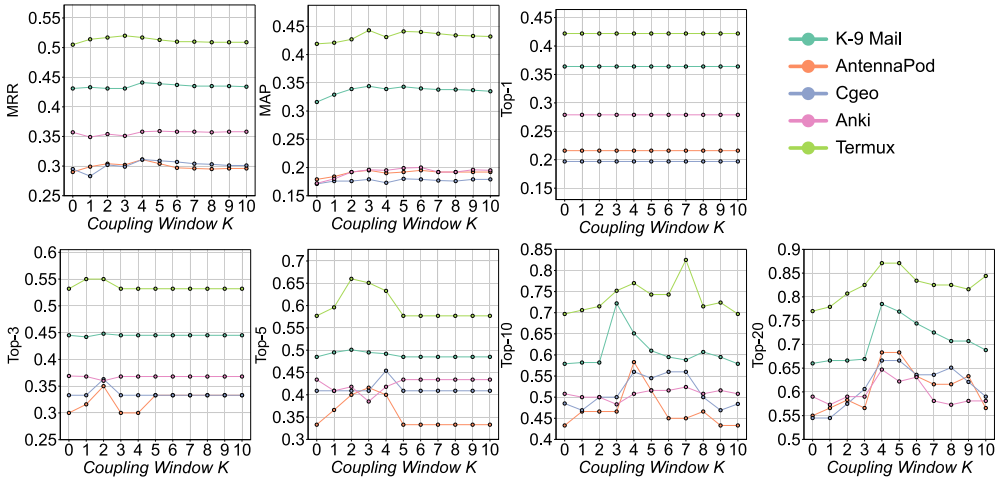


Fig. 14. The effect of coupling window K value on the result of method-level code localization.

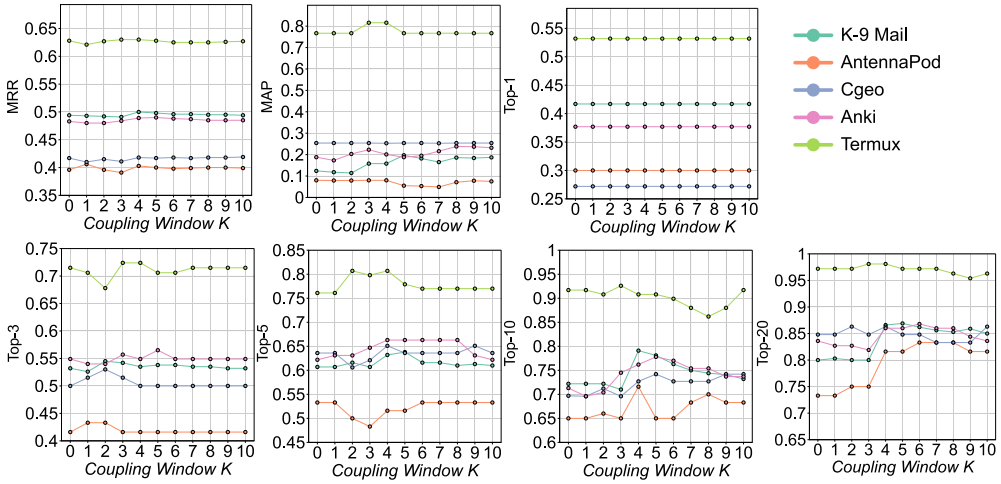


Fig. 15. The effect of coupling window K value on the result of file-level code localization.

larger window to capture. The design of the coupling window should be adjusted according to the characteristics of the project. For projects with a strong coupling relationship, adjusting K can bring about a large improvement, while for projects with a weak coupling relationship, increasing K can have a limited effect on the results.

Summary. The optimal value of K varies for each app project, and even within the same project, the best K value differs between method-level and file-level code localization. Most app projects achieve the best code localization results when the value of K is in the range of 3 to 5.

5.4 Case Study

We present some cases to demonstrate the effectiveness of our proposed YourCoLo approach. If the retrieved method belongs to the ground truth, it is highlighted in green.

The first case study focuses on the comparison of code localization capabilities among models at the method level. As shown in Figure 16, CodeBERT and CodeLlama perform comparably, each locating three correct methods. Although GraphCodeBERT performs better, correctly locating

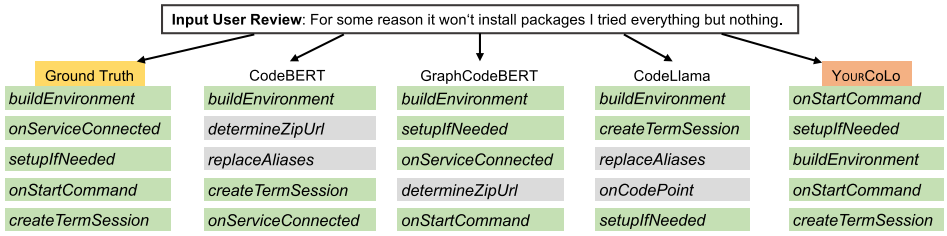


Fig. 16. A case study: the comparison of code localization capabilities among models at the method level.

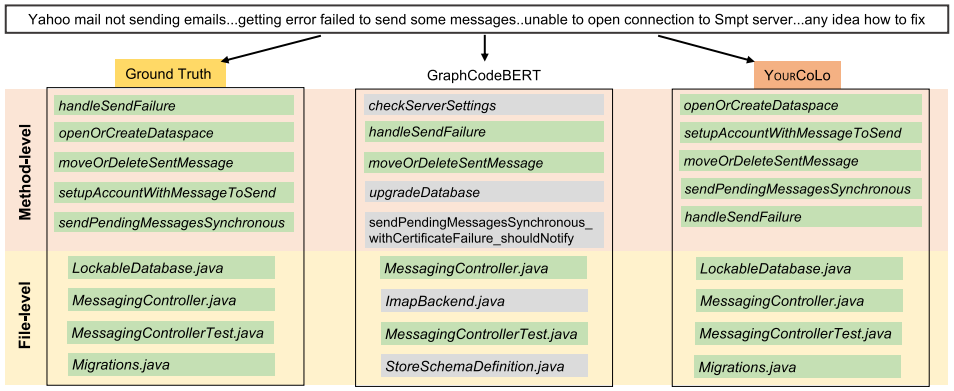


Fig. 17. A case study: the comparison between GraphCodeBERT and YourCoLo at method level and file level.

four target methods (*buildEnvironment*, *setupIfNeeded*, *onServiceConnected*, *onStartCommand*), A non-target method, *determineZipUrl*, is incorrectly located. YourCoLo performs best, accurately locating five code methods. In this case, YourCoLo demonstrates a stronger semantic understanding of URs and code methods than other models.

The second case study focuses on the comparison between GraphCodeBERT and YourCoLo at the method level and the file level. As shown in Figure 17, in terms of method level, GraphCodeBERT successfully located some of the error methods, such as *handleSendFailure* and *moveOrDeleteSentMessage*. However, some prediction results such as *checkServerSettings*, *upgradeDatabasedb*, and *sendPendingMessagesSynchronous_withCertificateFailure_shouldNotify* are not in the ground truth. In contrast, YourCoLo performs better at the method level. It successfully locates all methods in the ground truth, demonstrating higher performance. In file-level localization, GraphCodeBERT successfully locates some of the relevant documents, such as *MessagingController.Java* and *MessagingControllerTest.Java*, but it also gives incorrect files, such as *ImapBackend.Java* and *StoreSchemaDefinition.Java*. YourCoLo also performs well at the file level, completely covering all relevant files in ground truth, showing its high accuracy in locating code at the file level.

The third case study focuses on demonstrating how YourCoLo can optimize the ranking of matched code snippets, improving their rank in the final results. As shown in Figure 18, the first five methods names in the original results do not appear in the ground truth, and the first code snippet that matches the UR is ranked 35th. Through optimization of YourCoLo, we identify the most relevant code snippets from the coupling dictionary for the top five snippets in the original results. The *addLatestLogs* method does not have a corresponding entry in the coupling dictionary. We insert the matched code snippets from the coupling dictionary into positions 6 through 9 of the original results. After re-ranking, the *doInBackgroundInternal* method is moved to the ninth position.

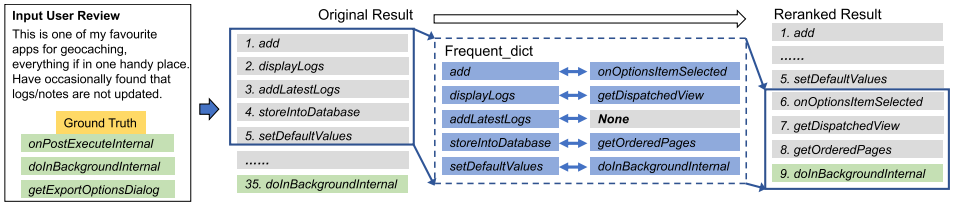


Fig. 18. A case study: YourCoLo can optimize the ranking of matched code snippets in the Cgeo app.

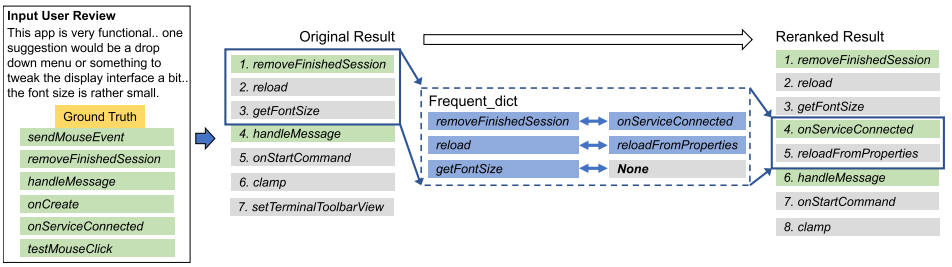


Fig. 19. A case study: YourCoLo can enhance the coupling of code localization results in the Termux app.

The fourth case study focuses on how YourCoLo can enhance the coupling of code localization results. As shown in Figure 19, only two of the top seven method names in the original results appear in the ground truth. Through optimization of YourCoLo, we identify the most relevant code snippets from the coupling dictionary for the top five snippets. The *getFontSize* method does not have a corresponding entry in the coupling dictionary. We insert the matched code snippets from the coupling dictionary into positions 4 and 5 of the original results. After re-ranking, three of the top seven method names in the final results appear in the ground truth.

As shown in Figure 20, we can see that both *app/src/main/java/com/termux/view/TerminalView.java* and *terminal-view/src/main/java/com/termux/view/TerminalView.java* contain the *onCreateInputConnection* method. The content of these two methods is identical, but they are located in different files. However, only the *onCreateInputConnection* method in *terminal-view/src/main/java/com/termux/view/TerminalView.java* is the ground-truth method. YourCoLo is confused by method snippets identical to the ground-truth method but originating from different files during similarity computation, causing the true ground-truth method to be ranked lower. Similarly, as shown in Figure 21, we can see that both *main/src/cgeo/geocaching/connector/ec/ECLogin.java* and *main/src/cgeo/geocaching/connector/gc/GCLogin.java* contain the *getLoginStatus* method. Although their implementations differ, both serve the function of detecting login status and are highly related functionally. However, only the *getLoginStatus* method in *main/src/cgeo/geocaching/connector/gc/GCLogin.java* is the ground-truth method. YourCoLo is also misled by method snippets that are functionally similar to the ground truth but come from different files. Our method still struggles to accurately distinguish the responsibilities of functionally similar or identical methods residing in different files, leading to interference in localization results.

As shown in Figure 22, we can see that the ground-truth methods are concentrated in two files (*core/src/main/java/de/danoeh/antennapod/core/service/playback/PlaybackService.java* and *core/src/main/java/de/danoeh/antennapod/core/feed/FeedMedia.java*). However, our method does not leverage file-level information for pre-filtering, resulting in a large number of candidate methods and reduced localization performance. This demonstrates that when ground-truth methods are concentrated in a few files, operating only at the method level can impair both ranking accuracy and computational efficiency.

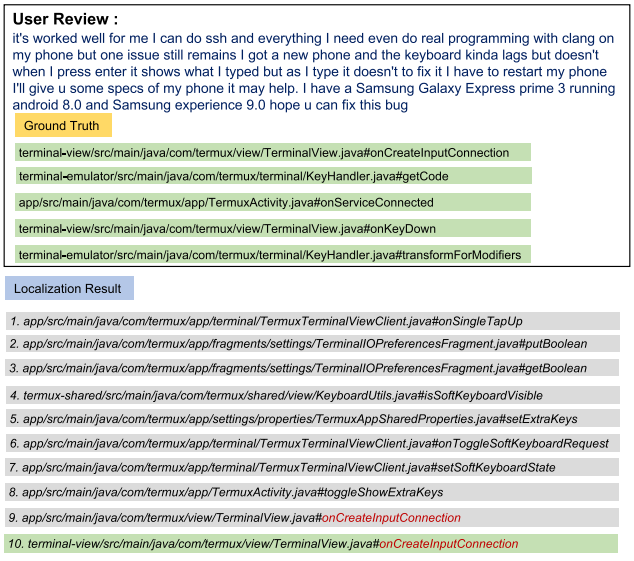


Fig. 20. A case study: YourCoLo is confused by method snippets that are identical to the ground-truth method but come from different files during similarity computation, causing the actual ground-truth method to be ranked lower.

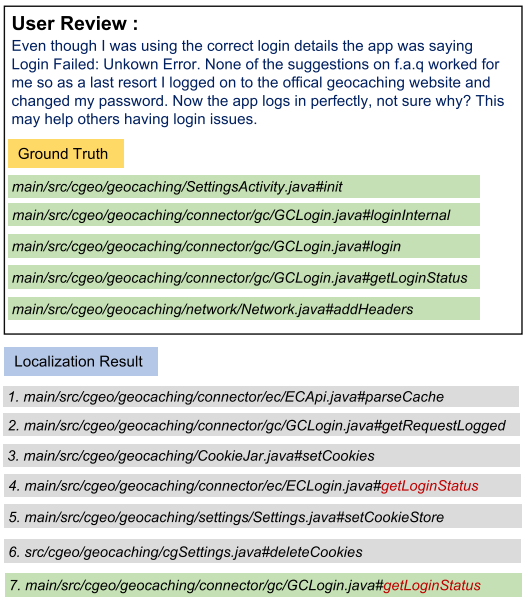


Fig. 21. A case study: YourCoLo is confused by method snippets that are functionally similar to the ground-truth method but come from different files during similarity computation, causing the actual ground-truth method to be ranked lower.

The three case studies above illustrate the importance of incorporating file-level semantic information and project module knowledge to narrow the scope of code localization, addressing both the challenge of distinguishing functionally similar or identical methods across different files and the concentration of most ground-truth methods in a small number of files. Building on



Fig. 22. A case study: YourCoLo method does not leverage file-level information for pre-filtering, resulting in an excessive number of candidate methods being considered, which affects the localization performance.

these insights, our future work will focus on leveraging file-level semantic information and project module knowledge to provide richer contextual cues about file and module affiliations. In addition, we plan to adopt a hierarchical localization strategy that first performs coarse-grained filtering at the file level to identify potentially relevant files, followed by fine-grained method-level localization within these files. By combining contextual information with this two-stage approach, we aim to more accurately differentiate methods based on their organizational roles within the project, reduce noise from irrelevant methods, improve computational efficiency, and significantly enhance the model's ability to localize code accurately in complex, modularized software systems.

6 Threats to Validity

Construct Validity. We use MRR as an evaluation metric for the UR-based code localization task. However, since reviews often correspond to multiple code snippets, MRR may not fully capture the performance of the models in such one-to-many scenarios, leading to potential construct validity threats. To mitigate this, we also adopt Top-K as an additional metric. In future work, we plan to incorporate more evaluation metrics to comprehensively assess the model's performance.

Internal Validity. The hyperparameter K (coupling window size) poses an internal threat since it significantly affects code localization results. To mitigate this, we explore the values of K ranging from 1 to 10 in Section 5.3. In future work, we will further investigate the effect of larger K values on the final results.

External Validity. Although our proposed YourCoLo 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 article, we propose a novel approach named YourCoLo to address three key challenges in UR-based code localization. To improve the cross-project generalization ability, we propose to use prompts when fine-tuning to provide project information to the model. Then, we introduce a new

loss calculation method to take into account the one-to-many relationship between URs and code snippets. Finally, in order to identify and utilize the interconnection between multiple code snippets corresponding to the same UR, we analyze the dataset and propose a new code snippet re-ranking method. Experimental results show that YourCoLo significantly outperforms the previous SOTA, with gains of 7.7 and 7 percentage points in average MRR at the method and file levels, respectively, and improvements of 8.8 and 6.8 percentage points in MAP at the corresponding levels. We also show that each component presented in YourCoLo has a positive effect on the overall result. Finally, we conduct some case studies to demonstrate the superior performance of YourCoLo.

In future work, we plan to incorporate richer contextual information (e.g., function call graphs and file-level features) and develop an enhanced UR-based code localization approach to further improve localization accuracy.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[2] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *44th International Conference on Software Engineering*, 1443–1455.

[3] 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.

[4] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. arXiv:2207.14255. Retrieved from https://arxiv.org/abs/2207.14255

[5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 964–974.

[6] Genevieve Caumartin, Qiaolin Qin, Sharon Chatragadda, Janmitsinh Panjrolia, Heng Li, and Diego Elias Costa. 2024. Exploring the potential of Llama models in automated code refinement: A replication study. arXiv:2412.02789. Retrieved from https://arxiv.org/abs/2412.02789

[7] 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:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[8] 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.

[9] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805

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

[11] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. 2013. Development and deployment at Facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[12] Mikolaj Fejzer, Jakub Narebski, Piotr Przymus, and Krzysztof Stencel. 2021. Tracking buggy files: New efficient adaptive bug localization algorithm. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2557–2569.

[13] 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:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[14] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[15] Shanquan Gao, Lei Liu, Yuzhou Liu, Huaxiao Liu, and Yihui Wang. 2020. Updating the goal model with user reviews for the evolution of an app. *Journal of Software: Evolution and Process* 32, 8 (2020), e2257.

[16] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. On the use of relevance feedback in IR-based concept location. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 351–360.

[17] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *40th International Conference on Software Engineering*, 933–944.

[18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv:2203.03850. Retrieved from https://arxiv.org/abs/2203.03850

[19] 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:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366

[20] Emitza Guzman and Walid Maalej. 2014. How do users like this feature? A fine grained sentiment analysis of app reviews. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. IEEE, 153–162.

[21] 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.

[22] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. arXiv:2005.06980. Retrieved from https://arxiv.org/abs/2005.06980

[23] 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.

[24] 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.

[25] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 524–527.

[26] Minqing Hu and Bing Liu. 2004. Mining opinion features in customer reviews. In *American Association for Artificial Intelligence*, Vol. 4, 755–760.

[27] Xinwen Hu, Yu Guo, Jianjie Lu, Zheling Zhu, Chuanyi Li, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Lighting up supervised learning in user review-based code localization: Dataset and benchmark. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 533–545.

[28] Xuan Huo and Ming Li. 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *26th International Joint Conference on Artificial Intelligence*, 1909–1915.

[29] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2020. Control flow graph embedding based on multi-instance decomposition for bug localization. In *AAAI Conference on Artificial Intelligence*, Vol. 34, 4223–4230.

[30] Xuan Huo, Ferdian Thung, Ming Li, David Lo, Shu-Ting Shi. 2019. Deep transfer bug localization. *IEEE Transactions on Software Engineering* 47, 7 (2019), 1368–1380.

[31] 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.

[32] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[33] SayedHassan Khatoonabadi, 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.

[34] 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.

[35] Kisub Kim, Sankalp Ghatpande, Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2022. DigBug—pre/post-processing operator selection for accurate bug localization. *Journal of Systems and Software* 189 (2022), 111300.

[36] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *2015 10th Joint Meeting on Foundations of Software Engineering*, 579–590.

[37] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A survey of DevOps concepts and challenges. *ACM Computing Surveys* 52, 6 (2019), 1–35.

[38] Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022. Exploring representation-level augmentation for code search. arXiv:2210.12285. Retrieved from https://arxiv.org/abs/2210.12285

[39] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An evolving code generation benchmark aligned with real-world code repositories. arXiv:2404.00599. Retrieved from https://arxiv.org/abs/2404.00599

[40] Zhengliang Li, Zhiwei Jiang, Xiang Chen, Kaibo Cao, Qing Gu. 2021. Laprob: A label propagation-based software bug localization method. *Information and Software Technology* 130, (2021), 106410.

[41] Hongliang Liang, Dengji Hang, and Xiangyu Li. 2022. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering* 27, 7 (2022), 186.

[42] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive deep code search. In *28th International Conference on Program Comprehension*, 48–59.

[43] Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E. Hassan, and Shanping Li. 2021. Codematcher: Searching code based on sequential semantics of important query words. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–37.

[44] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[45] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 664–676.

[46] Yi-Fan Ma and Ming Li. 2022. Learning from the multi-level abstraction of the control flow graph via alternating propagation for bug localization. In *2022 IEEE International Conference on Data Mining (ICDM)*. IEEE, 299–308.

[47] Walid Maalej, Zijad Kurtanović, Hadeer Nabil, and Christoph Stanik. 2016. On the automatic classification of app reviews. *Requirements Engineering* 21 (2016), 311–331.

[48] 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.

[49] Stuart Mcilroy, Weiyi Shang, Nasir Ali, and Ahmed E. Hassan. 2017. User reviews of top mobile apps in apple and google app stores. *Communications of the ACM* 60, 11 (2017), 62–67.

[50] 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.

[51] 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 '11)*. IEEE, 263–272.

[52] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: Integrating semantic features with expert features for defect prediction and localization. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 672–683.

[53] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2136–2148.

[54] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *44th International Conference on Software Engineering*. 2006–2018.

[55] Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. 2024. Comparing the pretrained models of source code by re-pretraining under a unified setup. *IEEE Transactions on Neural Networks and Learning Systems* 35, 12 (2024), 17768–17778.

[56] Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. 2023. Crosscodebench: Benchmarking cross-task generalization of source code models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 537–549.

[57] Feifei Niu, Wesley K. G. Assunçao, 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.

[58] 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.

[59] 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.

[60] 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.

[61] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. 2016. Ardoc: App reviews development oriented classifier. In *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1023–1027.

[62] 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.

[63] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug localization on millions of files. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 225–236.

[64] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. 2021. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability* 71, 1 (2021), 235–249.

[65] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. 2024. Improving language understanding by generative pre-training. Mikecaptain.com. Retrieved February 15, 2024 from https://www.mikecaptain.com/resources/pdf/GPT-1.pdf

[66] 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:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[67] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[68] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–355.

[69] Abdus Satter and Kazi Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. *In 2016 19th International Conference on Computer and Information Technology (ICCIT)*. IEEE, 586–591.

[70] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. 2017. Listening to the crowd for the release planning of mobile apps. *IEEE Transactions on Software Engineering* 45, 1 (2017), 68–86.

[71] 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:2204.03293. Retrieved from https://arxiv.org/abs/2204.03293

[72] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 39–51.

[73] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *28th International Conference on Program Comprehension*, 196–207.

[74] Statista. 2024. Number of Mobile Cellular Subscriptions Worldwide from 2010 to 2028 (in millions). Retrieved December 08, 2024 from https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/

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

[76] A. Vaswani. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

[77] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One adapter for all programming languages? Adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 5–16.

[78] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *22nd International Conference on Program Comprehension*, 53–63.

[79] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *31st IEEE/ACM International Conference on Automated Software Engineering*, 262–273.

[80] 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.

[81] 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.

[82] Zhou Yang, Jieke Shi, Shaowei Wang, and David Lo. 2021. Incbl: Incremental bug localization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1223–1226.

[83] 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.

[84] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *28th International Conference on Program Comprehension*, 219–229.

[85] 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.

[86] 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.

[87] 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.
[88] Liming Zhu, Len Bass, and George Champlin-Scharff. 2016. DevOps and its practices. *IEEE Software* 33, 3 (2016), 32–34.
[89] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. 2020. Cooba: Cross-project bug localization via adversarial transfer learning. In *26th International Joint Conference on Artificial Intelligence*.

Received 25 December 2024; revised 28 August 2025; accepted 15 September 2025