A Knowledge Enhanced Large Language Model for Bug Localization

YUE LI, Nanjing University, China
BOHAN LIU, Nanjing University, China
TING ZHANG, Singapore Management University, Singapore
ZHIQI WANG, Nanjing University, China
DAVID LO, Singapore Management University, Singapore
LANXIN YANG[*], Nanjing University, China
JUN LYU, Nanjing University, China
HE ZHANG[*], Nanjing University, China

A significant number of bug reports are generated every day as software systems continue to develop. Large Language Models (LLMs) have been used to correlate bug reports with source code to locate bugs automatically. The existing research has shown that LLMs are effective for bug localization and can increase software development efficiency. However, these studies still have two limitations. First, these models fail to capture context information about bug reports and source code. Second, these models are unable to understand the domain-specific expertise inherent to particular projects, such as version information in projects that are composed of alphanumeric characters without any semantic meaning.

To address these challenges, we propose a **K**nowledge **E**nhanced **P**re-**T**rained model using project documents and historical code, called **Kept**, for bug localization. Project documents record, revise, and restate project information that provides rich semantic information about those projects. Historical code contains rich code semantic information that can enhance the reasoning ability of LLMs. Specifically, we construct knowledge graphs from project documents and source code. Then, we introduce knowledge graphs to the LLM through soft-position embedding and visible matrices, enhancing its contextual and professional reasoning ability. To validate our model, we conducted a series of experiments on seven open-source software projects with over 6,000 bug reports. Compared with the traditional model (*i.e.*, Locus), Kept performs better by 33.2% to 59.5% in terms of mean reciprocal rank, mean average precision, and Top@N. Compared with the best-performing non-commercial LLM (*i.e.*, CodeT5), Kept achieves an improvement of 36.6% to 63.7%. Compared to the state-of-the-art commercial LLM developed by OpenAI, called *text-embedding-ada-002*, Kept achieves an average improvement of 7.8% to 17.4%. The results indicate that introducing knowledge graphs contributes to enhance the effectiveness of the LLM in bug localization.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: large language model, knowledge enhancement, bug localization, information retrieval

[*]He Zhang is the corresponding author. Lanxin Yang is also the corresponding author.

Authors' Contact Information: Yue Li, Nanjing University, Nanjing, China, yue.li@smail.nju.edu.cn; Bohan Liu, Nanjing University, Nanjing, China, bohanliu@nju.edu.cn; Ting Zhang, Singapore Management University, Singapore, Singapore, tingzhang.2019@phdcs.smu.edu.sg; Zhiqi Wang, Nanjing University, Nanjing, China, 502022320013@smail.nju.edu.cn; David Lo, Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg; Lanxin Yang, Nanjing University, Nanjing, China, lxyang@nju.edu.cn; Jun Lyu, Nanjing University, Nanjing, China, lvjun_dnt@outlook.com; He Zhang, Nanjing University, Nanjing, China, hezhang@nju.edu.cn.

ACM 2994-970X/2025/7-ARTFSE086
https://doi.org/10.1145/3729356

ACM Reference Format:
Yue Li, Bohan Liu, Ting Zhang, Zhiqi Wang, David Lo, Lanxin Yang, Jun Lyu, and He Zhang. 2025. A Knowledge Enhanced Large Language Model for Bug Localization. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE086 (July 2025), 23 pages. https://doi.org/10.1145/3729356

1 Introduction

As software systems are developed and evolve, a large number of bug reports are generated, making bug localization critical and labor-intensive [8, 16, 44]. Effective bug localization enhances software maintenance efficiency and ensures software product quality. Bug localization is a time-consuming task that requires understanding bug reports and locating the source code that needs to be fixed. A recent study shows that 32% of time and cost is spent on bug localization during the software development life cycle [42]. Bug localization remains a challenging task due to the significant gap between the natural language used to describe bugs in bug reports and the programming language used to develop software products in source code [57]. Further research is required to improve the effectiveness and accuracy of bug localization techniques.

In the literature, there are quite a few studies that propose automatic linkage of bug reports in bug tracking systems with buggy code in version control systems [19, 28, 31, 55]. Information retrieval (IR) techniques are proposed and explored to be effective in bug localization, demonstrating the potential to improve software developers' productivity [4]. These techniques take historical bug reports and source code as inputs and output a ranked list of source code that may cause the bugs. Several researchers have calculated the semantic similarity of bug reports and source code by extracting semantic features between them. In recent years, LLMs have gained considerable attention from researchers in bug localization since it has achieved performance breakthroughs in various natural language processing tasks (*e.g.*, IR) [12].

LLMs have demonstrated boosted performance in various areas, prompting researchers to use them in bug localization [46]. LLMs can effectively capture knowledge from large amounts of data because of their sophisticated pre-training objectives and massive model parameters [12]. The rich knowledge implicitly encoded in massive parameters can benefit a variety of downstream tasks when stored in many parameters and fine-tuned for specific tasks, as demonstrated in extensive experimental verifications and empirical analyses [46].

LLMs are used for a variety of purposes, including bug localization since they possess powerful capabilities for capturing knowledge. Ciborowska and Damevski [4] applied BERT to rapidly locate bugs based on changesets by encoding changesets in different ways. The evaluation results reveal the advantages of using the proposed BERT model in comparison with the baselines for bug localization. Discriminative LLMs, such as BERT, serve as a kind of auto-denoising encoder for automatically corrupted texts [56]. Agnieszka and andKostadin explore several design decisions related to using BERT for changeset-based bug localization [4]. The evaluation results demonstrate that the proposed BERT model has advantages over the baselines, especially for bug reports that do not contain any hints regarding related code elements. Du and Yu represent source code through semantic flow graphs, and evaluation results indicate that their method achieves state-of-the-art bug localization performance [5].

Although these LLMs have shown promising results in bug localization, they suffer from the following two weaknesses.

First, these models have difficulty in capturing the context information of bug reports and source code. The descriptions of bug reports and the contents of changesets are usually brief and cannot provide sufficient background information. As a result, the model is unable to fully comprehend the content of the bug report and the content of the source code.

Second, these models fail to understand the domain-specific expertise of particular projects. As domain-specific text, bug reports contain large amounts of domain-specific knowledge that rarely appears in natural language corpora, making it difficult for LLMs to understand their accurate meaning. Furthermore, source code is a highly structured and formalized language that encapsulates a wide range of project-specific knowledge. For example, the version information is included in bug reports, and the class names are defined in the code.

To alleviate these problems, we propose KEPT, an LLM that leverages domain knowledge of project documents and source code, which introduces domain knowledge to enhance an LLM. By leveraging terminology and natural language descriptions in documents and the historical source code of the project, which can be readily accessed from public repositories, KEPT empowers LLMs to memorize and reason about bug reports and source code. In addition, it is consistent with the process of bug localization performed by experts, who also need to consult project documents and source code when they encounter unfamiliar bug reports.

In this paper, we design novel model structures to effectively use domain knowledge to improve the LLMs' memorization and reasoning abilities for bug reports and source code. First, we construct a text knowledge graph based on the historical documents and a code knowledge graph based on the source code. Then, we integrate the extracted knowledge graphs into the LLM to enhance its reasoning abilities for expertise in bug reports and source code. Incorporating too much knowledge could lead to a deviation from the intended meaning of the sentence. To overcome these challenges, we use soft-position embedding to reconstruct the position of the token and visible matrix to limit the visible range of each token.

To evaluate the effectiveness of KEPT, we compare its results with five state-of-the-art models (*i.e.*, Locus [47], GPT-2 [32], GraphCodeBERT [11], CodeT5 [46], and *text-embedding-ada-002*). Locus is the first model to use changesets to locate bugs. GPT-2 and CodeT5 have been widely recognized as leading LLMs because of their robust capabilities. GraphCodeBERT is an advanced LLM that uses graphs to illustrate the inherent structure of code. *text-embedding-ada-002* is an efficient text embedding model developed by OpenAI, capable of generating high-quality vector representations [20]. We conduct a series of evaluations of seven open source software (OSS) with over 6,000 bug reports to validate our model. As measured across OSS projects, KEPT outperforms Locus by 33.2% to 59.5% on average, and outperforms CodeT5, the best-performing LLM, by 36.6% to 63.7% on various evaluation metrics (*i.e.*, mean reciprocal rank, mean average precision, and Top@N). The experimental results show that KEPT outperforms all five state-of-the-art models in bug localization. In addition, the results indicate that our approach can improve the effectiveness of three LLMs for bug localization.

The main contributions of this study can be concluded as follows:

- **A knowledge enhanced pre-trained approach:** To the best of our knowledge, this study is the first attempt to enhance the LLM by introducing domain knowledge in bug localization, which has the potential to improve LLMs' memorization and reasoning abilities for bug reports and source code.
- **A novel model:** We present a model for representing knowledge graphs using soft-position embedding and visible matrices to effectively inject domain knowledge without introducing knowledge noise into the model.
- **An extensive evaluation:** We evaluate the effectiveness of the knowledge-enhanced LLM on over 6,000 bug reports in seven OSS projects. The experimental results indicate that our knowledge-enhanced LLM outperforms the state-of-the-art models. In particular, LLMs with knowledge enhancement perform significantly better than those without.

2 Background and Motivation

2.1 IR-based Bug Localization

IR technologies play an important role in bug localization, which involves matching bug reports with source code. LLMs have exhibited outstanding performance in various IR tasks, including bug localization [27]. To illustrate the process, we describe the steps involved in using an LLM for bug localization. The architecture of an LLM typically consists of multiple Transformer encoders, which are designed to model sequence data using a self-attention mechanism [6]. The concept of attention is to assign different weights to specific words in the sequence, *i.e.*, to encode a stronger relationship between each word in the sequence and its semantically related word. The process of using an LLM for bug localization involves three main steps: first, the model is pre-trained on a large corpus of Software Engineering (SE) data, then it is fine-tuned for bug localization, and finally, suspicious code files are retrieved for a new bug report.

2.2 Existing Methods with LLM

2.2.1 LLM For Bug Localization. During pre-training, an LLM uses a large corpus of relevant text to construct a domain-specific language model, such as an SE language model trained on Stack Overflow data [39]. Since the pre-training step requires a large amount of data and resources, many researchers opt to fine-tune LLMs for downstream tasks rather than training models from scratch.

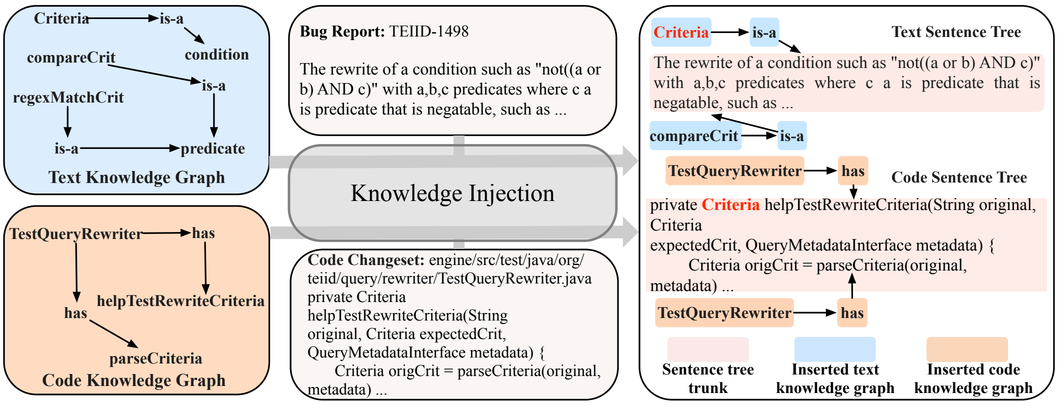
The fine-tuning is designed to train an LLM on a small dataset to apply it to specific tasks, such as bug localization. Typically, the LLM is fine-tuned by adding additional layers, which incorporate the output of an LLM as inputs. Since the goal of this study is to locate the bug-inducing code files, it is necessary to select a task-specific dataset that includes bug reports and buggy code files.

2.2.2 Knowledge Enhancement LLMs. Despite the advantages that LLMs possess, such as efficiently learning rich knowledge from large training texts and utilizing downstream tasks during their fine-tuning phase, they still have some limitations, such as poor inference abilities due to deficiencies in external knowledge.

LLMs have been widely applied to many fields in recent years due to their boosted performance. Although LLMs have the advantage of efficiently learning rich knowledge from large training texts and benefiting downstream tasks during their fine-tuning stages, they still possess some limitations, such as poor inference abilities due to a lack of external knowledge. To address these issues, research has been dedicated to integrating knowledge graphs into LLMs. Generally, knowledge-enhanced approaches are divided into the following two categories [13]: By integrating knowledge graphs through pre-trained tasks, LLMs can learn representations related to knowledge graphs during the pre-trained phase.

Through pre-trained tasks. These approaches design a joint pre-trained task that integrates entity embedding with text embedding. This task combines knowledge graph embedding tasks with training tasks using masked language modeling (MLM). While injecting knowledge graphs into LLMs through pre-trained tasks is feasible, this integration approach is inefficient and expensive [38]. On the one hand, introducing knowledge graphs through pre-trained tasks increases the additional training workload. On the other hand, the model is unable to adapt to changes in the knowledge graph, increasing the training cost when the knowledge graph changes [13].

Changing the attention mechanisms. To integrate corresponding knowledge into LLMs, several approaches attempt to modify the attention mechanism within the Transformer architecture. For example, KEAR [51] splices knowledge graph entity-related triples, entity descriptions, and additional questions as external knowledge with the original input, and then applies a self-attention mechanism to the spliced input. This category of approaches can integrate knowledge graphs without changing the model structure, and it is suitable for most LLMs using the Transformer



Bug Report: TEIID-1498

The rewrite of a condition such as "not((a or b) AND c)" with a,b,c predicates where c a is predicate that is negatable, such as ...

Code Changeset: engine/src/test/java/org/teiid/query/rewriter/TestQueryRewriter.java

```
private Criteria
helpTestRewriteCriteria(String
original, Criteria expectedCrit,
QueryMetadataInterface metadata) {
     Criteria origCrit = parseCriteria(original,
metadata) ...
```

Fig. 1. An example of knowledge injection

encoder architecture. In addition, the models can adapt well to changes in the knowledge graph without retraining [13]. Since approach changing attention mechanisms can adapt to changes in knowledge graphs and introduce them at a lower cost than others, we also use the approach to integrating knowledge graphs into LLMs.

Bug localization is of paramount importance to software maintenance, and it has achieved promising advancements. However, bug localization remains an open question, given the limitation caused by the significant difference between the natural language describing bugs and the source code implementing software. Existing bug localization methods using LLMs often lack external knowledge, limiting their effectiveness.

Motivated by the aforementioned challenges, we propose a knowledge-enhanced approach. Figure 1 illustrates a real example in the JIRA system. A bug report[1] includes the following description: “The rewrite of a condition such as `"not((a or b) AND c)"` with a,b,c predicates where `c` is a predicate that is negatable, such as `col1 =` 1. The result should have the negated form of `col1 <>` 1, but instead has `col1 =` 1. This is because the `col1 = 1` predicate is repeated in the result, but not cloned and the rewriter modifies the same instance”. The provided description alone is insufficient for precise bug localization due to its generic nature and lack of specific code references.

2.3 Motivation

Our approach addresses these challenges by (1) extracting a textual knowledge graph from the historical project documentation, (2) creating a code knowledge graph from its historical source code, and (3) integrating these knowledge graphs into the sentence trees of a bug report and code changesets. Fig. 1 illustrates the extracted textual and code knowledge graphs on the left side.

We integrate external knowledge using knowledge graphs: (1) the entries in knowledge graphs are matched with the entities in bug reports and suspicious code; (2) the matched entries in knowledge graphs are added to the bug reports and suspicious code to effectively inject domain knowledge. As shown in Figure 1, incorporating **`Criteria`** `is-a condition` in the text sentence tree enables the bug report to be matched with buggy code. In addition, the model can benefit from integrating other knowledge graphs, such as `TestQueryRewriter` has `parseCriteria`, to facilitate understanding of relationships between entities. Obviously, without contextual information on the bug report and suspicious code, it is difficult for the model to understand them. A number of researchers have argued that bug reports and source code are difficult to understand without contextual information in the literature [59].

[1]https://issues.redhat.com/browse/TEIID-1498

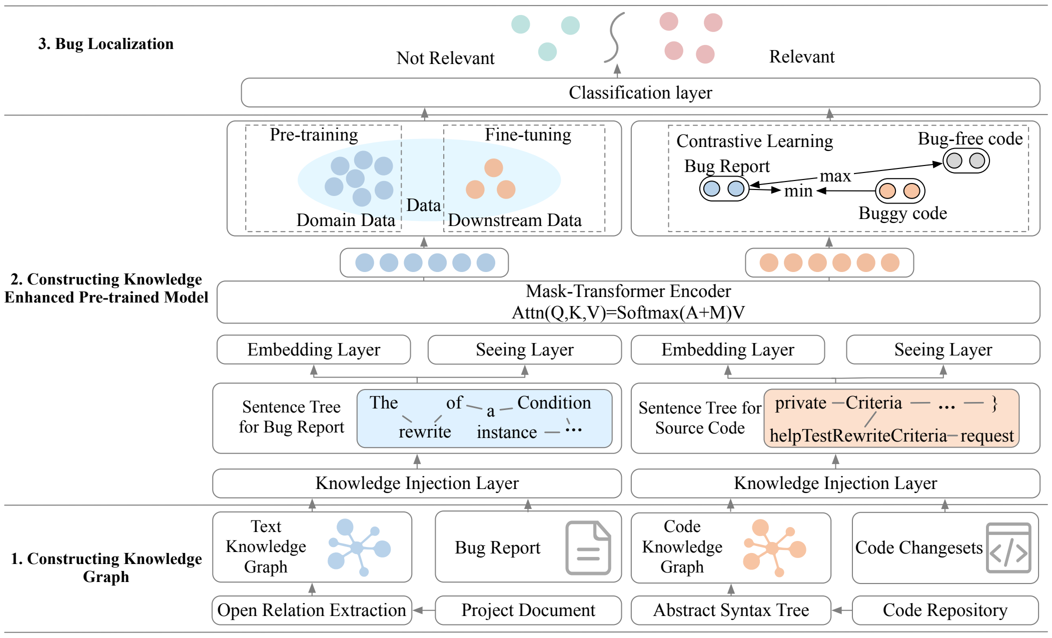


Fig. 2. An overview of the proposed approach

By enhancing LLMs with this external knowledge, we aim to improve the accuracy and effectiveness of bug localization, addressing the limitations of existing methods and bridging the gap between natural language bug descriptions and source code.

3 Approach

3.1 Definitions of Knowledge Graph

This study aims to construct a knowledge-enhanced LLM that leverages domain knowledge to enhance LLM for bug localization. This subsection provides a formal definition of related concepts to assist readers in understanding the scope of this project.

Definition 1: Knowledge graph. A knowledge graph is defined as a system in which all entities (*e.g.*, things, places) are stored in the form of connected graphs [17].

Definition 2: Text knowledge graph. We denote a sentence s =$\{w_0, w_1, w_2, ..., w_n\}$ as a sequence of tokens, where n is the length of this sentence. The tokens of the sentence are taken at the word level. Each token w_i is included in the vocabulary V, $w_i \epsilon V$. Text knowledge graph, denoted as TKG, is a collection of triples $\varepsilon = (w_i, wr_j, w_k)$, where w_i and w_k are the names of entities, and $wr_j \epsilon V$ is the relation between them.

Definition 3: Code knowledge graph. We denote a code snippet c =$\{c_0, c_1, c_2, ..., c_n\}$ as a sequence of tokens, where n is the length of this code snippet. The tokens of the sentence are taken at the character level. We construct an abstract syntax tree (AST) from the source code file using JavaParser. The nodes in AST constitute nodes in the code knowledge graph. Code knowledge graph, denoted as CKG, represents a set of triples $\zeta = (c_i, cr_j, c_k)$, in which c_i and c_k represent the names of entities, and $cr_j \epsilon AST$ is the relation between them.

3.2 Overview

In Figure 2, we provide an overview of Kept that aims to improve the accuracy of bug localization by incorporating domain-specific information into an LLM. The project documents and the source code of the project serve as input domain knowledge, while the history of bug reports and buggy code serve as input historical knowledge. The approach consists of three steps.

First, we construct knowledge graphs (Section 3.3). To construct a text knowledge graph, we use Natural Language Toolkit (NLTK) [24] to filter noise and use the open relationship extraction technology MinIE [7] to extract entities and relationships in project documents. NLTK and MinIE provide higher precision and recall in extracting knowledge graphs from real-world projects than most other tools [7, 24]. To construct a code knowledge graph, we extract the AST structure of the project source code using JavaParser[2], and traverse the AST to obtain entities and relationships in the source code.

Second, we construct the knowledge-enhanced LLM (Section 3.4). The step consists of four parts: (1) Model embedding: This part involves generating vector representations of text and code through model embedding. There are three layers involved in this process *e.g.*, the knowledge injection layer, the knowledge embedding layer, and the seeing layer. (2) Mask Transformer: The input embeddings and visible matrix are fed into a mask self-attention, which differs from the traditional Transformer encoder. (3) Pre-training model: The UniXcoder model is pre-trained using SE corpora (*i.e.*, CodeSearchNet) to adapt to language features and terminologies because of UniXcoder's superior performance in code understanding. (4) Fine-tuning model: The model is fine-tuned using historical data on bug localization.

Finally, we use the LLM to locate bugs. We input the semantic features learned by the LLM into the fully connected layer for bug localization.

3.3 Constructing Knowledge Graphs

We construct two types of knowledge graphs in this study: (1) a text knowledge graph and (2) a code knowledge graph.

3.3.1 Constructing Text Knowledge Graphs. We construct a text knowledge graph to effectively capture domain knowledge from project documents. First, we use the NLTK module for part-of-speech tagging to filter out noise and segment the content of project documents into lists of sentences. Second, we use the open relation extraction method MinIE [7] to extract entities and relationships from the project. MinIE extracts knowledge graphs efficiently from a variety of domains, demonstrating robust adaptability across varied corpora. In addition, MinIE constructs a minimized dictionary to filter and remove redundant data from the triples. MinIE is unable to extract triples directly from web pages, so preprocessing is necessary before extracting the triples. The preprocessing consists of three steps: (1) Text extraction: We use Python's BeautifulSoup library to extract textual elements from the documents, which is a widely used library for parsing HTML and XML, and constructs a structured parse tree to efficiently extract HTML elements; (2) Noise reduction: To minimize noise, we use regular expressions to remove unnecessary whitespace, non-linguistic elements (e.g., URLs and email addresses), and special characters. (3) Sentence segmentation: The document is segmented into sentences by using the NLTK library, a widely used natural language processing tool that accurately segments sentences across a variety of languages and formats. After preprocessing, MinIE's recognizer extracts triples from the sentences. The extracted triples are then refined using MinIE's minimization dictionary, which filters out redundant information from the triples.

3.3.2 Constructing Code Knowledge Graphs. By using JavaParser, we can obtain the AST of the source code, and then traverse the AST to obtain a collection of triplets that are entities of the knowledge graph. Following this, we extract the relations possessed by the head and tail entities

[2]https://github.com/javaparser/javaparser

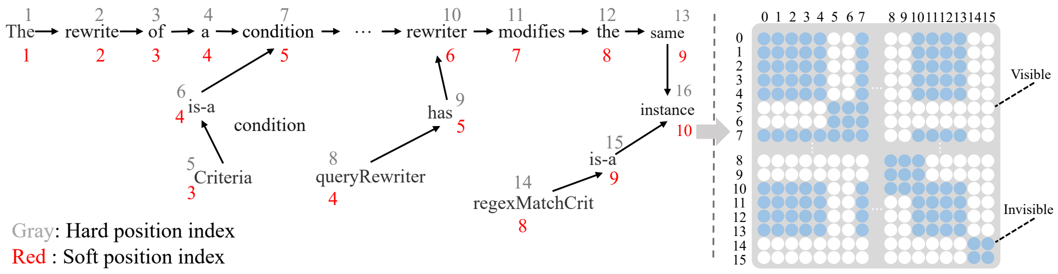


Fig. 3. The process of converting a sentence tree into a visible matrix

of the triples. We traverse the AST to obtain all triples of the relevant source code to build a code knowledge graph.

3.4 Constructing Knowledge Enhanced LLM

3.4.1 Model Embedding. Model embedding consists of three main components: the knowledge injection layer, the embedding layer, and the seeing layer. KEPT uses the knowledge injection layer to retrieve triples corresponding to input bug reports or buggy code (referred to as input sequences). Then, the triples are injected into the input sequence to generate a sentence tree infused with knowledge. Finally, the sentence tree is fed into both the knowledge injection layer and the seeing layer, where it is transformed into input embeddings and visible matrixes. Through this structure, the model can determine the visible range of each token in the self-attention mechanism, thus enabling it to acquire external knowledge without losing its original meaning.

The sentence tree is constructed as follows: (1) we segmented bug reports using UniXcoder's word splitter and code using Pygments[3], forming the original sentence tree; (2) we matched entities in the original sentence tree with those in the constructed knowledge graph; (3) we enriched the sentence tree by incorporating matched triples from the knowledge graph, producing the final sentence tree.

Knowledge Injection Layer. The knowledge injection layer is used to query the triples contained in the original input based on specific rules, and then integrate the original input with the triplet knowledge to form a sentence tree. Specifically, we use Pygments for code disambiguation and UniXcoder for bug report disambiguation and then traverse the segmented words to generate the corresponding sentence tree nodes. Afterward, the algorithm then iterates over the segmented words and generates a sentence tree node for each word. In the next step, the algorithm determines if the word has a corresponding entity in the knowledge graph, and if so, adds a ternary relationship between the node and the entity. Finally, the algorithm adds the processed sentence tree node to the sentence tree.

Given an input sentence ($sentence = w_0, w_1, w_2, ..., w_n$) and a knowledge graph K, the knowledge injection layer outputs a sentence tree ($tree = w_0, w_1, ..., w_i((r_{i0}, w_{i0}), ..., (r_{ik}, w_{ik})), ..., w_n$), where w_i represents a word or entity in the input sentence, and $T = (w_i, r_{i0}, w_{i0}), ..., (w_i, r_{ik}, w_{ik})$ represents the set of triples associated with the word w_i, where w_i, r_{ik}, and w_{ik} are the head entities, relation, and tail entities of the triple, respectively.

Embedding Layer. The embedding layer converts the sentence tree into corresponding embedding representations that can be fed into the mask Transformer. Similar to the Transformer architecture, the input embedding of KEPT is composed of token embedding and position embedding, with the difference being the use of a soft-position embedding. The structure facilitates the transformation

[3]https://pygments.org/

of the sentence tree into token embedding and positional embedding for the mask Transformer to retain its structural features.

Soft-position embedding ensures the correct order of a sentence tree. As shown in Figure 3, introducing triples from the knowledge graph can disrupt the original structure of the sentence tree. For example, the sentence "`The rewrite of a condition`" was reordered to "`The rewrite of a Criteria`" due to the triple (`condition, is-a, Criteria`) introduced from the knowledge graph. Soft-position embedding resolves such disruptions, maintaining the correct order in the sentence tree.

Seeing Layer. Although soft-position embedding ensures the sequential correctness of input sequences with their triples, tokens with identical position indices may cause improper correlation in the self-attention mechanism. In addition, expanding the sentence tree introduces newly inserted triples that affect all tokens, changing the original meaning of the input sequence. For instance, in the Figure 3, `Criteria` is used to decorate `condition`, which is unrelated to `rewrite`. Accordingly, `rewrite` should not be influenced by `Criteria`.

To address this problem, we introduce the seeing layer, a visibility matrix, which reduces the impact of irrelevant information on model learning. The visible matrix is a technique used to control the exposure or visibility of different tokens during the learning or prediction process. It serves as a mechanism to ensure that the model does not inject excessive or irrelevant knowledge that could distort the meaning of the original sentence or context.

The visible matrix M is defined as shown in Equation 1.

$$M_{ij} = \begin{cases} 0 & t_i \oplus t_j \\ -\infty & \text{otherwise} \end{cases} \tag{1}$$

The visible matrix comprises elements of 0 and $-\infty$, where t_i and t_j represent token elements in the token embedding after the sentence tree is transformed, and i and j are their respective indices in the token embedding. $t_i \oplus t_j$ indicates that t_i and t_j are either in the original input sequence or belong to the same relation branch of the sentence tree, meaning they are members of the same triple. Figure 3 illustrates the structure of the visible matrix in detail, as gray numbers represent the true positions of tokens in the token embedding. The blue nodes indicate that the two tokens are visible in the visible matrix, and their value is 0, while the white nodes indicate that the two tokens are invisible, and their value is $-\infty$.

3.4.2 Mask Transformer. The traditional Transformer architecture cannot use a visible matrix as input, nor can it selectively limit the attention range of each input token. To address this limitation, we propose a mask self-attention mechanism called mask Transformer. The mask Transformer comprises N mask self-attention layers stacked on top of each other. We use a multi-head mask attention layer based on the mask self-attention mechanism to compute the attention mechanism. The formula for the mask self-attention mechanism is shown in Equation 2.

In the equation, H_i represents the hidden state of each layer. When $i = 0$, H_0 is the input embedding of the model. W_q, W_k, and W_v are learnable parameters of the model, and M is the visible matrix. The equation calculates the query vector Q_i, key vector K_i, and value vector V_i corresponding to the input H_i. Then, it calculates the attention score A_i using the query vector Q_i and key vector K_i, where d_k represents the dimension of the hidden vector of the model. After that, the attention score A_i is added to the visible matrix M, and the SoftMax function is applied to obtain the attention distribution S_i. Finally, the attention mechanism output is calculated by multiplying the attention distribution S_i with the value vector V_i.

$$\begin{aligned} Q^i &= H^i \times W_q \\ K^i &= H^i \times W_k \\ V^i &= H^i \times W_v \\ A^i &= \frac{Q^i K^{i\top}}{\sqrt{d_k}} \\ SoftMax(x) &= \frac{e^x}{\sum_{j=1}^{n} e^{x_j}} \\ S^i &= \text{SoftMax}\left(A^i + M\right) \\ H^{i+1} &= S^i \times V^i \end{aligned} \tag{2}$$

In the self-attention mechanism, A_{ij} represents a matrix where $A_i \in A_i$ signifies the attention score between the jth and kth tokens in the input embedding. When tokens i and j are not visible, $M_{jk} = -\infty$, thus $M_{jk} + A_{ij} = -\infty$. Consequently, $\text{SoftMax}(A_{ij} + M_{jk}) = 0$, ensuring that the attention distribution S_i does not allow token j to influence the hidden state of token i, thereby restricting attention between different tokens. Under the masked self-attention mechanism, newly introduced triple information does not directly affect the hidden state of the [CLS] token but serves as a bridge through the original input sequence's triple head entity. As illustrated in Figure 3, `is-a` is a token in the knowledge triple, and it is invisible to `condition` implies no impact of h_{i-1} on h_i. However, `has` and the head entity `rewriter` in the triple is visible, allowing h_{i-1} to influence h_i. Similarly, `is-a` is visible to `instance`, enabling h_{i+1} to indirectly acquire information from h_{i-1}. This approach maximizes the utilization of triple information to enhance the representation of entities in the original input sequence, introducing knowledge information to improve the representation capability of the original input.

3.4.3 Pre-training Model. We use the UniXcoder [10] due to its superior performance on various code understanding and generation tasks to learn the word distribution between bug reports and buggy codes. UniXcoder demonstrates adaptability to various tasks. Its performance in code understanding tasks, such as code search, is superior to other LLMs such as GPT-2 [10]. The model uses mask attention matrixes with prefix adapters to control its behavior and leverages cross-modal content, including AST and code annotations, to enhance its code representation.

We conduct pre-training of Kept using a large code search corpus to improve its reasoning ability for the relationship between textual data and code snippets. This task involves matching code with text, aligned with the bug localization objective to increase the accuracy of matching bug reports with code snippets. CodeSearchNet serves as the training data set for code search training, which uses the same model architecture as bug localization training. The data set consists of millions of code snippets and text on GitHub, with the Java language part containing 503,502 text-code pairs.

3.4.4 Fine-tuning Model. After pre-training, we fine-tune Kept to ensure it is effective at implementing downstream tasks and to maximize the utilization of the knowledge graph. During the fine-tuning process, we employ the $\langle R, C, L\rangle$ as input of Kept, where R represents the bug report, C represents the buggy code related or unrelated to the bug report, and L is the label of the relationship between them. Contrastive learning is used to reduce the distance between bug reports and buggy changesets and to increase the distance between bug reports and non-buggy changesets. If R is related to C, then $L = 1$; otherwise, $L = 0$. In contrastive learning, the online negative sampling method proposed by Lin et al. [23] is used to select negative samples.

3.5 Bug Localization

We use historical bug reports and code files (*i.e.*, data in the training set) to train our models for new bug reports. The learned semantic features are input into a fully connected layer for bug localization. As the model is trained iteratively, the loss function is monitored and the weights of semantic features are optimized over several epochs.

After receiving a new bug report, Kept processes the new bug report and all historical changesets, creating separate hidden state matrices for them. The hidden state matrices are pooled and concatenated to generate joint feature vectors for the new bug report and changesets. Then, the joint feature vectors are fed into a classifier, which predicts a two-dimensional vector (m, n), where m represents the unrelated score, and n represents the related score. Finally, suspicious changesets related to the new bug report are localized based on the scores of the two-dimensional vector.

4 Experimental Evaluation

In this section, we present the research questions, the studied dataset, the baselines, and the evaluation metrics. The dataset and the source code are available online[4].

4.1 Research Questions

We first compare our model with the state-of-the-art bug localization techniques in our experiments to evaluate its effectiveness (RQ1). Afterward, we implement an ablation experiment to evaluate the impact of introducing knowledge graphs on the LLM for bug localization (RQ2). Finally, we conduct another ablation experiment to evaluate the impact of soft-position embedding and visible matrices on three LLMs (RQ3).

- **RQ1:** How effective is Kept compared to the state-of-the-art models for bug localization?
- **RQ2:** How do knowledge graphs impact the effectiveness of Kept?
- **RQ3:** How do soft-position embedding and visible matrices impact the accuracy of LLMs in bug localization?

4.1.1 RQ1: Effectiveness of Kept. In recent years, graph-related techniques have been used to incorporate structural information contained in code into LLMs. These LLMs based on code structure graphs have demonstrated reasonable performance improvements in software engineering. However, previous research on LLMs often overlooks the interrelated information within the code and fails to integrate domain knowledge to address the bug localization problem. In this paper, we propose an approach to locating bugs that leverages a knowledge-enhanced LLM to incorporate domain knowledge and interrelated code information. To validate our proposed model, we compare it to five advanced and representative baseline models.
Experimental Settings: The five collected baselines are Locus [47], GPT-2 [32], GraphCodeBERT [11], CodeT5 [46], and *text-embedding-ada-002*. Locus is a traditional IR-based method for locating bugs based on changesets [47]. GPT-2, GraphCodeBERT, and CodeT5 are advanced LLMs. We use the Locus implemented by Bench4BL [19] to conduct experiments. *text-embedding-ada-002* represents the state-of-the-art commercial text embedding model [20]. To ensure fairness in our experiment, we used the open-source models and parameters reported in their papers for GPT-2, CodeT5, GraphCodeBERT, and *text-embedding-ada-002*. In addition, we performed the fine-tuning of the baseline for all large language models to ensure fairness. To implement our Kept model, we mainly use two Python libraries, *i.e.*, Transformers [48] and PyTorch [30]. Since the impact of time on the training and testing set, we adopted the strategy proposed by Lin et al. [23] to build the training and testing datasets. We used 50% of bug reports and buggy code in projects as the

[4]The source dataset and source code for this study are available online at https://github.com/keptmodel/KEPT

Table 1. Details of the OSS projects

Project	Time Span	#Bug Report	#Code Files	#Changesets			Code knowledge graph		Text knowledge graph	
				Commits	Files	Hunks	Relation	Entity	Relation	Entity
Teiid	2004-04-27~2017-04-03	1297	2291	1637	8843	21295	96985	212882	1748	8208
Hornetq	2006-05-17~2015-03-31	270	1618	327	1439	3826	63253	137068	500	1794
Seam2	2005-09-11~2014-03-16	776	1510	881	2172	4222	17076	35473	719	3149
Weld	2009-02-07~2017-03-24	560	2515	704	3668	6101	21910	40701	353	1171
Drools	2006-03-10~2017-03-30	1281	2231	1649	8091	24526	73081	155280	1021	3993
Derby	2004-09-28~2016-11-27	1778	2408	2590	11435	26485	111960	243542	964	3764
Log4j2	2011-09-16~2017-03-12	441	920	735	3516	6886	18853	37527	475	1523
Total	-	6403	13493	8523	39164	93341	403118	862473	5780	23602

training set and the other 50% as the testing set. To prevent data leakage, we only used project documents and code available up to the end of the training set as data sources for constructing the knowledge graph. To address the RQs posed in this paper, experiments were conducted on a server equipped with two NVIDIA A100 Tensor Core GPUs. To construct the LLMs, two Python libraries were primarily utilized: Transformers V4.36.2 and PyTorch V1.11.0. During training, the PyTorch library provides computational support for Transformer-based model architectures, while the Transformers library provides an API for accessing Transformer-based model architectures. Regarding the configuration parameters of the model, the number of layers in the masked self-attention layer and the number of heads in the multi-head attention mechanism were denoted as L and A, respectively. Furthermore, the hidden dimension of the embedding vectors was represented by H. The parameters of the model were configured as follows: L = 12, A = 12, H = 768.

4.1.2 RQ2: Effectiveness of Knowledge Graph. In this study, we use project documents to construct the text knowledge graph. The knowledge graph contains domain-specific information about the projects, which has the potential to enhance the LLM's memorization and reasoning abilities for bug reports. In addition, we use project source code to build the code knowledge graph. The code knowledge graph represents the interrelated information of the code, offering potential enhancements to LLMs' memorization and reasoning abilities for source code. We conduct an ablation experiment to study the effectiveness of knowledge graphs on the model performance.
Experimental Settings: To answer RQ2, we establish four scenarios: no knowledge graphs, only text knowledge graphs, only code knowledge graphs, and both text and code knowledge graphs. Similar to RQ1, we use the first 50% of bug reports and buggy code from projects as a training set, while the remaining 50% served as a testing set.

4.1.3 RQ3: Effectiveness of Soft-Position Embedding and Visible Matrix. To effectively represent the information extracted from knowledge graphs, we propose a structure using soft-position embedding and visible matrix. We use the structure to convert the sentence tree into the token embedding and position embedding of the mask encoder, while retaining the structural characteristics of the sentence tree. To validate the impact of the proposed model structure on the models, we conduct an ablation experiment to evaluate the effect of this model structure on three LLMs in bug localization.
Experimental Settings: To answer RQ3, we conduct an ablation experiment with three LLMs: BERT, CodeBERT, and UniXcoder. All three models adopt an encoder structure. We evaluate the impact of the proposed model structure on bug localization by incorporating the proposed structure into these models and comparing them with the original LLMs. The experimental data and model settings are consistent with those of RQ1.

4.2 Dataset

4.2.1 Datasets for bug localization. To address the RQs, we utilized the bug localization dataset collected and validated by Rath et al [33]. We reuse seven of the 15 OSS projects from their dataset. For the remaining eight projects, the source code repository or the project documents could not be accessed. We exclude these eight projects from the dataset to ensure a fair comparison. The

Table 2. The number of entities in the code knowledge graph

Project	Entity						Relation							
	Class	Interface	Property	Method	Parameter	Variable	Contain Member	Contain Variable	Contain Parameter	Inheritance	Instantiation	Invocation	Return Type	Implementation
Teiid	5123	521	9434	26560	17369	37978	34919	37978	17369	1111	60337	35461	25076	631
Hornetq	3095	479	6418	16859	10615	25787	22844	25787	10615	1008	42352	17362	16409	691
Seam2	1571	142	1938	5998	4120	3307	7354	3307	4120	327	9311	5422	5397	235
Weld	4089	371	2849	7208	4350	3043	9668	3043	4350	525	9867	5802	6748	698
Drools	4285	401	10914	18926	14260	24295	29321	24295	14260	1236	46965	19740	18194	1269
Derby	3112	257	15786	27417	30307	35081	42821	35081	30307	1353	77548	28947	26802	683
Log4j2	1748	473	2637	5038	3486	5471	7125	5471	3486	334	10598	5948	4344	221
Total	23023	2644	49976	108006	84507	134962	154052	134962	84507	5894	256978	118682	102970	4428

dataset includes both bug reports and the corresponding changesets submitted to resolve them. We collect project documents such as version control documents from these projects to construct the knowledge graph. We also collect commit logs and source code from all Git repositories of the seven projects for constructing the code knowledge graph. As shown in Table 1, we provide detailed information about the seven projects. A total of more than 6,000 bug reports are used to evaluate the effectiveness of our model. To explore the granular impact of different changesets, we further split the changesets into commit-level, file-level, and hunk-level code changes. As a result, we evaluate changesets at three different levels of granularity: commits, files, and hunks.

4.2.2 Datasets for knowledge graphs. Based on the methods described in Section 3.3, this paper constructs corresponding code knowledge graphs and text knowledge graphs for each project in the bug localization dataset. To ensure fairness in the bug localization experiments and avoid introducing future knowledge during the construction of the knowledge graphs, we select the code repository and project documents before the bug report appears as the data source for constructing the knowledge graph. Specifically, we select the creation time of the bug report at 50% (*i.e.*, the training set) as the reference time point for constructing knowledge graphs. Then we select the software code repository and project documents of the projects that existed before the reference time point as the original data for constructing knowledge graphs.

For the construction of the code knowledge graph, the "Master" and "Main" branches from the official code repositories were used as the primary data sources. The code files extracted for the knowledge graph included the main systems, subsystems, and test cases for each project. The resulting code knowledge graph encompasses 403,118 code entities and 862,473 code relationships. The statistical details of the code knowledge graphs for different projects are provided in Table 2.

To ensure the accuracy of the text knowledge graphs, project documents from the official websites of each project are used. These documents included user manuals, developer guides, technical design documents, and API references. The constructed text knowledge graph contains 5,780 knowledge entities and 20,604 corresponding relationships. An overview of the text knowledge graphs for different projects is shown in Table 1.

4.3 Baselines

To evaluate the performance of the proposed knowledge-enhanced LLM in bug localization, we selected a traditional IR-based model, Locus, as the baseline. In addition, we included advanced LLMs as the baselines, such as GPT-2, CodeT5, and GraphCodeBERT. Finally, we selected one of the most advanced commercial models developed by OpenAI, called *text-embedding-ada-002*, as another baseline.

Locus [47]: Locus is a traditional text similarity-based bug localization approach. Based on the VSM model, it locates suspicious code snippets at a more granular level of bug localization by using similarity scores between bug reports and changesets.

GPT-2 [32]: GPT-2 is a large-scale language model based on the Transformer architecture developed by OpenAI. It uses self-attention mechanisms to capture long-range dependencies in text data. GPT-2 can be adapted to a wide range of downstream tasks through fine-tuning. Based on the vast WebText dataset of 8 million web pages totaling 40GB, GPT-2 shows remarkable performance. Based on the vast WebText dataset of 8 million web pages totaling 40GB, GPT-2 shows remarkable performance.

GraphCodeBERT [11]: GraphCodeBERT is a natural language processing model that is specifically designed for a variety of code-related tasks. The technique builds on BERT and incorporates structures to understand programming language. GraphCodeBERT can effectively handle the interaction between source code and natural language by combining the grammatical structure of the code and natural language description. It leverages graph structures, constructing AST and data flow graphs to capture information about code structure.

CodeT5 [46]: CodeT5 is a pre-trained encoder-decoder Transformer model that leverages the code semantics conveyed from developer-assigned identifiers. It is based on the T5 architecture and has been pre-trained on data sets from a wide range of programming languages. CodeT5 uses a novel identifier-aware pre-training target that is able to capture information about the code structure and key tag types from the code. In this way, it comprehends programming language semantics well and can be applied to various downstream tasks. We evaluated the effectiveness of multiple baseline models, including both CodeT5 and CodeT5+, on our specific datasets. The experimental results indicate that CodeT5 consistently outperforms CodeT5+ on all evaluation metrics. Given these findings, we chose CodeT5 as the baseline instead of CodeT5+ to ensure that our evaluation reflects the most effective model for bug localization.

text-embedding-ada-002 [29]: *text-embedding-ada-002* is an advanced neural embedding model developed by OpenAI, designed for text representation and similarity tasks [54]. The model generates dense vector representations of textual inputs using a transformer-based architecture, enabling efficient natural language processing (NLP). *text-embedding-ada-002* exhibits state-of-the-art performance across various embedding benchmarks, demonstrating robustness in tasks such as semantic search, clustering, recommendation systems, and knowledge retrieval.

4.4 Evaluation Metrics

To evaluate the performance of the model, we use the following metrics that are widely used in bug localization: *i.e.*, *MRR* (Mean Reciprocal Rank), *MAP* (Mean Average Precision), and $Top@N$.

MRR (Mean Reciprocal Rank) measures a process that produces a list of possible responses to a query. The reciprocal rank of a query response is equal to the multiplicative inverse of the rank of the first correct answer. The mean reciprocal rank is the average of the reciprocal ranks of the results for a sample of queries Q:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{3}$$

MAP (Mean Average Precision) measures a single-figure quality of IR when a query may contain multiple relevant files. The AvgP (Average Precision of a single query) is calculated as the average of the precision values obtained for a single query, as follows:

$$\text{Avg}\, P_i = \sum_{i=1}^{M} \frac{P(i) \times \text{pos}(i)}{\text{number of positive instances}} \tag{4}$$

Where i represents the rank, M represents the number of instances retrieved, $P(i)$ is defined as the precision at the given cut-off rank i, and $pos(i)$ indicates the relevance of the instance.

Table 3. Comparison of KEPT with baselines for bug localization.

Project	Technique	*Commits*					*Files*					*Hunks*				
		MRR	**MAP**	**Top@1**	**Top@3**	**Top@5**	**MRR**	**MAP**	**Top@1**	**Top@3**	**Top@5**	**MRR**	**MAP**	**Top@1**	**Top@3**	**Top@5**
Teiid	Locus	0.394	0.371	**0.300**	0.439	0.515	0.389	0.226	0.297	0.433	0.505	0.387	0.197	0.296	0.431	0.501
	GPT-2	0.277	0.257	0.183	0.297	0.364	0.267	0.157	0.183	0.286	0.353	0.261	0.133	0.181	0.279	0.336
	GraphCodeBERT	0.287	0.270	0.172	0.317	0.401	0.279	0.169	0.178	0.308	0.389	0.271	0.143	0.169	0.305	0.376
	CodeT5	0.326	0.305	0.209	0.376	0.444	0.315	0.190	0.206	0.367	0.426	0.312	0.159	0.207	0.365	0.418
	text-embedding-ada-002	0.380	0.359	0.274	0.420	0.491	0.371	0.223	0.274	0.409	0.472	0.368	0.184	0.274	0.407	0.460
	KEPT	**0.422**	**0.401**	0.297	**0.485**	**0.565**	**0.414**	**0.261**	**0.299**	**0.472**	**0.540**	**0.409**	**0.218**	**0.297**	**0.467**	**0.531**
Hornetq	Locus	0.359	0.333	0.287	0.435	0.443	0.356	0.194	0.287	0.426	0.443	0.353	0.169	0.287	0.417	0.426
	GPT-2	0.374	0.345	0.261	0.391	0.470	0.353	0.228	0.261	0.374	0.426	0.348	0.181	0.261	0.374	0.400
	GraphCodeBERT	0.351	0.335	0.243	0.365	0.478	0.331	0.222	0.243	0.322	0.461	0.321	0.187	0.243	0.313	0.400
	CodeT5	0.361	0.343	0.235	0.400	0.504	0.334	0.230	0.235	0.365	0.417	0.326	0.188	0.235	0.357	0.409
	text-embedding-ada-002	0.561	0.527	0.452	**0.617**	**0.670**	**0.549**	0.366	0.452	**0.600**	**0.652**	**0.546**	0.286	0.452	**0.591**	**0.643**
	KEPT	**0.567**	**0.535**	**0.470**	0.591	**0.670**	0.548	**0.399**	**0.470**	0.583	0.609	0.544	**0.324**	**0.470**	0.583	0.609
Seam2	Locus	0.407	0.389	0.339	0.453	0.482	0.400	0.326	0.333	0.442	0.472	0.396	0.293	0.333	0.442	0.469
	GPT-2	0.339	0.326	0.233	0.374	0.450	0.328	0.260	0.233	0.360	0.425	0.321	0.218	0.233	0.352	0.398
	GraphCodeBERT	0.266	0.251	0.157	0.306	0.363	0.253	0.200	0.157	0.287	0.336	0.246	0.167	0.157	0.274	0.333
	CodeT5	0.356	0.340	0.236	0.393	0.482	0.340	0.277	0.236	0.369	0.450	0.333	0.233	0.236	0.363	0.434
	text-embedding-ada-002	0.492	0.474	0.377	0.550	0.640	0.482	0.395	0.377	0.534	0.607	0.478	0.339	0.377	0.528	0.588
	KEPT	**0.538**	**0.520**	**0.434**	**0.577**	**0.664**	**0.531**	**0.448**	**0.436**	**0.572**	**0.645**	**0.522**	**0.388**	**0.431**	**0.564**	**0.618**
Weld	Locus	0.293	0.261	0.195	0.346	0.401	0.299	0.128	0.210	0.339	0.389	0.294	0.123	0.210	0.339	0.385
	GPT-2	0.304	0.277	0.187	0.342	0.405	0.288	0.169	0.187	0.311	0.385	0.283	0.141	0.187	0.307	0.377
	GraphCodeBERT	0.280	0.254	0.179	0.307	0.362	0.262	0.172	0.179	0.272	0.346	0.258	0.149	0.179	0.265	0.339
	CodeT5	0.307	0.277	0.198	0.323	0.405	0.286	0.186	0.198	0.296	0.350	0.280	0.155	0.198	0.288	0.335
	text-embedding-ada-002	0.445	0.400	**0.331**	0.502	0.576	0.432	0.265	**0.331**	0.482	**0.553**	0.429	0.222	**0.331**	**0.482**	**0.549**
	KEPT	**0.456**	**0.421**	0.327	**0.510**	**0.603**	**0.439**	**0.284**	0.327	**0.494**	**0.553**	**0.432**	**0.250**	0.327	0.475	0.533
Drools	Locus	0.163	0.151	0.123	0.186	0.208	0.158	0.085	0.121	0.181	0.201	0.153	0.073	0.121	0.166	0.193
	GPT-2	0.166	0.152	0.085	0.164	0.231	0.155	0.084	0.085	0.156	0.219	0.150	0.065	0.085	0.153	0.204
	GraphCodeBERT	0.187	0.173	0.101	0.184	0.262	0.175	0.096	0.103	0.169	0.231	0.169	0.074	0.101	0.163	0.226
	CodeT5	0.223	0.209	0.123	0.248	0.321	0.212	0.116	0.123	0.231	0.304	0.206	0.088	0.121	0.226	0.292
	text-embedding-ada-002	0.301	0.283	0.196	0.321	0.414	0.290	0.153	0.196	0.314	0.380	0.281	0.115	0.196	0.299	0.367
	KEPT	**0.339**	**0.318**	**0.221**	**0.382**	**0.462**	**0.328**	**0.176**	**0.221**	**0.365**	**0.435**	**0.322**	**0.134**	**0.221**	**0.359**	**0.422**
Derby	Locus	0.370	0.334	0.287	0.412	0.463	0.365	0.229	0.293	0.395	0.444	0.360	0.185	0.296	0.379	0.430
	GPT-2	0.291	0.258	0.194	0.311	0.386	0.280	0.177	0.195	0.298	0.362	0.273	0.133	0.195	0.289	0.349
	GraphCodeBERT	0.229	0.207	0.123	0.257	0.335	0.212	0.140	0.117	0.238	0.318	0.208	0.107	0.123	0.223	0.299
	CodeT5	0.320	0.289	0.230	0.345	0.406	0.307	0.201	0.230	0.331	0.381	0.301	0.156	0.229	0.321	0.371
	text-embedding-ada-002	0.336	0.304	0.234	0.381	0.441	0.327	0.210	0.234	0.371	0.417	0.319	0.156	0.234	0.355	0.401
	KEPT	**0.427**	**0.388**	**0.310**	**0.487**	**0.565**	**0.419**	**0.280**	**0.313**	**0.470**	**0.543**	**0.412**	**0.218**	**0.311**	**0.464**	**0.525**
Log4j2	Locus	0.437	0.371	0.336	0.482	0.573	0.439	0.207	0.355	0.473	0.550	0.437	0.206	0.359	0.468	0.536
	GPT-2	0.408	0.352	0.282	0.468	0.550	0.397	0.239	0.277	0.464	0.536	0.388	0.210	0.277	0.450	0.523
	GraphCodeBERT	0.401	0.352	0.264	0.455	0.573	0.387	0.239	0.264	0.441	0.527	0.381	0.209	0.264	0.441	0.518
	CodeT5	0.454	0.399	0.323	0.527	0.595	0.439	0.271	0.323	0.495	0.559	0.434	0.245	0.323	0.495	0.541
	text-embedding-ada-002	0.567	0.488	0.450	0.627	0.723	**0.561**	0.321	0.450	0.618	0.691	0.554	0.290	0.450	0.600	0.677
	KEPT	**0.600**	**0.517**	**0.477**	**0.686**	**0.782**	0.588	**0.375**	**0.473**	**0.664**	**0.741**	**0.583**	**0.338**	**0.477**	**0.636**	**0.732**
Average	Locus	0.346	0.316	0.267	0.393	0.441	0.344	0.199	0.271	0.384	0.429	0.340	0.178	0.272	0.378	0.420
	GPT-2	0.308	0.281	0.204	0.335	0.408	0.295	0.188	0.203	0.321	0.387	0.289	0.154	0.203	0.315	0.370
	GraphCodeBERT	0.286	0.263	0.177	0.313	0.396	0.271	0.177	0.177	0.291	0.373	0.265	0.148	0.177	0.283	0.356
	CodeT5	0.335	0.309	0.222	0.373	0.451	0.319	0.210	0.222	0.351	0.412	0.313	0.175	0.221	0.345	0.400
	text-embedding-ada-002	0.440	0.405	0.331	0.488	0.565	0.430	0.276	0.331	0.475	0.539	0.425	0.227	0.331	0.466	0.526
	KEPT	**0.478**	**0.443**	**0.362**	**0.531**	**0.616**	**0.467**	**0.318**	**0.363**	**0.517**	**0.581**	**0.461**	**0.267**	**0.362**	**0.507**	**0.567**

[1] The higher the value of the MAP, MRR, and Top@N, the better.

$$MAP = \frac{1}{|Q|} \sum_{i=1}^{|Q|} AvgP_{Q_i} \tag{5}$$

Top@N measures the percentage of bugs whose associated files are detected in top N (N=1,2,3,...) of the returned suspicious list of code files. The higher values indicate less effort required by developers to locate the bug and, thus, better performance.

In terms of MAP, MRR, and $Top@N$, a higher value is better.

5 Results

5.1 Effectiveness of KEPT (RQ1)

Table 3 presents the evaluation results for the effectiveness of KEPT, with the best results highlighted in bold. The evaluation results indicate that KEPT achieves almost all the best results for five metrics at three levels of changesets. Taking the commit-level as an example, KEPT shows the best performance on average across five metrics, achieving a 35.0% to 40.3% improvement over Locus. In comparison with CodeT5, the highest-performing non-commercial LLM, KEPT achieves an average improvement of 36.6% to 63.1%. Compared with commercial LLM, *text-embedding-ada-002*, KEPT achieves an average improvement of 7.8% to 17.4%.

The results of our approach show significant improvements over all the baselines at three levels of granularity. In Table 3, all LLMs achieve better results at the coarser-grained commits level compared to the finer-grained hunks level. For instance, CodeT5 achieved an MRR of 0.335 at the commits level, 0.319 at the files level, and 0.313 at the hunks level. Notably, KEPT exhibits a more substantial improvement at the finer-grained levels. Using MRR as an indicator, KEPT outperformed CodeT5 by 42.56% at the commits level, 46.30% at the files level, and 47.09% at the hunks level. We

Table 4. Comparison of the results *with* and *without* introducing knowledge graph.

Approach	Metric	Teiid	Hornetq	Seam2	Weld	Drools	Derby	Log4j2	Average
NoKG	MRR	0.383	0.486	0.509	0.398	0.267	0.393	0.560	0.428
	MAP	0.209	0.305	0.382	0.243	0.118	0.205	0.330	0.256
	Top@1	0.282	0.400	0.417	0.288	0.184	0.297	0.450	0.331
	Top@3	0.427	0.539	0.550	0.463	0.284	0.437	0.623	0.475
	Top@5	0.491	0.600	0.626	0.510	0.349	0.490	0.691	0.537
TextKG	MRR	0.395	0.481	0.518	0.406	0.256	0.400	**0.585**	0.434
	MAP	0.212	0.305	0.388	**0.252**	0.111	0.213	0.335	0.259
	Top@1	0.291	0.383	0.431	0.300	0.159	0.301	**0.482**	0.335
	Top@3	0.441	0.530	0.566	0.436	0.291	0.443	0.645	0.479
	Top@5	0.517	0.583	0.615	0.498	0.350	0.504	0.709	0.539
CodeKG	MRR	0.391	0.508	**0.532**	0.401	0.294	0.403	0.578	0.444
	MAP	0.209	0.297	**0.389**	0.243	0.126	0.213	0.332	0.258
	Top@1	0.285	0.409	**0.444**	0.296	0.203	0.305	0.477	0.346
	Top@3	0.437	0.557	**0.569**	0.440	0.314	0.454	0.636	0.487
	Top@5	0.509	0.609	**0.623**	0.502	0.382	0.506	0.686	0.545
TextKG+ CodeKG	MRR	**0.409**	**0.544**	0.522	**0.432**	**0.322**	**0.412**	0.583	**0.461**
	MAP	**0.218**	**0.324**	0.388	0.250	**0.134**	**0.218**	**0.338**	**0.267**
	Top@1	**0.297**	**0.470**	0.431	**0.327**	**0.221**	**0.311**	0.477	**0.362**
	Top@3	**0.467**	**0.583**	0.564	**0.475**	**0.359**	**0.464**	**0.636**	**0.507**
	Top@5	**0.531**	**0.609**	0.618	**0.533**	**0.422**	**0.525**	**0.732**	**0.567**

attribute this enhanced performance to the incorporation of a knowledge graph, which allows the model to reason more effectively, especially at the hunks level where the context is often lacking.

To further validate the evaluation results, we use the Wilcoxon signed-rank test on the five metrics. The Wilcoxon signed-rank test is a non-parametric hypothesis test used for determining whether results are significantly different between groups, which is used in bug localization [3] and other fields [21]. A *p-value* result less than 0.05 indicates a statistically significant difference between KEPT and the baselines with 95% confidence. According to the test results, KEPT outperforms Locus, GPT-2, GraphCodeBERT, CodeT5, and *text-embedding-ada-002* on average across all metrics at three granularities, showing statistically significant differences with 95% confidence.

For fine-tuning, KEPT required 355 minutes, making it the fastest approach among all the LLMs evaluated. GPT-2 followed closely, taking 364 minutes. CodeT5 requires 468 minutes, while GraphCodeBERT requires 479 minutes.

Overall, KEPT achieves average improvements on the five metrics by 33.2% to 59.5% under three levels of changesets when compared with the classical method, Locus. Compared to the best LLM, CodeT5, KEPT achieved average improvements of 36.6% to 63.7%. The Wilcoxon signed-rank test indicated that KEPT is statistically significant compared to all the baselines on average across all metrics at 95% confidence. The results indicate that our method, KEPT, is more effective in capturing the semantics of code and text, making the proposed approach advantageous for bug localization.

Answer to RQ1: *KEPT outperforms the state-of-the-art models for bug localization, especially achieving 7.8% to 78.6% better results on average across five metrics with a statistically significant difference with a 95% confidence level.*

5.2 Effectiveness of Knowledge Graph (RQ2)

To evaluate the impact of knowledge graphs on model performance, we conduct ablation experiments at three granularity of changesets. Table 4 presents the evaluation results for the effectiveness of the knowledge graph.

Taking the commit-level results as an example, the best performance is obtained by introducing both the code knowledge graph and the text knowledge graph (TextKG+CodeKG). TextKG+CodeKG achieves the best results on average, with MRR at 0.461, MAP at 0.267, Top@1 at 0.362, Top@3 at 0.507, and Top@5 at 0.567, respectively. Compared to scenarios without knowledge graph, TextKG+CodeKG achieved an average performance improvement of 6.70%, 6.24%, 9.37%, 4.93%, and 4.44% across MRR, MAP, Top@1, Top@3, and Top@5, respectively.

Table 5. Comparison of the results *with* and *without* soft-position embedding and visible matrix.

Approach	Metric	Teiid	Hornetq	Seam2	Weld	Drools	Derby	Log4j2	Average
BERT-w/o	MRR	0.124	0.169	0.173	**0.190**	**0.097**	0.137	**0.240**	0.161
	MAP	0.061	0.100	0.103	0.100	**0.042**	0.062	0.120	0.084
	Top@1	0.060	0.087	0.092	**0.117**	**0.047**	0.073	0.145	0.089
	Top@3	0.132	0.209	**0.195**	0.198	**0.098**	0.149	**0.268**	**0.178**
	Top@5	0.166	0.261	**0.236**	0.249	**0.128**	0.193	**0.345**	0.225
CodeBERT-w/o	MRR	0.292	0.398	0.386	0.307	**0.228**	0.260	0.457	0.333
	MAP	0.148	0.233	0.284	0.177	0.097	0.143	0.261	0.192
	Top@1	0.200	0.296	0.276	0.191	**0.146**	0.162	0.336	0.230
	Top@3	0.322	0.443	0.439	**0.362**	**0.259**	0.306	0.509	0.377
	Top@5	0.381	0.487	0.512	**0.432**	**0.306**	0.357	0.595	0.439
KEPT-w/o	MRR	0.383	0.486	0.509	0.398	0.267	0.393	0.560	0.428
	MAP	0.209	0.305	0.382	0.243	0.118	0.205	0.330	0.256
	Top@1	0.282	0.400	0.417	0.288	0.184	0.297	0.450	0.331
	Top@3	0.427	0.539	0.550	0.463	0.284	0.437	0.623	0.475
	Top@5	0.491	0.600	**0.626**	0.510	0.349	0.490	0.691	0.537
BERT-w/SPVM	MRR	**0.138**	**0.228**	**0.180**	0.189	0.089	**0.143**	0.234	**0.172**
	MAP	**0.069**	**0.118**	**0.111**	**0.102**	0.037	**0.064**	**0.120**	**0.089**
	Top@1	**0.074**	**0.165**	**0.119**	0.109	0.040	**0.080**	**0.159**	**0.107**
	Top@3	**0.133**	**0.217**	0.176	**0.218**	0.081	**0.151**	0.245	0.174
	Top@5	**0.184**	**0.304**	0.228	**0.265**	0.115	**0.194**	0.314	**0.229**
CodeBERT-w/SPVM	MRR	**0.303**	**0.450**	**0.428**	**0.312**	0.221	**0.273**	**0.502**	**0.356**
	MAP	**0.155**	**0.241**	**0.307**	**0.180**	**0.100**	**0.144**	**0.277**	**0.201**
	Top@1	**0.204**	**0.383**	**0.331**	**0.206**	0.136	**0.171**	**0.395**	**0.261**
	Top@3	**0.339**	**0.470**	**0.472**	0.350	0.241	**0.314**	**0.568**	**0.393**
	Top@5	**0.412**	**0.522**	**0.534**	0.412	0.304	**0.384**	**0.632**	**0.457**
KEPT-w/SPVM	MRR	**0.409**	**0.544**	**0.522**	**0.432**	**0.322**	**0.412**	**0.583**	**0.461**
	MAP	**0.218**	**0.324**	**0.388**	**0.250**	**0.134**	**0.218**	**0.338**	**0.267**
	Top@1	**0.297**	**0.470**	**0.431**	**0.327**	**0.221**	**0.311**	**0.477**	**0.362**
	Top@3	**0.467**	**0.583**	**0.564**	**0.475**	**0.359**	**0.464**	**0.636**	**0.507**
	Top@5	**0.531**	**0.609**	0.618	**0.533**	**0.422**	**0.525**	**0.732**	**0.567**

[1] SPVM is an acronym for soft-position embedding and visible matrix.

At the commit-level granularity, enhancing the LLM using the text knowledge graph alone (TextKG) and using the code knowledge graph alone (CodeKG) achieves better results than those without using the knowledge graph. In most cases, CodeKG performs better than TextKG, which might be attributed to the number of CodeKG additions in most projects greater than those of TextKG. A possible reason is that, compared to historical documentation, historical code repositories have constructed a more extensive knowledge graph. The code knowledge graph consists of 403,118 code entities and 862,473 code relationships, along with 5,780 knowledge entities and 20,604 corresponding relationships.

Again, a Wilcoxon signed-rank test is performed on the results of the ablation experiments at three levels of changesets to verify their statistical significance. The results show that the differences are statistically significant with a *p-value* less than 0.05, indicating a significant improvement without introducing knowledge graphs.

For fine-tuning, NoKG takes 291 minutes, TextKG takes 308 minutes, and CodeKG takes 328 minutes. TextKG+CodeKG together require 355 minutes, which is only a 22% increase over NoKG.

In general, the performance of the LLM enhanced by introducing the code knowledge graph and text knowledge graph has been improved by 4.3% to 9.5% when compared to the LLM without introducing a knowledge graph. With 95% confidence, the difference between the knowledge-enhanced LLM and the LLM without using a knowledge graph is statistically significant.

Answer to RQ2: *The model KEPT enhanced with knowledge graph performs better than the model without knowledge graph enhancement. With a 95% confidence level, there is a significant difference between introducing knowledge graphs and not introducing one.*

5.3 Effectiveness of Soft-Position Embedding and Visible Matrix (RQ3)

To evaluate the impact of our proposed model structure, we selected two widely used encoder-structured models, BERT and CodeBERT, which have structures similar to KEPT.

The evaluation results indicate that our proposed model structure, which uses Soft-Position embedding and Visible Matrices (SPVM) for representation, generally achieves better performance.

In comparison with BERT -w/o, BERT -w/SPVM achieved an average improvement of 8.3% to 18.9%. The evaluation results show that the model structure proposed in this paper achieves the best results for almost all projects. The results indicate that the bug localization technique proposed in this paper can be extended to be a general and effective structure for various advanced LLMs.

Answer to RQ3: *Three LLMs (i.e., BERT, CodeBERT, KEPT) that use the model structure proposed in this study perform better than those that do not use it, demonstrating the effectiveness of this model structure.*

6 Threats to Validity

In this section, we discuss the potential threats to the validity of the study and our efforts to mitigate their impacts.

Internal validity. We identify two threats to internal validity for this study. The first concern is that our experimental results depend on the knowledge graph construction tool we use. To mitigate this threat, we selected a popular tool that achieves the highest precision and recall [7, 24]. The second threat to internal validity is potential errors in implementing our approach and the baselines. We mitigated this threat by using the original source code and hyperparameter settings that were shared by the baselines [11, 32, 46, 47]. In addition, we carefully check the datasets and source code used in this study to make sure they are correct.

External validity. The threat to external validity of this study arises from the ability to generalize our research to the external. KEPT is implemented and evaluated on seven OSS projects, and the performance of KEPT on commercial projects is unknown. We selected seven projects that contained more than 6,000 bugs to possibly reduce this external validity. In addition, the main structure of KEPT is based on UniXcoder, which is used in other fields.

Construct validity. The construct validity of a study is derived from the metrics that we select, that is, making sure that the chosen metrics are compatible with the study's approach and design to generate reliable results. As a result, we use widely used evaluation metrics in bug localization, such as MAP, MRR, and Top@N, in an effort to minimize this threat.

7 Related Work

IR techniques can significantly reduce the developer's burden of debugging [26] and maintaining software [15, 35], and it is one of the most widely researched techniques for automatic bug localization [31, 53]. While IR-based methods form a substantial part of bug localization research, alternative approaches have also been explored. One such line of research focuses on the coverage of failing and passing tests. A notable recent contribution in this area is Fonte, proposed by An et al. [1]. Fonte is an efficient and accurate bug-inducing commit identification technique that uniquely relies solely on test coverage. It innovatively combines fault localization with bug-inducing commit identification to rank commits based on the suspiciousness of the code elements they modify. It is important to note that fault localization and IR-based methods are not mutually exclusive; in fact, they can complement each other effectively. However, due to the constraints of our dataset, which lacks the failed test execution and commit history required by Fonte, we are unable to include it as a baseline in our study.

Given these considerations, our review of related work will primarily focus on relevant IR-based methods and research that has informed the design of our approach.

7.1 Traditional Machine Learning Approaches

Machine learning methods are commonly used in bug localization since they can acquire and integrate knowledge from large-scale observations, and are able to improve with the acquisition of new information. Early research in bug localization relied on machine learning methods, which can process large-scale observations and improve with new information Using the bag of words, early researchers calculated the similarity between the bug report and the source code by comparing the frequency of the same term in them. To calculate the similarity between the query and the source code file, Robertson et al. proposed a method based on TF-IDF, which combines the reverse document of the query word in the corpus with the word frequency in the source code file [34]. In later work, Wang et al. [44] proposed several vector space models based on different forms of TF and IDF calculation. The experimental results show that the combined method outperforms the standard VSM model.

Gore et al. [8] proposed a hybrid model that combines VSM and N-gram for bug localization. Compared with traditional methods, the hybrid model outperforms some existing state-of-the-art techniques for bug localization. Several bug localization models have also been developed, including DHbPd [37], BLUiR [36], BRTracer [49], Amalgam [43], LOCUS [47], and others, that combine additional information on software project versions and code changes to locate bugs.

7.2 Deep Learning Approaches

Deep learning has gradually gained traction after machine learning for bug localization. Generally, these models rely on deep learning models to extract semantics from bug reports and source code, and to match the similarity between them. In deep learning-based bug localization, feature extraction is often at the core of the process [58]. Deep learning models are primarily used to extract complex semantic relationships between source code and bug reports. The three most common deep learning models are CNN, RNN, and DNN. Huo et al. [14] proposed NP-CNN, a CNN-based deep learning network for bug localization that uses both lexical and program structure information to learn unified features from natural language and source code for automatically locating buggy code. Experimental results on a wide range of software projects show that NP-CNN significantly outperforms state-of-the-art methods in bug localization. Wang et al. [40] proposed a Multi-Dimension Convolutional Neural Network (MD-CNN) model for bug localization based on bug reports. According to the evaluation results, Wang et al. found that MD-CNN outperformed existing bug localization techniques. Yang et al. [52] propose a hybrid RNN-attention model called MRAM, which combines bug-fixing features and method-structured features to explore their relevance. Experimental results indicate that the model performs significantly better than the baseline. Lam et al. [18] combined a DNN with rVSM, using rVSM to collect features of textual similarity between bug reports and source files. Experimental results indicate that combining a DNN with rVSM performs significantly better than the baselines. Only one study proposed a knowledge graph-based approach based on structure features and applied hyperbolic attention embedding to get the link prediction scores [50]. This approach has three main limitations. Firstly, it represents bug report IDs and code IDs as nodes in a knowledge graph, which results in a significant loss of textual and code information. Second, the method relies solely on syntactic information from the text and code, overlooking the rich semantic information. Finally, the approach does not incorporate external knowledge that is often available in projects, such as technical documentation, which can assist the model to better learn the information contained in both bug reports and code. As the replication package for this work was not provided, we were not able to reproduce the results for comparison. We chose Locus as the baseline because it provides a replication package and is a traditional approach for bug localization based on changesets.

7.3 LLM-based Approaches

LLMs have gradually been applied to bug localization in recent years, with high performance in many fields. Transformer-based neural network architecture has become dominant, achieving promising performance in many natural language processing tasks and code representations. Zhu et al. [58] proposed a COOBA model that embeds the text in bug reports using an unsupervised learning algorithm, GloVe, and encodes them using bidirectional LSTMs. The code file is converted into AST and then embedded by GloVe. Finally, they extracted private and public features of the projects. Lin et al. [23] propose a novel framework for generating traceable links between source code and natural language artifacts. The experimental results show that their model outperforms the VSM model in stabilization among the three OSS projects. Liang et al. [22] proposed the FLIM model that uses an LLM, CodeBERT, and fine-tunes it for code search tasks. Furthermore, they employed the FLIM model to locate bugs using semantic features extracted from code files and bug reports. Asudani et al. [2] embedded code files using a LLM, GloVe, to improve DeepLoc, and embedded bug reports with sent2vec. The results indicate the effect of extracting features based on different embedding methods.

Several studies have proposed enhancing LLMs with knowledge in natural language processing [9, 13, 25, 41, 45]. The success of these studies inspires us to focus on domain-specific data. In contrast to these studies, we focus on bug reports and changesets that are relevant to bug localization. Bug reports and changesets often lack grammatical structure, making understanding challenging. Different from them, we introduce domain knowledge to enhance LLMs. We also incorporate soft-position embedding, visible matrices, and mask Transformer to represent domain knowledge.

8 Conclusion and Future Work

In this paper, we propose Kept, a knowledge-enhanced LLM designed to improve bug localization performance. First, we construct the text knowledge graph and the code knowledge graph using project documents and source code, respectively, to represent external domain knowledge. Second, we use soft-position embedding, visible matrices, and mask Transformer to represent this domain knowledge. Finally, we pre-train the model on the tasks aligned with the target task before fine-tuning it for downstream tasks. The evaluation results indicate that Kept significantly improves bug localization performance. As compared to the traditional method, Locus, Kept improves the performance of evaluation metrics by 33.2% to 59.5% in bug localization. As compared to the best-performing LLM in baselines, CodeT5, Kept shows a significant improvement of 36.6% to 63.7%. The Wilcoxon signed rank test confirms that Kept exhibits statistically significant differences from all baseline methods. Therefore, the evaluation results significantly demonstrate the effectiveness of Kept for bug localization, as well as confirm the positive impact of our approach by leveraging the knowledge enhanced LLM for bug localization. In the future, we intend to continuously reinforce bug localization by improving the knowledge-enhanced LLM. In addition, we plan to evaluate our model in a wider range of scenarios and projects.

9 Data Availability

The datasets and source code used in this study are open source, and can easily be accessed by anyone. The datasets and source code are released at: https://github.com/keptmodel/KEPT.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No.62072227, No.62202219, No.62302210), the Jiangsu Provincial Key Research and Development Program (No.BE2 021002-2), the Natural Science Foundation of Jiangsu Province (No.BK20241195), and the Innovation Project and Overseas Open Project of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2024A18, ZZKT2024B07, KFKT2023A09, KFKT2023A10, KFKT2024A02, KFKT2024A13, KFKT2024A14).

References

[1] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)*. IEEE, Melbourne, Australia, 589–601.

[2] Deepak Suresh Asudani, Naresh Kumar Nagwani, and Pradeep Singh. 2023. Impact of word embedding models on text analytics in deep learning environment: a review. *Artificial Intelligence Review* 56, 9 (2023), 10345–10425.

[3] Dylan Callaghan and Bernd Fischer. 2023. Improving Spectrum-Based Localization of Multiple Faults by Iterative Test Suite Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*. ACM, Seattle, WA, USA, 1445–1457.

[4] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast Changeset-Based Bug Localization with BERT. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM, Pittsburgh, Pennsylvania, 946–957.

[5] Yali Du and Zhongxing Yu. 2023. Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. ACM, San Francisco, CA, USA, 579–591.

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). ACM, Online, 1536–1547.

[7] K Gashteovski, R Gemulla, and L Del Corro. 2017. MinIE: Minimizing facts in open information extraction. *Association for Computational Linguistics* (2017), 1–11.

[8] Alpa Gore, Siddharth Dutt Choubey, and Kopal Gangrade. 2016. Improved Bug Localization Technique Using Hybrid Information Retrieval Model. In *Proceedings of the 12th Distributed Computing and Internet Technology (ICDCIT'16)*, Nikolaj Bjørner, Sanjiva Prasad, and Laxmi Parida (Eds.). Springer, Bhubaneswar, India, 127–131.

[9] Jian Guan, Fei Huang, Zhihao Zhao, Xiaoyan Zhu, and Minlie Huang. 2020. A Knowledge-Enhanced Pretraining Model for Commonsense Story Generation. *Transactions of the Association for Computational Linguistics* 8 (2020), 93–108.

[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations (ICLR'21)*. 1–18.

[12] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[13] Linmei Hu, Zeyi Liu, Ziwang Zhao, Lei Hou, Liqiang Nie, and Juanzi Li. 2024. A Survey of Knowledge Enhanced Pre-Trained Language Models. *IEEE Transactions on Knowledge and Data Engineering* 36, 4 (2024), 1413–1430.

[14] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, New York, USA, 1606–1612.

[15] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2021. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1368–1380.

[16] Darryl Jarman, Jeffrey Berry, Riley Smith, Ferdian Thung, and David Lo. 2022. Legion: Massively Composing Rankers for Improved Bug Localization at Adobe. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3010–3024.

[17] Ashwini Jaya Kumar, Christoph Schmidt, and Joachim Köhler. 2017. A knowledge graph based speech interface for question answering systems. *Speech Communication* 92 (2017), 1–12.

[18] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17)*. IEEE, Buenos Aires, Argentina, 218–229.

[19] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4BL: Reproducibility Study on the Performance of IR-Based Bug Localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'2018)*. ACM, Amsterdam, Netherlands, 61–72.

[20] Yuxuan Lei, Jianxun Lian, Jing Yao, Mingqi Wu, Defu Lian, and Xing Xie. 2024. Aligning Language Models for Versatile Text-based Item Retrieval. In *Companion Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 935–938.

[21] Yue Li, Zhong Ren, Zhiqi Wang, Lanxin Yang, Liming Dong, Chenxing Zhong, and He Zhang. 2024. Fine-SE: Integrating Semantic Features and Expert Features for Software Effort Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)* (Lisbon, Portugal). ACM, Article 27, 12 pages.

[22] Hongliang Liang, Dengji Hang, and Xiangyu Li. 2022. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering* 27, 7 (2022), 1–26.

[23] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, Madrid, ES, 324–335.

[24] Edward Loper and Steven Bird. 2002. NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics (ETMTNLP '02) (ETMTNLP '02)*. ACM, Philadelphia, Pennsylvania, 63–70.

[25] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. 2024. KnowLog: Knowledge Enhanced Pre-trained Language Model for Log Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)*. ACM, Lisbon, Portugal, Article 32, 13 pages.

[26] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-Scale IR-Based Bug Localization: A Perspective from Facebook. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'21)*. IEEE, Madrid, ES, 188–197.

[27] 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 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, Singapore, Singapore, 672–683.

[28] Feifei Niu, Wesley K. G. Assunção, LiGuo Huang, Christoph Mayr-Dorn, Jidong Ge, Bin Luo, and Alexander Egyed. 2023. RAT: A Refactoring-Aware Traceability Model for Bug Localization. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. IEEE, Melbourne, Australia, 196–207.

[29] OpenAI. 2022. text-embedding-ada-002. https://platform.openai.com/docs/guides/embeddings.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Curran Associates Inc., Vancouver Canada, 1–12.

[31] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug Localization on Millions of Files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. ACM, Virtual Event, USA, 225–236.

[32] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. arXiv, 1–24.

[33] Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing Requirements and Traceability Information to Improve Bug Localization. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. ACM, Gothenburg, Sweden, 442–453.

[34] Stephen.E. Robertson and Karen. Spärck Jones. 1994. *Simple, proven approaches to text retrieval.* Technical Report. University of Cambridge, Computer Laboratory.

[35] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-Informed Oracle. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, Madrid, Spain, 436–447.

[36] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE'13)*. IEEE, Silicon Valley, CA, USA, 345–355.

[37] Bunyamin Sisman and Avinash C. Kak. 2012. Incorporating version histories in Information Retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*. IEEE, Zurich, Switzerland, 50–59.

[38] Tianxiang Sun, Yunfan Shao, Xipeng Qiu, Qipeng Guo, Yaru Hu, Xuanjing Huang, and Zheng Zhang. 2020. Co-LAKE: Contextualized Language and Knowledge Embedding. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING'20)*. ICCL, Barcelona, Spain (Online), 3660–3670.

[39] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and Named Entity Recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL'20)*. ACL, Online, 4913–4926.

[40] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. 2022. Multi-Dimension Convolutional Neural Network for Bug Localization. *IEEE Transactions on Services Computing* 15, 3 (2022), 1649–1663.

[41] Jianing Wang, Chengyu Wang, Minghui Qiu, Qiuhui Shi, Hongbin Wang, Jun Huang, and Ming Gao. 2022. KECP: Knowledge Enhanced Contrastive Prompting for Few-shot Extractive Question Answering. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. ACL, Abu Dhabi, United Arab Emirates, 3152–3163.

[42] Qing Wang, Lang Gou, Nan Jiang, Meiru Che, Ronghui Zhang, Yun Yang, and Mingshu Li. 2008. Estimating fixing effort and schedule based on defect injection distribution. *Software Process: Improvement and Practice* 13, 1 (2008), 35–50.

[43] Shaowei Wang and David Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. ACM, Hyderabad, India, 53–63.

[44] Shaowei Wang, David Lo, and Julia Lawall. 2014. Compositional Vector Space Models for Improved Bug Localization. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, Victoria, BC, Canada, 171–180.

[45] Xiaozhi Wang, Tianyu Gao, Zhaocheng Zhu, Zhengyan Zhang, Zhiyuan Liu, Juanzi Li, and Jian Tang. 2021. KEPLER: A Unified Model for Knowledge Embedding and Pre-trained Language Representation. *Transactions of the Association for Computational Linguistics* 9 (2021), 176–194.

[46] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv, 1–13.

[47] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*. ACM, Singapore, Singapore, 262–273.

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP'20)*. ACL, online, 38–45.

[49] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, Victoria, BC, Canada, 181–190.

[50] Xi Xiao, Renjie Xiao, Qing Li, Jianhui Lv, Shunyan Cui, and Qixu Liu. 2023. BugRadar: Bug localization by knowledge graph link prediction. *Information and Software Technology* 162 (2023), 1–13.

[51] Yichong Xu, Chenguang Zhu, Shuohang Wang, Siqi Sun, Hao Cheng, Xiaodong Liu, Jianfeng Gao, Pengcheng He, Michael Zeng, and Xuedong Huang. 2022. Human parity on commonsenseqa: Augmenting self-attention with external attention. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'22)*. 2762–2768.

[52] Shouliang Yang, Junming Cao, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2021. Locating Faulty Methods with a Mixed RNN and Attention Model. In *Proceedings of the 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, Madrid, Spain, 207–218.

[53] Zhou Yang, Jieke Shi, Shaowei Wang, and David Lo. 2021. IncBL: Incremental Bug Localization. In *Proceedings of the 36th International Conference on Automated Software Engineering (ASE'21)*. IEEE, Melbourne, Australia, 1223–1226.

[54] Jinsung Yoon, Yanfei Chen, Sercan Arik, and Tomas Pfister. 2024. Search-Adaptor: Embedding Customization for Information Retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL'24)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). ACM, Bangkok, Thailand, 12230–12247.

[55] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Xin Xia, and David Lo. 2023. Context-Aware Neural Fault Localization. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3939–3954.

[56] Zhuosheng Zhang, Hai Zhao, Masao Utiyama, and Eiichiro Sumita. 2023. Language Model Pre-training on True Negatives. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'23)*, Vol. 37. Washington DC, USA, 14002–14010.

[57] 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 *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, Zurich, Switzerland, 14–24.

[58] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. 2020. CooBa: Cross-project Bug Localization via Adversarial Transfer Learning. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI'20)*. ACM, Vienna, Austria, 3565–3571.

[59] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering* 46, 8 (2020), 836–862.

Received 2024-09-13; accepted 2025-04-01