

# Automatic Smart Contract Comment Generation via Large Language Models and In-Context Learning

Junjie Zhao<sup>a</sup>, Xiang Chen<sup>a,\*</sup>, Guang Yang<sup>b</sup>, Yiheng Shen<sup>a</sup>

<sup>a</sup>School of Information Science and Technology, Nantong University, Nantong, China

<sup>b</sup>College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

## Abstract

**Context:** Designing effective automatic smart contract comment generation approaches can facilitate developers' comprehension, boosting smart contract development and improving vulnerability detection. The previous approaches can be divided into two categories: fine-tuning paradigm-based approaches and information retrieval-based approaches.

**Objective:** However, for the fine-tuning paradigm-based approaches, the performance may be limited by the quality of the gathered dataset for the downstream task and they may have knowledge-forgetting issues, which can reduce the generality of the fine-tuned model. While for the information retrieval-based approaches, it is difficult for them to generate high-quality comments if similar code does not exist in the historical repository. Therefore we want to utilize the domain knowledge related to smart contract code comment generation in large language models (LLMs) to alleviate the disadvantages of these two types of approaches.

**Method:** In this study, we propose an approach SCCLLM based on LLMs and in-context learning. Specifically, in the demonstration selection phase, SCCLLM retrieves the top- $k$  code snippets from the historical corpus by considering syntax, semantics, and lexical information. In the in-context learning phase, SCCLLM utilizes the retrieved code snippets as demonstrations for in-context learning, which can help to utilize the related knowledge for this task in the LLMs. In the LLMs inference phase, the input is the target smart contract code snippet, and the output is the corresponding comment generated by the LLMs.

**Results:** We select a large corpus from a smart contract community Etherscan.io as our experimental subject. Extensive experimental results show the effectiveness of SCCLLM when compared with baselines in automatic evaluation and human evaluation. We also show the rationality of our customized demonstration selection strategy in SCCLLM by ablation studies.

**Conclusion:** Our study shows using LLMs and in-context learning is a promising direction for automatic smart contract comment generation, which calls for more follow-up studies.

**Keywords:** Smart Contract Comment, Large Language Model, In-Context Learning, Demonstration Selection, Information Retrieval

## 1. Introduction

Smart contracts [1, 2] are self-executing digital contracts running on blockchain technology. They automate, validate, and enforce agreement terms without intermediaries, offering transparency and security. However, Yang et al. [3] found that most of the smart contract code comments are unavailable, which can make it challenging for developers to understand the code's logic, purpose, and intended functionality. Moreover, smart contracts are susceptible to vulnerabilities and exploits. In a previous study, He et al. [4] found that 10% of the vulnerabilities were caused by code clones. If smart contract code lacks comments to explain potential risks and mitigate strategies, it becomes difficult to identify and address security vulnerabilities, which can increase the chances of hacks or attacks. To this end, it is necessary to automatically generate concise and

fluent natural language descriptions for smart contract codes. Based on the above analysis, we can find designing effective automatic comment generation approaches can facilitate developers' comprehension, boosting smart contract development and detecting vulnerabilities. However, when compared to source code summarization [5, 6, 7], the specific challenges associated with smart contract comment generation can be summarized as follows. First, smart contracts are typically written in Solidity. Understanding their codes requires specialized knowledge of these languages and the Ethereum platform. Second, given the implications of smart contracts in financial transactions and agreements, generating relevant and concise comments is crucial. Any misinterpretation could potentially have significant financial or legal implications. Finally, smart contracts encapsulate detailed business logic, which can be complex and multifaceted. This business logic should be adequately captured in the comments for a comprehensive understanding.

Until now, smart contract comment generation has received continuous attention. For example, Yang et al. [3] proposed the approach MMTrans. This approach learns the smart contract code representation from two heterogeneous modalities: SBT

\*Corresponding author

Email addresses: zhaojunjie225@gmail.com (Junjie Zhao), xchencs@ntu.edu.cn (Xiang Chen), novelyg@outlook.com (Guang Yang), yiheng.s@outlook.com (Yiheng Shen)

sequences [8] (i.e., global semantic information) and graphs (i.e., local semantic information) based on abstract syntax trees. Later MMTrans uses two encoders to extract the semantic information from these two modalities respectively and then uses a joint decoder to generate code comments. Later we [9] proposed an information retrieval-based approach CCGIR due to the widespread presence of code cloning in smart contract development. This approach employs CodeBert [10] to extract semantic vectors from the target code snippet and retrieve the top- $k$  code snippets based on their semantic similarity scores. Subsequently, it further considers the syntactic and lexical similarity of the code by combining these scores, which leads to the retrieval of the most similar code snippet. The comment from this retrieved code snippet is then reused for the target code snippet.

However, for the fine-tuning paradigm-based approaches [3], the performance may be limited by the quality of the gathered dataset for the downstream task. Moreover, they may have the issue of knowledge forgetting [11]. Specifically, if a pre-trained model is fine-tuned on a specific downstream task, the model might start to forget the general knowledge it acquired during the pretraining phase. This issue can limit the model’s generalization and versatility. While for the information retrieval-based approaches [9], it is difficult for them to generate high-quality comments if similar smart contract codes do not exist in the historical repository. To overcome the limitations of these two kinds of approaches for the smart contract comment generation, we want to leverage the emerging capabilities of large language models (LLMs), which have been pre-trained on vast amounts of data and possess a wealth of hidden domain knowledge, for automatic smart contract comment generation. However, merely utilizing LLMs without effectively leveraging their related domain knowledge may not yield optimal results. As highlighted in Section 4.3, our experiments reveal that directly employing LLMs in the zero-shot learning setting<sup>1</sup> fails to outperform baselines based on fine-tuning. To address this limitation, recent research has shown that the in-context learning paradigm offers a promising solution for harnessing the domain knowledge encapsulated within LLMs [12]. Specifically, given limited examples as the prompt, this paradigm can imitate the human ability to leverage prior knowledge (i.e., demonstration examples) to generate comments without parameter updating. However, the effectiveness of in-context learning heavily relies on the quality and quantity of demonstration examples provided [13, 14].

Based on the above research motivations, we propose a novel approach SCCLLM (Smart Contract Comment Generation via Large Language Models), which mainly contains three phases. In particular, we employ a customized two-phase retrieval strategy during the demonstration selection phase. This strategy allows us to retrieve the top- $k$  high-quality demonstration examples from a historical corpus, considering the semantic, syntactic, and lexical information of the code snippets. Subsequently, in the in-context learning phase, we leverage these retrieved

top- $k$  demonstrations to construct a customized prompt. By incorporating these demonstrations, we can utilize the knowledge related to smart contract comment generation within LLMs through in-context learning. Once the prompt is constructed, we proceed to the LLMs inference phase. Here, we directly utilize the interface of LLMs, providing the customized prompt along with the target smart contract code snippet as input. The output is then generated by LLMs, representing the corresponding comment for the given code snippet.

To evaluate the effectiveness of our proposed approach SCCLLM, we conduct extensive experiments on a dataset with 29,720  $\langle$ method, comment $\rangle$  pairs, which were gathered from 40,933 smart contracts in a smart contract community EtherScan<sup>2</sup>. We use ChatGPT<sup>3</sup> as the representative LLM due to its promising performance for code intelligence tasks (such as automated program repair [15, 16, 17], automatic code generation [18, 19]). In our empirical study, we first compare SCCLLM with three state-of-the-art baselines [9, 20, 21] in terms of automatic performance measures. For example, SCCLLM can average improve the performance by 7.70%, 8.14%, 2.49%, and 17.26% in terms of BLEU, ROUGE-1, ROUGE-2, and ROUGE-L respectively. Moreover, we show the effectiveness of our customized demonstration selection strategy through ablation studies. Our ablation studies show that our used strategy can help to select higher-quality demonstration examples when compared to a set of control strategies, which were designed to evaluate the rationality of the component settings in our customized strategy. Later, we also analyze the influence of the number of demonstrations on SCCLLM and find that the performance of SCCLLM is low when only a small number of demonstrations are provided. However, when more high-quality demonstrations are provided, the performance of SCCLLM can be substantially improved, which can eventually outperform state-of-the-art baselines. Since the automatic performance measures can only reflect the lexical similarity between the generated smart contract comment and the ground-truth smart contract comment, we finally conduct a human study to evaluate the quality of the generated comments. By following the human study methodology considered in the previous study for similar tasks [22, 23], we find SCCLLM can generate higher-quality comments than baselines in terms of similarity, naturalness, and informativeness perspectives.

Our automatic and human evaluation results show using LLMs and in-context learning is a promising direction to improve the smart contract comment quality. Therefore, we hope that more researchers can conduct follow-up research in this promising direction, and our proposed approach can be also customized for other software document generation tasks (such as commit message generation [24, 25], issue title generation [26, 27], and pull request title generation [28]).

The main contributions of our study can be summarized as follows:

- **Direction.** Recently, LLMs have shown high performance

---

<sup>1</sup>For large language models, zero-shot learning refers to the capability of the model to perform tasks without explicit examples.

<sup>2</sup><https://etherscan.io/>

<sup>3</sup><https://chat.openai.com/>

in different software engineering tasks (such as program repair, code generation, and test case generation) [29, 30]. However, to our best knowledge, the potential of LLMs to enhance the performance of smart contract comment generation has not been thoroughly investigated in previous studies. In light of the promising results shown by our research, we encourage more follow-up studies to the exploration of using LLMs in this specific task.

- **Approach.** We propose a novel approach SCCLLM based on the representative LLM (i.e., ChatGPT). Specifically, SCCLLM uses an effective customized retrieval strategy for selecting top- $k$  high-quality demonstrations and performs in-context learning by these retrieved demonstrations. After the in-context learning, SCCLLM can generate a corresponding comment for the target smart contract code snippet.
- **Study.** We conducted a comprehensive empirical study on the dataset with 29,720 (method, comment) pairs. Comparison results based on automatic performance measures and human studies show the effectiveness of SCCLLM. Ablation studies also show the rationality of our customized demonstration selection strategy.

To encourage the follow-up studies for applying LLMs to smart contract code comment generation, we share data, code, and detailed results at our project home:

<https://github.com/jun-jie-zhao/SCCLLM>.

The rest of this paper is organized as follows. Section 2 shows the framework and details of our proposed approach SCCLLM. Section 3 shows the empirical settings of our study, including research questions and design motivation, experimental subjects, performance measures, baselines, implementation details, and running platform. Section 4 presents our result analysis for research questions. Section 5 discusses the limitations of our study and threats to validity analysis. Section 6 summarizes related studies to our work and emphasizes the novelty of our study. Finally, Section 7 concludes our study and shows potential future directions.

## 2. Our Proposed Approach

We show the overall framework of our proposed approach SCCLLM in Figure 1. In this figure, we can find that SCCLLM mainly contains three phases. Specifically, in the **demonstration selection phase**, SCCLLM retrieves the top- $k$  smart contract code snippets from the historical corpus that are most similar to the target smart contract code snippet by considering syntax, semantics, and lexical information. In the **in-context learning phase**, SCCLLM utilizes the retrieved top- $k$  smart contract code snippets and their associated comments as demonstration examples, which can be used to mine potentially domain knowledge related to smart contract code comment generation from LLMs by in-context learning. In the **LLMs inference phase**, the input is the target smart contract code snippet, and the output is the corresponding comment generated by the

LLMs. In the rest of this section, we show the details of these three phases.

### 2.1. Demonstration Selection Phase

According to the recent survey for in-context learning [31], a customized demonstration example selection strategy can effectively utilize the domain knowledge hidden in the LLMs since high-quality demonstration examples that are highly related to the target can help LLMs better understand the investigated task. In previous study [13], demonstration selection strategies have typically been designed using token-based and sequence-based methods. However, these methods only consider one type of code information during demonstration retrieval. To address this issue, our study aims to employ a novel retrieval strategy that integrates multiple types of code information. The primary challenge faced in our design is how to effectively fuse various code information types (i.e., semantic information, lexical information, and syntactic similarity). Therefore, in this phase, we adopt our previously proposed information retrieval approach CCGIR [9] as our demonstration selection strategy. By using our customized demonstration selection strategy, we can select top- $k$  similar smart contract code snippets from the historical repository by considering semantic, syntactic, and lexical information when given the target smart contract code snippet. Our demonstration selection strategy can be divided into two parts: (1) the semantic-based retrieval part (i.e., the first part) and (2) the syntax and lexical-based retrieval part (i.e., the second part). Specifically, in the first part, we use CodeBERT [10] and BERT-whitening [32] to extract semantic information from smart contract code snippets. Then, we retrieve the top- $n$  smart code snippets from the historical corpus that are most similar to the target smart contract code snippet as candidates. However, directly using these top- $n$  candidate smart contract code snippets in terms of only semantic information may ignore their structural information and lexical information. Therefore, in the second part, we further consider the syntax and lexical information of these top- $n$  candidates. We calculate their lexical and syntactic similarities and then obtain the top- $k$  code snippets as the final demonstration examples based on the weighted sum of these two similarities. In the rest of this subsection, we show detailed information for these two parts.

#### 2.1.1. Semantic-based Retrieval Part

In this part, we first split the smart contract codes in the historical corpus (i.e., the training set) according to the CamelCase naming convention to obtain input sequences  $\{x_i\}_{i=1}^N$ , which  $N$  denotes the number of the smart contract code snippets in the historical corpus. Then, by following previous studies [33, 34, 35], we feed these sequences into CodeBERT [10] to obtain semantic vectors  $X_i \in \mathbb{R}^D$ , in which  $D$  represents the hidden dimension. Later, we further process the semantic vectors using BERT-whitening [32], which uses a simple linear transformation to enhance the isotropy of sentence representations, to reduce the dimensionality of the vector from  $D$  to  $d$ , and perform a linear transformation to obtain  $\{\tilde{X}_i\}_{i=1}^N$ . The purpose of using

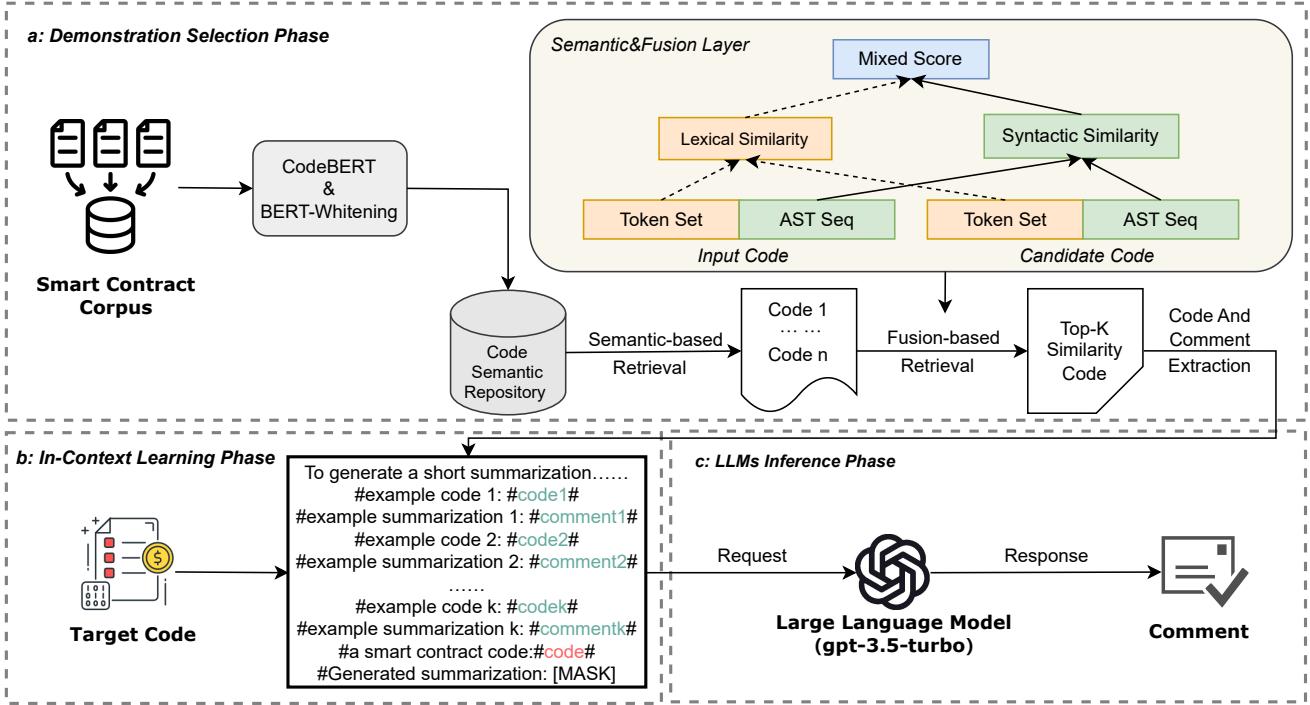


Figure 1: Framework of our proposed approach SCCLLM

BERT-whitening is to improve the quality and effectiveness of the embeddings generated by the BERT model. Previous studies show that this technique can help reduce the redundancy in the embeddings, result in faster training times, and better handle noisy data [36, 37]. Finally, we calculate the semantic similarity between the two smart contract code embeddings  $\tilde{X}_a$  and  $\tilde{X}_b$  by the L2 distance, which can be calculated as follows.

$$\text{semantic\_similarity}(\tilde{X}_a, \tilde{X}_b) = \sum_{i=1}^d (\tilde{X}_a[i] - \tilde{X}_b[i])^2 \quad (1)$$

#### 2.1.2. Syntax and Lexical-based Retrieval Part

Based on semantic similarity, we can retrieve the top- $n$  candidate smart contract code snippets from the corpus. However, only considering semantic information based on CodeBERT and BERT-whitening may ignore the structural and lexical information in the smart contract code. Therefore, in this part, we further incorporate syntactic and lexical similarity. Specifically, we employ AST (Abstract Syntax Tree) sequences and code tokens to compute a mixed score, which can help to identify more similar smart contract code snippets. The reason why we first consider semantic similarity in our two-stage demonstration selection strategy is that compared to lexical similarity or syntax similarity, the retrieval quality of semantic similarity is higher [9, 38, 39].

For two smart contract code snippets  $A$  and  $B$ , we use the method SimSBT [40] to generate two sequences  $\tilde{A}$  and  $\tilde{B}$ . SimSBT is used to generate the sequence for each AST, which can better represent the structure of the AST. We calculate the syntax similarity using the following formula:

$$\text{syntactic\_similarity}(A, B) = \frac{\sum(\text{len}(\tilde{A}), \text{len}(\tilde{B})) - \text{lev}}{\sum(\text{len}(\tilde{A}), \text{len}(\tilde{B}))} \quad (2)$$

Where lev is the Levenshtein distance [41] between sequences  $\tilde{A}$  and  $\tilde{B}$ .

Lexical information mainly considers tokens in two smart contract code snippets. Since code snippets often contain many repeated tokens, to address this issue, we treat the code as a sequential structure and represent the tokens of two smart contract code snippets through sets. Based on the code sequence, we remove duplicate tokens to obtain two token sets  $\text{set}_A$  and  $\text{set}_B$ . Then we calculate the lexical similarity by the Jaccard similarity.

$$\text{lexical\_similarity}(A, B) = \frac{|\text{set}_A \cap \text{set}_B|}{|\text{set}_A \cup \text{set}_B|} \quad (3)$$

Based on the top- $n$  similar smart contract code snippets retrieved in the first part, we further select the top- $k$  most similar smart contract code snippets by fusing the syntax similarity and the lexical similarity as follows:

$$\begin{aligned} \text{mixed\_score}(A, B) = & \lambda \times \text{lexical\_similarity}(A, B) \\ & + (1 - \lambda) \times \text{syntactic\_similarity}(A, B) \end{aligned} \quad (4)$$

where  $\lambda$  is a parameter that can adjust the weights between different similarities.

#### 2.2. In-Context Learning Phase

In-Context Learning [31] is a novel paradigm distinct from the fine-tuning paradigm. Fine-tuning is a resource-intensive

process, particularly as the current training parameters of large language models have significantly increased, leading to promising performance in many downstream tasks [42]. In contrast, in-context learning is a paradigm that utilizes a small number of demonstration examples to leverage the related domain knowledge in the LLMs for new tasks. It eliminates the need for gathering massive high-quality training data for new tasks, thereby avoiding the limitations of the fine-tuning paradigm.

However, the effectiveness of in-context learning is determined by the quality and quantity of the selected demonstration examples. In this phase, we utilize the top- $k$  smart contract code snippets retrieved in our customized demonstration selection phase as the demonstration examples. Based on these examples, we can construct the prompt for in-context learning. The prompt template used by SCCLLM is shown in Figure 2. Specifically, the constructed prompt consists of three parts: natural language prompt part, code demonstration part, and test query part. In particular, in the **natural language prompt part**, we first inform the LLMs to generate comments for the target smart contract code. According to the study of Sun et al. [43], a classical LLM ChatGPT, which is used in our study, often tends to generate overly lengthy comments, which may contain redundant information. In their empirical study, they find that using “short”, “in one sentence”, and “no more than xx words” in the prompt can effectively limit the length of generated comments. Based on their findings, we designed our introductory content for constructing the prompt as “To generate a short summarization in one sentence for smart contract code”. To help the LLMs capture demonstration examples in in-context learning, we added “To alleviate the difficulty of this task, we will give you top- $k$  examples. Please learn from them” in this part. In the **code demonstration part**, we use “#” to separate comments in the natural language and smart contract code, and add the top- $k$  demonstration examples in sequence to this prompt, which can help to utilize the related domain knowledge for smart contract comment generation in the LLMs. Finally, in the **test code part**, we input the target smart contract code and provide a prompt for generating a corresponding comment. Notice that to further prevent the LLMs from generating lengthy comments, we add the prompt “The length should not exceed <comment>” after generating the comment. The purpose of this setting is to further limit the length of the generated comment. Notice in our study, we fill the comment of the retrieved code with the highest similarity into “comment”.

### 2.3. LLMs Inference Phase

For previous deep learning-based methods [3], the performance depends on high-quality labeled data, which was time-consuming and laborious for the data labeling process. With the continuous development of ChatGPT, API invocation has also become the main means of using ChatGPT for various tasks. Our proposed approach SCCLLM does not require any model training and can directly generate code comments by calling the API interface. Specifically, we input the constructed prompt directly through the API gpt-3.5-turbo provided by OpenAI and then get the generated comment for the target smart contract code. The API gpt-3.5-turbo is the current mainstream version

```

To generate a short summarization in one sentence for smart contract code.
To alleviate the difficulty of this task, I will give you top-k examples.
Please learn from them.

Natural Language Prompt Part

#example code 1 :
function is running pre ico ( uint date )
public view returns ( bool ) {
    return start pre ico date <= date && date <= end pre ico date ;
}

#example summarization 1 :
return true if date in phase with give number .

.....
#example code k : <code k>
#example summarization k : <comment k>

Code Demonstration Part

#a smart contract code :
function is phase ( uint number , uint date )
view public returns ( bool ) {
    return start dates [ number ] <= date && date <= end dates [ number ] ;
}

#Generated summarization(The length should not exceed <comment>):
Test Code Part

```

Figure 2: The prompt template used by our proposed approach SCCLLM

of ChatGPT, which is trained on more training data and has lower usage costs.

## 3. Experimental Setup

In this section, we show the details of our experimental setup, including research questions and their design motivation, experimental subject, performance measures, state-of-the-art baselines, implementation details, and running platform.

### 3.1. Research Questions

To evaluate the effectiveness of our proposed approach SCCLLM and the rationality of the component setting in SCCLLM, we design the following four research questions (RQs).

#### RQ1: How effective is SCCLLM when compared with state-of-the-art baselines via automatic evaluation?

**Motivation.** In this RQ, we want to investigate whether SCCLLM can generate higher-quality smart contract code comments than state-of-the-art baselines. Therefore, we select CC-GIR [9], CodeT5 [20], and Rencos [21] as the state-of-the-art baselines. To evaluate the quality of smart contract code comments generated by different approaches automatically, we consider BLEU [44], ROUGE-1, ROUGE-2 and ROUGE-L [45] as our automatic performance measures.

#### RQ2: How effective is our proposed demonstration example selection strategy in SCCLLM?

**Motivation.** In previous studies [13, 46, 47], the researchers found that the quality of demonstrations can have a significant impact on the effectiveness of in-context learning. Therefore, in this RQ, we want to investigate whether our customized demonstration selection strategy can help to select high-quality

demonstrations, which can further improve the performance of SCCLLM. Specifically, we compare our demonstration selection strategy with three control approaches, which can investigate the influence of demonstration selection, BERT-whitening usage, and fusion layer usage by considering syntactic similarity and lexical similarity.

### RQ3: Whether the number of demonstration examples affect the effectiveness of SCCLLM?

**Motivation.** In RQ2, we mainly analyze the influence of different demonstration selection strategies. In this RQ, we want to further investigate the influence of the number of demonstration examples on the effectiveness of SCCLLM.

### RQ4: How effective is SCCLLM when compared with state-of-the-art baselines via human study?

**Motivation.** Performance measures (such as BLEU [44], ROUGE-1, ROUGE-2, and ROUGE-L [45]) can only evaluate the lexical similarity between the generated smart contract comments and the ground-truth comments. However, these performance measures are inadequate in reflecting the real semantic differences for comments [48]. Therefore, we want to conduct a human study for smart contract comment quality evaluation for different approaches in this RQ by considering similarity, naturalness, and informativeness perspectives.

### 3.2. Experimental Subject

The raw data of our experimental subject was originally shared by Zhuang et al. [49] for studying smart contract vulnerability detection, which was gathered from 40,932 smart contracts written in solidity on a popular and active smart contract community Etherscan.io<sup>4</sup>. Then this raw data was processed by Yang et al. [3] for studying the smart contract comment generation. For example, they only considered normal functional methods and modifiers. They removed smart contract codes, which contain less than four words. However, after the manual analysis in our study, we found there were still low-quality pairs in their processed dataset: (1) There are smart contract code snippets with different semantics but the duplicated comments, and (2) There are template comments, which may be automatically generated by smart contract development tools or paste copy behavior from developers. Figure 3 illustrates corresponding cases for these two problems. In the case of duplicated comments with different semantics, the two code snippets in this Figure have the same comments, but they have different semantics. In code 1, the triggering condition for the modifier is when the block number is NULL, whereas in code 2, the triggering condition is when the block number is not NULL. Although these two smart contract code snippets have different semantics, their corresponding comments are the same. In the case of template comments, these comments appear more frequently in the dataset than other comments and these comments cannot provide very clear semantics. However, the two smart contract code snippets shown in this Figure clearly have different semantics and need more clarified and concise comments. To improve the quality of our experimental subject, we removed

<sup>4</sup><https://etherscan.io/>

<pre><b>Duplicated Comments With Different Semantics</b>  <b>code 1</b> modifier <b>no feeding</b> in progress () {     require ( pending feedings [ msg . sender ] . block number == NUM_ , STR_ ); } <b>comment 1:</b> require that the user isn't feed a horsey already .  <b>code 2</b> modifier <b>feeding</b> in progress () {     require ( pending feedings [ msg . sender ] . block number != NUM_ , STR_ ); } <b>comment 2:</b> require that the user isn't feed a horsey already .</pre>
<pre><b>Template Comments</b>  <b>code 1</b> function <b>transfer</b> ( address _to , uint256 _value ) public returns ( bool success ) {     _transfer ( msg . sender , _to , _value );     <b>return</b> <b>BOOL_</b>; } <b>comment 1:</b> set the mine leader .  <b>code 2</b> function <b>set mint helper</b> ( address _mint helper ) external only owner {     <b>mint helper</b> = _mint helper ; } <b>comment 2:</b> set the mine helper .</pre>

Figure 3: Examples of two problems in the dataset shared by Yang et al. [3].

low-quality pairs with these problems as many as possible in a manual way. Finally, we obtain 29,720 ⟨method, comment⟩ pairs in our experimental subjects.

By following the previous experimental setting for smart contract code comment generation studies [49, 3], we split the dataset into the training set (80%), the validation set (10%), and the testing set (10%). Notice, different from baselines, our proposed approach SCCLLM does not need to use the validation set.

The statistical information of our experimental subject can be found in Table 1. After the dataset partitioning, the dataset is divided into 23,776 pairs for the training set, 2,972 pairs for the validation set, and 2,972 pairs for the testing set. Moreover, we also show the average (Avg.) tokens in the smart contract code snippets and comments for different sets.

Table 1: Statistical information of our experimental subject

Statistic	Train	Validation	Test
Number	23,776	2,972	2,972
Avg. tokens in codes	80.54	80.13	82.27
Avg. tokens in comments	12.05	11.97	12.1

### 3.3. Performance Measures

To evaluate the performance of SCCLLM and baselines, we consider automatic performance measures, such as BLEU, ROUGE-1, ROUGE-2, and ROUGE-L. These performance measures can effectively evaluate the lexical similarity between the generated smart contract code comments and the ground-truth comments and have been widely used in previous smart contract code comment generation studies [9, 3] and similar generation tasks for software engineering (such as source code sum-

marization [8, 40], Stack Overflow title generation [50, 51], issue title generation [26, 27], code generation [52, 53, 54]). We show the details of these performance measures as follows.

- **BLEU.** BLEU [44] (Bilingual Evaluation Understudy) is a machine translation metric that measures the text similarity between two texts by measuring the overlap of  $n$ -grams. In our study, we select the BLEU-4 variant ( $n$ -gram precision of 4) to measure the quality of the generated smart contract comments.
- **ROUGE-N.** ROUGE (Recall-Oriented Understudy for Gisting Evaluation)- $N$  [45] ( $N$  refers to  $n$ -gram, with values 1, 2, 3, 4) is an automatic evaluation measure based on  $n$ -grams. It assesses the quality of comments by counting the number of overlapping basic units between the generated and ground-truth comments. In our study, we only select ROUGE-1 and ROUGE-2, as they are well-suited for short comments and accurately capture text similarity.
- **ROUGE-L.** ROUGE (Recall-Oriented Understudy for Gisting Evaluation)-L [45] is an evaluation measure for measuring the similarity between the ground-truth comment and the generated comment by comparing their longest common subsequence.

For these performance measures, the value range is between 0 to 1, and the values are displayed as a percentage. Notice the higher the value of these performance measure values, the closer the generated comment is to the ground-truth comment (i.e., the better performance of the corresponding approach). To alleviate the internal threat due to implementation errors for these performance measures, we use nlg-eval library<sup>5</sup> to compute BLEU measure and Rouge library<sup>6</sup> to compute ROUGE measures.

#### 3.4. Baselines

To evaluate whether our proposed approach SCCLLM can achieve state-of-the-art performance, we consider the following three baselines related to our study.

- **CCGIR.** CCGIR [9] is an information retrieval technique for smart contract code comment generation that demonstrates superior performance among different information retrieval methods. It employs an information retrieval approach to retrieve the most similar smart contract code in the historical repository by considering semantic similarity, lexical similarity, and syntactic information. Finally, it reuses the comments associated with the retrieved most similar smart contract code.
- **CodeT5.** CodeT5 [20] is an encoder-decoder transformer model that is pre-trained based on T5 [55]. Compared to other deep learning models, it exhibits better comprehension of code information and possesses stronger

generation capabilities. This model employs a unified framework to seamlessly support code understanding and generation tasks, while also enabling multitask learning, thereby demonstrating excellent performance across various downstream tasks.

- **Rencos.** Rencos [21] is a hybrid approach for source code summarization that combines information retrieval and deep learning. Specifically, Rencos not only trains an attention-based encoder-decoder model using code snippets and comments from the training set but also incorporates two most similar code snippets retrieved based on semantic and syntactic similarities. During the encoding phase, the input code is combined with the two most similar code snippets. Finally, during the decoding phase, the comment is generated by incorporating the fused information.

Notice the first baseline CCGIR can be treated as the state-of-the-art baseline for smart contract code comment generation. In our previous study [9], we find CCGIR can significantly outperform the smart contract code comment generation approach MMTrans [3]. Therefore, we do not consider MMTrans as our baseline. To conduct a comprehensive evaluation, we also consider a representative deep learning-based baseline (i.e., CodeT5 [20]) and a representative hybrid baseline (i.e., Rencos [21]) for our investigated generation task.

To alleviate the internal threats, we utilize the scripts shared by these baselines [9, 21] and follow the hyperparameter settings suggested in their original studies. For the pre-trained model CodeT5, we implement it with Hugging Face<sup>7</sup>.

#### 3.5. Implementation Details

In our experiments, the detailed parameter settings in our demonstration selection phase can be found in Table 2. These values are configured based on the suggestions from previous studies [9, 32] and the optimization of our experimental results.

Table 2: The configuration of Hyper-parameters in our demonstration selection phase

Hyper-parameter	Value
Maximum input length of code snippet $x_a$	256
Dimension $D$ before BERT-whitening	768
Dimension $d$ after BERT-whitening	256
mixed_score coefficient $\lambda$	0.7
Number of top- $n$ candidates	10

In the LLMs inference phase, we only select five demonstration examples to construct the prompt for the in-context learning. The reason is the prompt size of the LLMs is limited, and the escalation in experimental expenses is associated

<sup>5</sup><https://github.com/Maluuba/nlg-eval>

<sup>6</sup><https://github.com/pltrdy/rouge>

<sup>7</sup><https://huggingface.co/Salesforce/codet5-base>

with an excess of demonstrations. For our experiments, we utilize the API interface version gpt-3.5-turbo, which has demonstrated good performance in various downstream tasks [56, 57]. To guarantee a fair comparison, we also optimize the parameters of the baseline methods to achieve optimal performance.

### 3.6. Running Platform

We run all the experiments on a computer (CPU 3.50GHz) with a GeForce RTX4090 GPU (24GB graphic memory). The running operating system is Windows 10.

## 4. Result Analysis

### 4.1. RQ1: Comparison with baselines via automatic evaluation

**Method.** To show the effectiveness of our proposed approach SCCLLM in smart contract comment generation, we select the information retrieval approach CCGIR [9], the deep learning approach CodeT5 [20], and the hybrid approach Rencos [21] as baselines. Notice in this RQ, for SCCLLM, we utilize the top-5 most similar code snippets as our demonstrations for in-context learning.

Table 3: Comparison results between SCCLLM and baselines in terms of four performance measures

Approach	BLEU	Rouge-1	Rouge-2	Rouge-L
CCGIR	31.21	32.94	17.33	24.05
CodeT5	30.93	32.63	16.89	23.69
Rencos	30.92	32.58	16.78	23.62
<b>SCCLLM</b>	<b>33.41</b>	<b>35.38</b>	<b>17.42</b>	<b>27.89</b>

**Result.** Table 3 presents the comparison results between SCCLLM and the baselines for smart contract comment generation. In this table, we emphasize the best performance for different performance measures in bold. Specifically, SCCLLM can achieve the performance of 33.41%, 35.38%, 17.42%, and 27.89% in terms of BLEU, ROUGE-1, ROUGE-2, and ROUGE-L performance measures. Compared to the baselines, the performance of SCCLLM can be improved on average by 7.70%, 8.14%, 2.49%, and 17.26% in terms of four performance measures. Then to check whether the performance difference between SCCLLM and baselines is significant, we conduct Wilcoxon signed-rank tests [58] at the confidence level of 95%. Our  $p$ -value is smaller than 0.05, which means the performance improvement of SCCLLM compared to baselines is significant.

When comparing two baselines CodeT5 and Rencos, we find they achieve similar results in our study. Although the experimental results of Rencos [21] indicate that the hybrid approach can outperform the deep learning approach. However, we consider CodeT5 as the deep learning-based baseline in our study, which is a state-of-the-art code pre-trained model, while the deep learning model part of Rencos only considers the attentional encoder-decoder model, which is trained from scratch. It is worth noting that the information retrieval approach CCGIR still outperforms the deep learning approach

CodeT5 and the hybrid approach Rencos in terms of all the performance measures in our study. This finding is consistent with our previous research findings [9] due to the extensive code reuse (i.e., code clone) during smart contract development. However, our proposed approach SCCLLM can outperform CCGIR, which shows applying LLMs to smart contract generation is a valuable direction and worth paying attention to. Here, we find all the approaches achieve low performance for the ROUGE-2 measure. The possible reason is that ROUGE-2 is mainly focused on 2-gram overlap when compared to ROUGE-1 and most generation-based approaches tend to generate lengthy code comments.

Finally, we use two cases in Figure 4 to show the effectiveness of our proposed approach. In the first case, CodeT5 and Rencos can learn the keyword “balance” in the target smart contract code. However, these two baselines fail to understand the specific meaning of the target code. CCGIR cannot retrieve sufficiently similar code, so its reused comment is independent of the semantics of the target code. While the comment generated by SCCLLM conveys an identical semantic when compared with the ground-truth comment. In the second case, the comments generated by SCCLLM as well as the ground-truth comment both contain the key phrases “modifier” and “not locked”. This indicates that SCCLLM can effectively understand the target smart contract code and generate the corresponding comment by LLMs and in-context learning. However, the comments generated by three baselines can only convey the basic notion of “not lock”, resulting in low-quality comments. Based on these two cases, we find that baselines struggle to learn the knowledge of the target smart contract code from the limited corpus, but SCCLLM can leverage the related knowledge of LLMs to generate better comments for smart contract code snippets.

### Answer to RQ1

By combining LLMs with in-context learning, SCCLLM outperforms all baselines in terms of four automatic performance measures. For example, SCCLLM can outperform the baselines by at least 7.70% in terms of BLEU.

### 4.2. RQ2: Ablation study on Demonstration Selection Strategy

**Method.** In this RQ, we want to show the effectiveness of our customized demonstration selection strategy in selecting high-quality demonstrations for in-context learning. As introduced in Section 2.1, our customized demonstration selection strategy includes two major components. The first component utilizes CodeBERT and BERT-Whitening for semantic vector extraction, capturing code semantic information. The second component integrates AST sequences with code tokens in the fusion layer, combining syntax and lexical information to identify more similar smart contract code snippets. Based on the component settings of our customized demonstration selection strategy, we design the following three control strategies.

- **The first control strategy.** To show the significance of our customized demonstration selection strategy for SCCLLM, we randomly select  $k$  smart contract code snip-

<p><b>Smart Contract Code</b></p> <pre>function get balance () {     public view returns ( uint256 balance ) {         return this . balance ;     }</pre> <hr/> <p><b>Ground Truth:</b> return current contract balance .</p> <p><b>SCCLLM:</b> return the balance of the contract .</p> <p><b>CCGIR:</b> return bonuspool .</p> <p><b>CodeT5:</b> get balance .</p> <p><b>Rencos:</b> check the balance .</p>
<p><b>Smart Contract Code</b></p> <pre>modifier not locked () {     require ( ! locked );     _;</pre> <hr/> <p><b>Ground Truth:</b> modifier to make a function callable only when the contract be not lock .</p> <p><b>SCCLLM:</b> this modifier ensures that the contract is not locked .</p> <p><b>CCGIR:</b> agreement not lock .</p> <p><b>CodeT5:</b> check if the generation period be not lock .</p> <p><b>Rencos:</b> agreement not lock .</p>

Figure 4: Comments generated by SCCLLM and baselines for two cases

pets from the historical repository and use “w/o DSS” to denote this control strategy.

- **The second control strategy.** In this control strategy, we aim to investigate whether using the BERT-whitening [32] can help to select demonstrations with higher quality and use “ w/o BERT-whitening” to denote this control strategy. Specifically, we remove the BERT-whitening operation in our demonstration selection strategy and directly utilize CodeBERT to generate code embeddings without any optimization or dimensionality reduction.
- **The third control strategy,** In this control strategy, we aim to investigate whether further considering the syntactic similarity and lexical similarity can help to select demonstrations with higher quality and use “ w/o (Lexical & Syntactic Similarity)” to denote this control strategy. Specifically, we remove the fusion layer based on lexical and syntactic similarity from our demonstration selection strategy, which can retrieve the top  $k$  most similar examples by only considering code semantic information.

Due to the high economic cost of calling ChatGPT to perform ablation experiments when considering all the smart contract code snippets in the test set, we employ a sampling method [59] that randomly selects samples from the test set. The formula for the sampled number can be computed as follows:

$$MIN = \frac{n_0}{1 + \frac{n_0 - 1}{size}} \quad (5)$$

Where  $n_0$  represents the confidence level, and  $size$  is the size of the test set. There is an error range  $(\frac{Z^2 \times 0.25}{e^2})$  for  $n_0$ , where  $e$  is the hyper-parameter and  $Z$  is the confidence level score. In RQ2, we chose the smallest sample based on a confidence level of 95% and  $e=0.05$ . In the end, we needed to randomly select 340 samples from the test set according to this formula.

**Result.** We show our ablation study results in Table 4. From this table, we observe that when randomly providing demonstrations without using the customized demonstration selection strategy, the quality of comments directly generated by LLMs is low, with a performance drop of 39.99%, 45.17%, 80.45%, and 49.65%, respectively in terms of four performance measures. This highlights using high-quality demonstrations in in-context learning can help to fully utilize the related domain knowledge in LLMs.

For the second control strategy, we find the performance of SCCLLM decreases when the code embeddings are not optimized by BERT-whitening, with a performance drop of 10.05%, 10.52%, 17.57%, and 10.97%, respectively. This demonstrates the effectiveness of using BERT-whitening in improving the retrieval ability of our demonstration selection strategy in selecting higher-quality demonstration examples.

For the third control strategy, we find the performance of SCCLLM decreases when removing the syntax and lexical-based retrieval part in our demonstration selection strategy, with a performance drop of 11.79%, 13.36%, 21.98%, and 13.29%, respectively. Therefore, further considering the lexical information and the syntax information can help to select higher-quality demonstration examples, which eventually improves the performance of SCCLLM.

#### Answer to RQ2

By using our customized demonstration selection strategy, SCCLLM can help to select higher-quality demonstrations for in-context learning, which can finally improve the performance of SCCLLM.

#### 4.3. RQ3: Performance influence on the number of demonstrations

**Method.** In this RQ, we want to examine the performance influence of the number of demonstrations for SCCLLM. Specifically, we conduct experiments with different settings (i.e., zero-shot learning, one-shot learning, and few-shot learning), and compare the performance of different settings with three baselines (i.e., CCGIR [9], CodeT5 [20] and Rencos [21]). We use “zero-shot learning” to refer to the setting where no demonstration is provided for SCCLLM. “one-shot learning” refers to the setting where only a single demonstration is provided for SCCLLM, while “few-shot learning” refers to settings where a few demonstrations are provided for SCCLLM.

Due to the maximum prompt size limitation for the API gpt-3.5-turbo, we use at most five demonstrations to construct the prompt in our study. In this experiment, we investigate the performance influence of different demonstration numbers by using all the smart contract code snippets in the test set. To guarantee a fair comparison, we use the same prompt template

Table 4: Ablation results of SCCLLM with different demonstration selection strategies

Demonstration Selection Strategy	BLEU	GOUGE-1	GOUGE-2	GOUGE-L
w/o DSS	19.04	17.15	3.06	13.49
w/o BERT-whitening	28.54	27.99	12.90	23.85
w/o (Lexical & Syntactic Similarity)	27.99	27.10	12.21	23.23
Our Customized Strategy	<b>31.73</b>	<b>31.28</b>	<b>15.65</b>	<b>26.79</b>

shown in Figure 2, but the difference is that the number of demonstrations differs.

**Result.** We show the performance of SCCLLM with the different number of demonstrations in Table 5. From this table, we find that when using zero-shot learning and one-shot learning settings, the performance of SCCLLM is very low in terms of four performance measures. For example, if using the zero-shot learning setting, the performance of SCCLLM decreases 27.48%, 49.71%, 83.13%, and 46.64%, respectively, when compared to the fine-tuning approach CodeT5. Our finding indicates that SCCLLM cannot effectively utilize the related domain knowledge for smart contract comment generation task in the LLMs if we at most give one demonstration for in-context learning. In this table, we find that using the zero-shot learning setting can outperform using the one-shot learning setting. The potential reason is that for zero-shot learning, we do not provide any demonstration for the prompt and simply set the length of the generated comment should not exceed 15 words. Therefore, shorter comments may result in a higher performance of SCCLLM when using zero-shot learning.

When using few-shot learning, we observe a significant performance improvement for SCCLLM. Specifically, when providing three demonstration examples for in-context learning, the performance of SCCLLM can achieve similar performance with CodeT5. When providing five demonstration examples for in-context learning, SCCLLM can eventually outperform all the baselines.

Table 5: The performance influence of different demonstration numbers for SCCLLM

Approach	BLEU	Rouge-1	Rouge-2	Rouge-L
CCGIR	31.21	32.94	17.33	24.05
CodeT5	30.93	32.63	16.89	23.69
Rencos	30.92	32.58	16.78	23.62
with zero-shot	22.43	16.41	2.85	12.64
with one-shot	18.42	16.09	3.10	12.47
with 3-shot	29.05	27.21	11.60	23.19
with 5-shot	<b>33.41</b>	<b>35.38</b>	<b>17.42</b>	<b>27.89</b>

### Answer to RQ3

When providing five high-quality demonstration examples for in-context learning, SCCLLM can more effectively utilize the related knowledge for smart contract comment generation and achieve better performance than baselines.

### 4.4. RQ4: Comparison with baselines via human study

**Method.** Based on the findings of the recent study by Sun et al. [43], the current automatic performance evaluation measures cannot be used to effectively assess the quality of comments generated by LLMs. To alleviate this construct threat, we perform a human study to further assess the effectiveness of our proposed approach SCCLLM. In our human study, we mainly follow the methodology used by previous studies for source code summarization [22, 23]. Specifically, we recruit five participants, who have at least three years of experience in smart contract development and maintenance. These participants are senior researchers or developers, who are not co-authors of our study. We employed the same sampling method used in RQ2 and finally randomly selected 340 smart contract code snippets in the test set. For each smart contract code, we show the participants the ground-truth comments and the comments generated by four different approaches. To guarantee a fair comparison, the participants do not know which approach generates the comment. Later, we allowed the participants to use the Internet to facilitate understanding the target smart contract codes if there exist concepts or codes that they are unfamiliar with. Before they participated in our human study, we ensured that all involved participants received a comprehensive tutorial. This process was designed to familiarize each participant with both the task expectations and the evaluation measures at hand. This training aimed to mitigate the risk of inherent biases and inconsistency in their assessments. Finally, we require each participant to evaluate only 20 smart contract codes in a half-day to avoid biases caused by fatigue. Each participant is asked to rate the generated four comments from three perspectives.

- **Similarity.** This perspective reflects the similarity between the generated smart code comments and the ground-truth comments.
- **Naturalness.** This perspective reflects the fluency of generated smart contract comments from grammar.
- **Informativeness.** This perspective reflects the information richness of the generated smart contract comments.

We use a five-point system for scoring (i.e., 1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent). We show a questionnaire used in our human study in Figure 5.

**Result.** We show our human study results in Figure 6. Specifically, SCCLLM can achieve 3.31, 3.80, and 3.92 in terms of Similarity, Naturalness, and Informativeness. In terms of

<b>Promble Description:</b> In the following questionnaire, for each given smart contract code snippet, there will be corresponding ground-truth comment and comments generated by three different approaches. You are required to rate these four comments. We adopt a five-point system for scoring (i.e., 1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent).			
<b>Smart Contract Code</b> <pre>function getBalance() {     public view returns (uint256 balance) {         return this.balance;     } }</pre>			
<b>Ground Truth:</b> return current contract balance .			
Comment 1: return the balance of the contract .	Comment 2: get balance .	Comment 3: return bonuspool .	Comment 4: check the balance .
Question 1: Please evaluate the similarity between these comments and the Ground Truth. Question 2: Please evaluate the fluency of these comments and whether their semantics are coherent. Question 3: Please assess whether these comments are information-rich from a content perspective.			

Figure 5: A questionnaire used in our human study

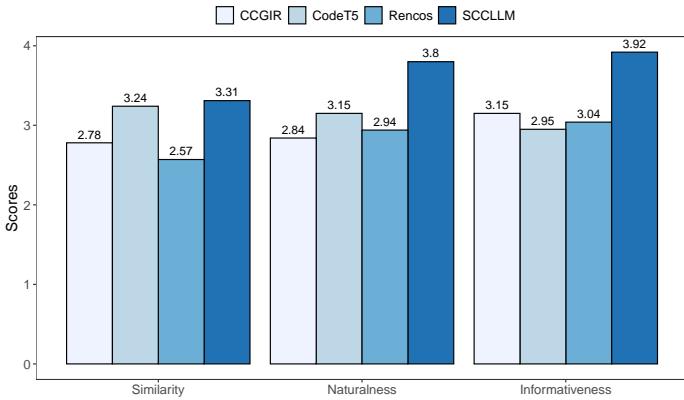


Figure 6: Our human study results between SCCLLM and baselines

similarity, due to the richer semantics in the smart contract comments generated by SCCLLM, they exhibit a higher semantic similarity with the ground-truth comments. The result of 3.31 implies that the smart contract comments generated by SCCLLM have higher quality than baselines. In terms of naturalness and informativeness, SCCLLM still performs better than all baselines, demonstrating that most of the smart contract comments generated by SCCLLM are easy to understand and read, and excel in semantic comprehension. These more readable comments will help smart contract developers understand the code better and increase development efficiency.

#### Answer to RQ4

Our human study results show that SCCLLM can outperform baselines when evaluating the quality of the smart contract comments from similarity, naturalness, and informativeness perspectives.

## 5. Discussions

### 5.1. Limitations of SCCLLM

Though SCCLLM shows competitive performance when compared to baselines for automatic evaluation and human evaluation, we also find that SCCLLM may generate low-quality comments. In this subsection, we randomly select 50 cases of this type and analyze these comments manually. Finally, we identify the three challenging types of smart contract comment generation for SCCLLM.

```

01. // Target Smart Contract Code
02. function add ( uint256 a , uint256 b ) {
03.   internal pure returns ( uint256 c ) {
04.     c = a + b ;
05.     assert ( c >= a ) ;
06.     return c ;
07.   }
08.
09. // Ground Truth: safemath add function .
10. // SCCLLM: adds two numbers, ensuring no overflow occurs .

```

Figure 7: Case with semantic similarity but low lexical similarity.

**The first challenge type** is generating comments with semantic similarity but low lexical similarity. In Figure 7, we present a case to show this challenge type. In this case, the ground-truth comment indicates that the smart contract code snippet employs the “add” function from the “safemath” library, which in the Solidity language is used for secure algorithms to prevent data overflow. While the comment generated by SCCLLM directly explains this code, indicating its purpose of preventing overflow. We observe that both the ground-truth comment and the comment generated by SCCLLM can accurately convey the functionality of this code, but differ in the level of expertise: the ground-truth comment being more professional and the comment generated by SCCLLM being straightforward and somewhat redundant. In this case, the low lexical similarity can result in a low score in automatic evaluation. Therefore, designing new performance measures based on comment semantic similarity can alleviate this challenge type.

**The second challenge type** is failing to fully comprehend the target smart contract code and directly reusing the comments from similar demonstrations. In our approach SCCLLM, we provide high-quality demonstrations for in-context learning. However, there are cases in the provided demonstrations where the code is the same but the comments differ, leading SCCLLM to directly reuse the comments. In Figure 8, we present a case to show this challenge type. We find that the target code and retrieved code are almost same, but correspond to different correct comments. SCCLLM reuses the demonstration comments directly after learning the demonstration. Although the reused comments are semantically correct, they will result in a lower score in terms of automatic evaluation measures and will be treated as low-quality comments.

**The third challenge type** is difficulties in effectively understanding code purposes/functionality for some smart contract code snippets. In Figure 9, we present a case to show this challenge type. The ground-truth comment shows the purpose of this smart contract code (i.e., “check for the possibility of buy tokens”). However, SCCLLM can only capture the words (such as “cap”, “period”, and “non-zero purchase”) in the target smart contract code, which can only provide shallow information for this code.

### 5.2. Threats to Validity

In this subsection, we mainly discuss the potential threats to our empirical findings.

**Internal threats.** The first internal threat is the potential implementation faults in SCCLLM. To alleviate this threat, we

```

 81. # Target Smart Contract Code
 82. function transfer for multi addresses ( address [ ] _addresses , uint256 [ ] _amounts )
 83. can transfer public returns ( bool ) {
 84.     for ( uint256 i = NUM_ ; i < _addresses . length ; i ++ ) {
 85.         require ( _addresses [ i ] != address ( NUM_ ) );
 86.         require ( _amounts [ i ] <= balances [ msg . sender ] );
 87.         require ( _amounts [ i ] > NUM_ );
 88.         balances [ msg . sender ] = balances [ msg . sender ] . sub ( _amounts [ i ] );
 89.         balances [ _addresses [ i ] ] = balances [ _addresses [ i ] ] . add ( _amounts [ i ] );
 90.         transfer ( msg . sender , _addresses [ i ] , _amounts [ i ] );
 91.     }
 92.     return BOOL_ ;
 93. }
 94.
 95. # Retrieved Code
 96. function transfer for multi addresses ( address [ ] _addresses , uint256 [ ] _amounts )
 97. can transfer public returns ( bool ) {
 98.     for ( uint256 i = NUM_ ; i < _addresses . length ; i ++ ) {
 99.         require ( _addresses [ i ] != address ( NUM_ ) );
100.         require ( _amounts [ i ] <= balances [ msg . sender ] );
101.         require ( _amounts [ i ] > NUM_ );
102.         balances [ msg . sender ] = balances [ msg . sender ] . sub ( _amounts [ i ] );
103.         balances [ _addresses [ i ] ] = balances [ _addresses [ i ] ] . add ( _amounts [ i ] );
104.         transfer ( msg . sender , _addresses [ i ] , _amounts [ i ] );
105.     }
106.     return BOOL_ ;
107. }
108. # Retrieved Code Comment: transfer tokens to multiple address .
109. # Ground Truth: same functionality a transfer .
110. # SCCLLM: transfer tokens to multiple address .

```

Figure 8: Case where the target smart contract code is not fully understood, and the comments from the provided demonstrations are directly reused.

```

 81. //Target Smart Contract Code
 82. function valid purchase ( )
 83. internal constant returns ( bool ) {
 84.     bool withinCap = wei raised . add ( msg . value ) <= hard cap ;
 85.     bool within period = now >= start time && now <= end time ;
 86.     bool non zero purchase = msg . value != NUM_ ;
 87.     return ( within period && non zero purchase )
 88.         && withinCap && is within sale time limit ( );
 89. }
 90.
 91. //Ground Truth: check for the possibility of buy tokens .
 92. //SCCLLM: check if the purchase is valid based on the cap, period, and non-zero purchase

```

Figure 9: Case where the purposes/functionality of the smart contract code is not effectively comprehended.

perform code inspection in our implemented code, especially the demonstration selection strategy part. The second internal threat is the number of demonstrations used in our study. To alleviate this threat, we analyze the performance influence of demonstration number for SCCLLM in Section 4.3 and find that only using five demonstrations can achieve promising performance and outperform baselines. However, improving the number of demonstrations may further improve the performance of SCCLLM but at the cost of a higher escalation in experimental expenses.

**External threats.** The first external threat is the quality of the experimental subject. To alleviate this threat, we use the gathered  $\langle$ method, comment $\rangle$  smart contract pairs shared by Yang [3]. In their study, they performed a set of processing to filter the low-quality pairs from the raw data provided by Zhuang et al. [49]. After the manual analysis in our study, we also found there still exists some low-quality pairs, which have duplicated comments but with different semantics or have template comments. We also identify and remove these kinds of pairs, which can further improve the experimental subject quality. The second external threat is the customized demonstration selection strategy used in SCCLLM. In our study, we designed a set of experiments to verify the rationality of the component setting in this strategy. Moreover, comparison results with baselines also show the effectiveness of this strategy for retrieving high-quality demonstrations for in-context learning.

**Conclusion threats.** The conclusion threat is related to evaluation bias in our human study. To alleviate this threat, we first invite participants who are familiar with smart contract development. Second, we provided a tutorial before our human

study, which ensured that all of the participants could understand our protocol. Finally, we follow the methodology used by previous studies for a similar task (i.e., source code summarization) [22, 23] to guarantee the quality of our human study.

**Construct threats.** The construct threat is related to the performance measures. To alleviate this threat, we consider four performance measures, which have been widely used in previous similar tasks, such as source code summarization [60, 61, 23]. Since the automatic measures can only evaluate the lexical similarity between the generated smart contract comments and the ground-truth comments, we further conducted a human study to evaluate the quality of the generated smart contract comments by considering similarity, naturalness, and informativeness.

## 6. Related Work

In this section, we present the relevant research on smart contract comment generation and recent advances in applying LLMs to Software Engineering tasks.

### 6.1. Smart Contract Code Comment Generation

To the best of our knowledge, Yang et al. [3] were the first to study the automatic smart contract comment generation problem. Their proposed approach MMTrans learns the smart contract code representation from two heterogeneous modalities (i.e., SBT sequences and graphs based on abstract syntax trees). Their experimental results show that MMTrans can outperform some state-of-the-art baselines for source code summarization (such as Hybrid-DeepCom [5], code+ggn+GRU [62], and Vanilla-Transformer [63]). Since code reuse is common in smart contract development, we [9] further propose a simple but effective information retrieval-based approach CCGIR. For the target smart contract code, this approach can effectively retrieve the most similar code from the historical repository and directly reuse the corresponding comment. In our empirical study, we find CCGIR can outperform different information retrieval approaches (such as NNGen [24]) and MMTrans [3].

However, the performance of the previous studies [3, 9] is still limited due to the amount of available training data. This issue is more obvious for the information retrieval-based approach CCGIR [9]. To alleviate this problem, we aim to generate smart contract comments by using LLMs. To achieve this goal, we use the customized demonstration selection strategy to select high-quality demonstrations and use in-context learning to fully utilize the related knowledge in the LLMs for the target smart contract code. To the best of our knowledge, we are the first to apply LLMs to smart contract comment generation. Our experimental results confirm that this direction is practical and feasible.

### 6.2. Applying LLMs to Software Engineering Tasks

Large language models (LLMs) refer to a class of artificial intelligence models that use an enormous amount of parameters and are designed to process and generate human-like text based on large-scale language datasets [64]. As a result,

LLMs have been used for many mainstream software engineering tasks. For the task of automated program repair, Xia et al. [15] utilized LLMs to directly generate correct code given the prefix and suffix context. Then they [16] conducted extensive empirical evaluations by considering nine different LLMs on five popular program repair datasets. Recently, they [17] further leveraged test failure information and earlier patch attempts in a conversational manner, which can prompt LLMs to generate more correct patches. For automated code generation tasks, Dong et al. [18] proposed a self-collaboration approach for code generation by ChatGPT. Liu et al. [19] guided ChatGPT to generate better code with prompt engineering for two code generation tasks (i.e., text-to-code generation and code-to-code generation). Liu et al. [65] proposed a code generation benchmarking framework, which can rigorously evaluate the functional correctness of the codes generated by ChatGPT. For the task of source code summarization, Zhu et al. [66] performed empirical studies between deep learning methods (including LLMs) and information retrieval methods. Sun et al. [43] performed source code summarization via ChatGPT and discussed the advantages and disadvantages of ChatGPT in this task. In our study, we aim to apply LLMs to a new software engineering task and propose a novel approach SCCLLM, which can effectively utilize the related domain knowledge in LLMs for smart contract comment generation via in-context learning.

## 7. Conclusion

In this study, we are the first to automatically generate smart contract comments by LLMs and in-context learning. In our proposed approach, we utilize the customized demonstration selection strategy to select high-quality demonstrations, which can effectively utilize the related knowledge in LLMs via in-context learning for smart contract comment generation. Our experimental results show SCCLLM can significantly outperform baselines in automatic evaluation and human evaluation. Our ablation studies also provided guidelines for effectively using SCCLLM.

In the future, we first aim to further improve the performance of SCCLLM by designing more effective demonstration selection strategies. We second want to guide ChatGPT to generate higher-quality comments by prompt engineering. Finally, we want to design more practical performance measures, which can effectively measure the semantic similarity between the ground-truth smart contract comments and the comments generated by SCCLLM.

## Acknowledgement

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. Junjie Zhao and Xiang Chen have contributed equally to this work and they are co-first authors. This work is supported in part by the National Natural Science Foundation of China (Grant no 62202419).

## Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## CRediT Authorship Contribution Statement

**Junjie Zhao:** Data curation, Software, Validation, Conceptualization, Methodology, Writing -review & editing. **Xiang Chen:** Conceptualization, Methodology, Writing -review & editing, Supervision. **Guang Yang:** Conceptualization, Data curation, Software. **Yiheng Shen:** Conceptualization, Validation.

## References

- [1] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: Challenges and opportunities, *IEEE Transactions on Software Engineering* 47 (10) (2019) 2084–2106.
- [2] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, M. Imran, An overview on smart contracts: Challenges, advances and platforms, *Future Generation Computer Systems* 105 (2020) 475–491.
- [3] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, M. Zhang, A multi-modal transformer-based code summarization approach for smart contracts, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, 2021, pp. 1–12.
- [4] N. He, L. Wu, H. Wang, Y. Guo, X. Jiang, Characterizing code clones in the ethereum smart contract ecosystem, in: Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24, Springer, 2020, pp. 654–675.
- [5] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, *Empirical Software Engineering* 25 (2020) 2179–2217.
- [6] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, D. Paul, Setrtransformer: A transformer-based code semantic parser for code comment generation, *IEEE Transactions on Reliability* 72 (1) (2022) 258–273.
- [7] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, D. Yu, Seccnn: A semantic cnn parser for code comment generation, *Journal of Systems and Software* 181 (2021) 111036.
- [8] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: Proceedings of the 26th conference on program comprehension, 2018, pp. 200–210.
- [9] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, H. Lin, Ccgir: Information retrieval-based code comment generation method for smart contracts, *Knowledge-Based Systems* 237 (2022) 107858.
- [10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codetbert: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.
- [11] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, T. Tuytelaars, A continual learning survey: Defying forgetting in classification tasks, *IEEE transactions on pattern analysis and machine intelligence* 44 (7) (2021) 3366–3385.
- [12] N. Nashid, M. Sintaha, A. Mesbah, Retrieval-based prompt selection for code-related few-shot learning, in: Proceedings of the 45th International Conference on Software Engineering (ICSE’23), 2023.
- [13] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning (2024).
- [14] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, *ACM Computing Surveys* 55 (9) (2023) 1–35.
- [15] C. S. Xia, L. Zhang, Less training, more repairing please: revisiting automated program repair via zero-shot learning, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 959–971.

- [16] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery, 2023.
- [17] C. S. Xia, L. Zhang, Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt, arXiv preprint arXiv:2304.00385 (2023).
- [18] Y. Dong, X. Jiang, Z. Jin, G. Li, Self-collaboration code generation via chatgpt, arXiv preprint arXiv:2304.07590 (2023).
- [19] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, M. Yan, Improving chatgpt prompt for code generation, arXiv e-prints (2023) arXiv–2305.
- [20] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.
- [21] J. Zhang, X. Wang, H. Zhang, H. Sun, X. Liu, Retrieval-based neural source code summarization, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1385–1397.
- [22] F. Mu, X. Chen, L. Shi, S. Wang, Q. Wang, Automatic comment generation via multi-pass deliberation, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–12.
- [23] D. Roy, S. Fakhoury, V. Arnaoudova, Reassessing automatic evaluation metrics for code summarization tasks, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1105–1116.
- [24] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, X. Wang, Neural-machine-translation-based commit message generation: how far are we?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 373–384.
- [25] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, W. Zhang, A large-scale empirical study of commit message generation: models, datasets and evaluation, Empirical Software Engineering 27 (7) (2022) 198.
- [26] S. Chen, X. Xie, B. Yin, Y. Ji, L. Chen, B. Xu, Stay professional and efficient: automatically generate titles for your bug reports, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 385–397.
- [27] H. Lin, X. Chen, X. Chen, Z. Cui, Y. Miao, S. Zhou, J. Wang, Z. Su, Gen-fl: Quality prediction-based filter for automated issue title generation, Journal of Systems and Software 195 (2023) 111513.
- [28] T. Zhang, I. C. Irsan, F. Thung, D. Han, D. Lo, L. Jiang, Automatic pull request title generation, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022, pp. 71–81.
- [29] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, Q. Wang, Software testing with large language model: Survey, landscape, and vision, arXiv preprint arXiv:2307.07221 (2023).
- [30] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, arXiv preprint arXiv:2308.10620 (2023).
- [31] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, Z. Sui, A survey for in-context learning, arXiv preprint arXiv:2301.00234 (2022).
- [32] J. Su, J. Cao, W. Liu, Y. Ou, Whitening sentence representations for better semantics and faster retrieval, arXiv e-prints (2021) arXiv–2103.
- [33] K. Liu, G. Yang, X. Chen, Y. Zhou, El-codebert: Better exploiting codebert to support source code-related classification tasks, in: Proceedings of the 13th Asia-Pacific Symposium on Internetworkware, 2022, pp. 147–155.
- [34] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, T. Chen, Exploitgen: Template-augmented exploit code generation based on codebert, Journal of Systems and Software 197 (2023) 111577.
- [35] K. Liu, X. Chen, C. Chen, X. Xie, Z. Cui, Automated question title reformulation by mining modification logs from stack overflow, IEEE Transactions on Software Engineering 49 (9) (2023) 4390–4410.
- [36] W. Yin, L. Shang, Efficient nearest neighbor emotion classification with bert-whitening, in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 4738–4745.
- [37] W. Zhuo, Y. Sun, X. Wang, L. Zhu, Y. Yang, Whitenedcse: Whitening-based contrastive learning of sentence embeddings, in: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2023, pp. 12135–12148.
- [38] J. K. Siow, S. Liu, X. Xie, G. Meng, Y. Liu, Learning program semantics with code representations: An empirical study, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 554–565.
- [39] C. Yu, G. Yang, X. Chen, K. Liu, Y. Zhou, Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022, pp. 82–93.
- [40] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, K. Liu, Comformer: Code comment generation via transformer and fusion method-based hybrid code representation, in: 2021 8th International Conference on Dependable Systems and Their Applications (DSA), IEEE, 2021, pp. 30–41.
- [41] L. Yujian, L. Bo, A normalized levenshtein distance metric, IEEE transactions on pattern analysis and machine intelligence 29 (6) (2007) 1091–1095.
- [42] X. Liu, D. McDuff, G. Kovacs, I. Galatzer-Levy, J. Sunshine, J. Zhan, M.-Z. Poh, S. Liao, P. Di Achille, S. Patel, Large language models are few-shot health learners, arXiv preprint arXiv:2305.15525 (2023).
- [43] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, et al., Automatic code summarization via chatgpt: How far are we?, arXiv preprint arXiv:2305.12865 (2023).
- [44] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
- [45] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text summarization branches out, 2004, pp. 74–81.
- [46] Z. Zhao, E. Wallace, S. Feng, D. Klein, S. Singh, Calibrate before use: Improving few-shot performance of language models, in: International Conference on Machine Learning, PMLR, 2021, pp. 12697–12706.
- [47] T. Gao, A. Fisch, D. Chen, Making pre-trained language models better few-shot learners, in: Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL-IJCNLP 2021, Association for Computational Linguistics (ACL), 2021, pp. 3816–3830.
- [48] S. Haque, Z. Eberhart, A. Bansal, C. McMillan, Semantic similarity metrics for evaluating source code summarization, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 36–47.
- [49] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, Q. He, Smart contract vulnerability detection using graph neural networks, in: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, 2021, pp. 3283–3290.
- [50] Z. Gao, X. Xia, J. Grundy, D. Lo, Y.-F. Li, Generating question titles for stack overflow from mined code snippets, ACM Transactions on Software Engineering and Methodology (TOSEM) 29 (4) (2020) 1–37.
- [51] K. Liu, G. Yang, X. Chen, C. Yu, Sotile: A transformer-based post title generation approach for stack overflow, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 577–588.
- [52] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, L. Zhang, Treegen: A tree-based transformer architecture for code generation, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, 2020, pp. 8984–8991.
- [53] G. Yang, Y. Zhou, X. Chen, C. Yu, Fine-grained pseudo-code generation method via code feature extraction and transformer, in: 2021 28th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2021, pp. 213–222.
- [54] G. Yang, X. Chen, Y. Zhou, C. Yu, Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 361–372.
- [55] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, The Journal of Machine Learning Research 21 (1) (2020) 5485–5551.
- [56] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, S. Garg, Verigen: A large language model for verilog code generation, arXiv preprint arXiv:2308.00708 (2023).
- [57] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, T. Bryksin,

- From commit message generation to history-aware commit message completion, arXiv preprint arXiv:2308.07655 (2023).
- [58] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in Statistics: Methodology and Distribution, Springer, 1992, pp. 196–202.
  - [59] R. Singh, N. S. Mangat, Elements of survey sampling, Vol. 15, Springer Science & Business Media, 2013.
  - [60] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, Z. Xu, Source code summarization with structural relative position guided transformer, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 13–24.
  - [61] J. Son, J. Hahn, H. Seo, Y.-S. Han, Boosting code summarization by embedding code structures, in: Proceedings of the 29th International Conference on Computational Linguistics, 2022, pp. 5966–5977.
  - [62] A. LeClair, S. Jiang, C. McMillan, A neural model for generating natural language summaries of program subroutines, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 795–806.
  - [63] W. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, A transformer-based approach for source code summarization, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 4998–5007.
  - [64] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al., A survey of large language models, arXiv preprint arXiv:2303.18223 (2023).
  - [65] J. Liu, C. S. Xia, Y. Wang, L. Zhang, Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, arXiv preprint arXiv:2305.01210 (2023).
  - [66] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, X. Li, Deep is better? an empirical comparison of information retrieval and deep learning approaches to code summarization, ACM Transactions on Software Engineering and Methodology (2023).

**Junjie Zhao** is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include software repository mining.

**Xiang Chen** received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the Department of Information Science and Technology, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Software Testing, Verification and Reliability, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software - Practice and Experience, Automated Software Engineering, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER). His research

interests include software engineering, in particular software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information about him can be found at:

<https://smartse.github.io/index.html>.

**Guang Yang** received the B.S. degree from Nantong University in 2019 and the M.Sc. degree from Nantong University in 2022. Now he is working on his Ph.D. degree at Nanjing University of Aeronautics and Astronautics. He has authored or co-authored more than 10 papers in refereed journals or conferences, such as ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Journal of Systems and Software, Knowledge-based Systems, IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), International Conference on Software Maintenance and Evolution (ICSME), Asia-Pacific Symposium on Internetworkware (Internetworkware), International Conference on Software Engineering and Knowledge Engineering (SEKE), and Asia-Pacific Conference on Software Engineering (APSEC). His research interests include software engineering, especially automatic code generation and empirical software engineering.

**Yiheng Shen** is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include software repository mining.