Learning to Generate Structured Code Summaries from Hybrid Code Context

Ziyi Zhou†, Mingchen Li†, Huiqun Yu*, Senior Member, IEEE, Guisheng Fan*, Penghui Yang, Zijie Huang

Abstract—**Code summarization aims to automatically generate natural language descriptions for code, and has become a rapidly expanding research area in the past decades. Unfortunately, existing approaches mainly focus on the "one-to-one" mapping from methods to short descriptions, which hinders them from becoming practical tools: 1) The program context is ignored, so they have difficulty in predicting keywords outside the target method; 2) They are typically trained to generate brief function descriptions with only one sentence in length, and therefore have difficulty in providing specific information. These drawbacks are partially due to the limitations of public code summarization datasets. In this paper, we first build a large code summarization dataset including different code contexts and summary content annotations, and then propose a deep learning framework that learns to generate structured code summaries from hybrid program context, named StructCodeSum. It provides both an LLM-based approach and a lightweight approach which are suitable for different scenarios. Given a target method, StructCodeSum predicts its function description, return description, parameter description, and usage description through hybrid code context, and ultimately builds a Javadoc-style code summary. The hybrid code context consists of path context, class context, documentation context and call context of the target method. Extensive experimental results demonstrate: 1) The hybrid context covers more than 70% of the summary tokens and significantly boosts the model performance; 2) When generating function descriptions, StructCodeSum outperforms the state-of-the-art approaches by a large margin; 3) According to human evaluation, the quality of the structured summaries generated by our approach is better than the documentation generated by Code Llama.**

Index Terms—**Code summarization, Program comprehension, Deep learning.**

I. Introduction

EXISTING studies have shown that developers spend more than half of their time on program comprehension activities during software development and maintenance [1, 2]. To alleviate the developers' cognitive efforts in comprehending programs, a text summary accompanying the source code has been proven to be useful [3]. High-quality code summaries can not only help developers understand programs, but also benefit important tasks like code search [4–6] and code categorization [7]. However, code summaries are often missing, incomplete, or outdated in practice due to the huge effort needed in commenting and documenting source codes as well as the rapid update of the software. Code summarization aims to automatically generate natural language descriptions for code, and has become a rapidly expanding research area in the past decades. Leading approaches to code summarization mainly refer to the encoder-decoder framework from neural machine translation, which encode code tokens or structures to latent representations and then use a decoder with attention mechanism to generate code summaries from such representations [8–17]. Recently, large language models (LLMs) pre-trained on massive unlabelled corpora of code are also skilled in code summarization [18–22]. They are typically trained using autoregressive objective for token generation, and can generate code summaries either in a zero-shot manner [22] or through in-context learning without any parameter update [23, 24], i.e., by providing several demonstrations of training examples in the input prompt.

Although these code summarization methods are good at extracting key information within source codes and generating fluent natural language summaries, they still exist two major issues:

1) In terms of model input, almost all neural models generate summary only according to the given target code snippet itself, e.g., a method. However, in practice, the functionality and behavior of the code are determined by the interactions of different subroutines in a software project. Therefore, the content of the desired summary may not directly appear in the target code but in its program context. Although code context is crucial for code summarization, there is currently a lack of study on utilizing and analyzing different types of code contexts. Existing context-aware methods only consider some specific code contexts, such as class or file context [25], call context [26], and randomly extracted project context [27], with limited improvements against conventional approaches that do not use code context.
2) In terms of model output, existing deep learning-based models typically aim to generate brief function descriptions for codes. However, developers often need to know more detailed and specific information about the target codes, such as the meaning and usage of method parameters and return values. In contrast, the generated summaries from most neural models are not rich, specific, and standardized in content, therefore

All authors are with the Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China. E-mails: {zhouzy,lmc}@mail.ecust.edu.cn, {yhq, gsfan}@ecust.edu.cn, {yph, hzj}@mail.ecust.edu.cn

Ziyi Zhou is also with the Institute of Natural Sciences & Shanghai National Centre for Applied Mathematics (SJTU Center), Shanghai Jiao Tong University, Shanghai 200240, China.

*: Huiqun Yu and Guisheng Fan are corresponding authors.

†: Ziyi Zhou and Mingchen Li contribute equally to this work.

lacking usefulness. On the other hand, although LLMs are intrinsically able to predict full method documentation, they suffer from hallucination [28] and their outputs have not been quantitatively evaluated yet.

One of the main causes is that the widely-used code summarization datasets do not contain program context as well as different types of summary content for each target code. In most public datasets, the input of each example is a method, and the expected output is the first sentence of its documentation comment (doc comment) [10, 11, 29], which makes the follow-up studies focus on the "one-to-one" mapping problem between code and summary. In addition, due to the varied quality of open source projects, the doc comments are often not compliant with specifications such as Javadoc. Thus, the first sentence of the documentation may not be the expected target to predict, which becomes a potential threat to validity. For example, in the dataset built by this work, more than 25% of the doc comments do not start with an overall function description (e.g., start with a parameter description or example code instead).

To address the above issues, we first build a large code summarization dataset that includes different code contexts and summary content annotations to support model training and evaluation. It is extracted from the Java projects selected by CodeSearchNet [30] from GitHub. In this dataset, available code contexts include path contexts, class contexts, call contexts, and documentation contexts. The summaries are split and categorized into function descriptions, return descriptions, parameter descriptions, exception descriptions, and usage descriptions. The dataset contains 620,226 examples of <method, contexts, annotated documentation> after exhaustive preprocessing and filtering. Then we propose StructCodeSum, a deep learning framework that learns to generate structured code summaries from hybrid code context. It provides an LLM-based approach and a lightweight approach for this task, which differ in their backbone models but share a unified workflow. In this framework, a target method and its hybrid code context are converted to instruction as model input, and the underlying model is trained to generate different descriptions through instruction tuning [31]. During inference, the instructions are used to query the model for the desired descriptions of the target method, and a structured code summary in Javadoc style is finally built. The example in Fig. 1 shows the differences between StructCodeSum and previous work in terms of model input and output.

We evaluate StructCodeSum through extensive experiments on the constructed dataset and analyze the impact of different code contexts on summary generation. Statistical analysis shows that our hybrid code context covers more than 70% of the tokens in the code summary, while the target method itself only covers about 36%. StructCodeSum significantly outperforms the state-of-the-art approaches in generating function descriptions, and the structured code summaries provided by StructCodeSum have higher quality than the documentation generated by the LLM baseline. In addition, we find that our lightweight approach is much faster and cheaper than our LLM-based approach without sacrificing much accuracy.

The contributions of this work are as follows.

1) We build a new code summarization dataset with various code contexts and summary content annotations, which may support future studies.
2) We propose StructCodeSum, the first neural code summarization framework that learns to generate structured code summaries from hybrid code context. It successfully combines the advantages of both deep learning and template-based methods.
3) We provide both an LLM-based solution and a lightweight solution in StructCodeSum. They are suitable for different scenarios when considering computational costs.
4) Extensive experiments show that the hybrid code context we adopt effectively boosts the model performance on code summarization, and StructCodeSum significantly surpasses the state-of-the-art models.

II. Related Work

Most of the recent studies on code summarization are data-driven and deep learning-based, applying different neural networks to represent source codes. These approaches can be further divided into four categories: to better learn code representations, to improve performance with auxiliary tasks, to utilize external knowledge of the target code, and to adopt LLMs.

A. Learning Code Representations

Iyer et al. [32] propose the first neural code summarization model. They use an RNN with an attention mechanism to produce summaries directly from token embeddings of codes. Allamanis et al. [33] propose a convolutional attention network to encode token sequences and generate function name-like summaries. Hu et al. [34] customize a traversal method called SBT to serialize ASTs and then apply a standard Seq2Seq model. Alon et al. [35] represent source code as a set of AST paths and encode them via RNN, and use attention to focus on relevant paths during decoding. Wan et al. [9] employ a Tree-RNN to model AST and an RNN to model token sequence of code, and then integrate their attention vectors to generate summaries. They also exploit reinforcement learning to cope with exposure bias. LeClair et al. [36] and Hu et al. [10] both combine the encoding results of SBT and token sequence to generate code summaries. There are also increased research interests in learning graph representations of code through GNN or modified Transformer. Graph representations of code can be built from AST [13, 37], dataflow graph [17, 38], or the more complicated code property graph [14]. Another branch of study aims to improve code summarization through hierarchical code representations [39–41].

B. Multi-task Learning Approaches

Different code-related tasks are possible to share their knowledge to improve them jointly. Chen et al. [4] propose a Bimodal Variational Auto Encoder to project natural language and code into a common semantic space, which could be used for both code retrieval and summarization tasks. In [5], the

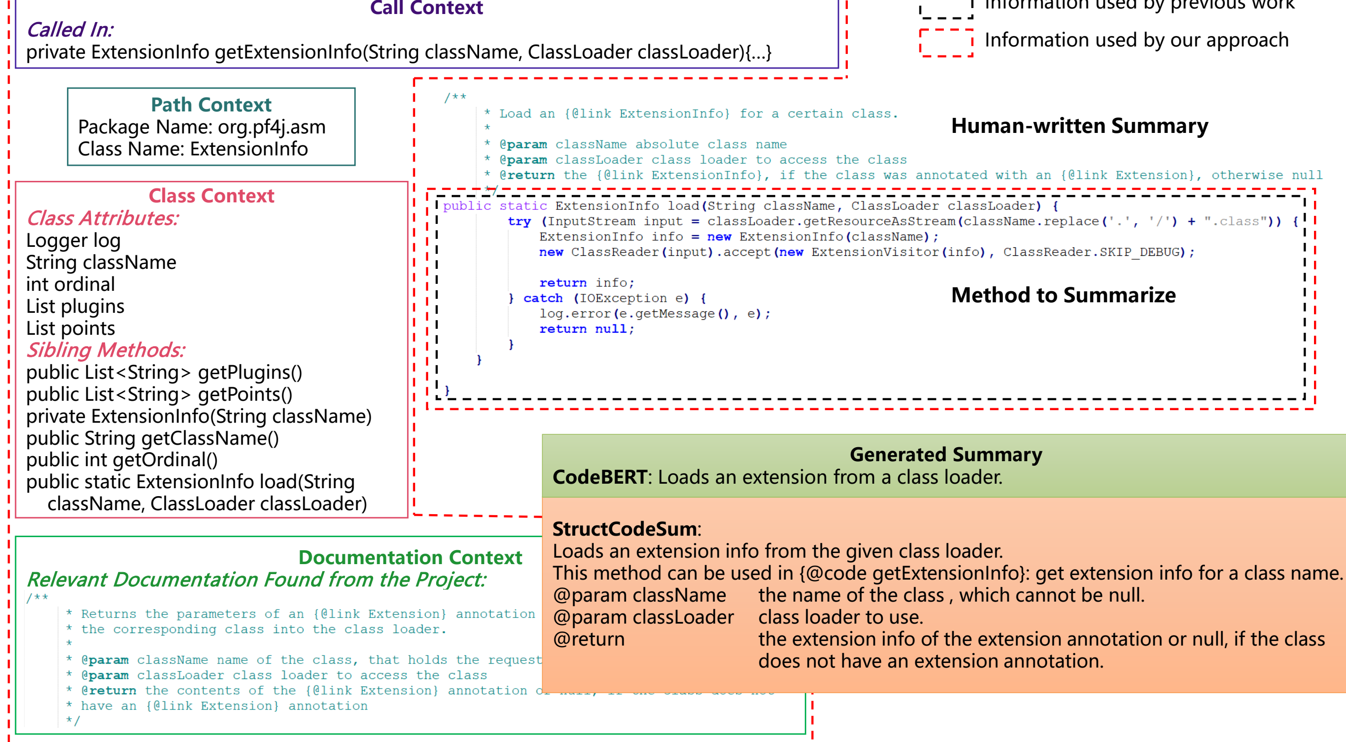


Fig. 1: An example to show the differences between previous work and StructCodeSum

authors investigate a novel perspective of code annotation for code retrieval, where a code annotation model is trained to generate code summaries that can be leveraged by a code retrieval model to better distinguish relevant codes. With dual learning, code summarization and generation models can be trained simultaneously to exploit their duality [6, 42]. Xie et al. [43] exploit the task of method name prediction to improve code summarization.

C. Utilizing External Knowledge

This type of work leverages information outside the target code snippet to help generate its summary. Zhang et al. [44] propose a retrieval-based neural code summarization approach where the neural model takes the most similar code snippets retrieved from the training set as extra inputs. Except for the similar code snippet, Wei et al. [15] further encode its paired comment (called exemplar) and the SBT sequence of the target code. Their work is later improved by Li et al. [45] by learning the edits between the retrieved code and the target code to better revise the exemplar. Zhou et al. [46] propose a code summarization framework based on meta-learning and code retrieval, which aims to optimize a unique code summarizer for each target code snippet using its similar examples as the few-shot training set. Haque et al. [25] use the file context, i.e., the methods within the same file as the target code, as additional input of the target code. Bansal et al. [27] encode randomly selected files from the project to provide project-level context for code summarization. Liu et al. [26] propose a code summarization model using the knowledge of the call dependencies between the target code and its related methods. Aghamohammadi et al. [47] generate summaries for event-driven programs by analyzing the dynamic call graphs via PageRank and applying a pre-trained code summarization model.

D. LLMs for Programming Languages

LLMs, which often contain billions of parameters and are pre-trained on massive unlabelled corpora, have achieved great success in various code-related tasks. They can be categorized into encoder-only models such as CodeBERT [48, 49], encoder-decoder models such as CodeT5 [19, 50] and PLBART [51], and decoder-only models such as Codex [18] and GPT [21, 52]. Due to the strong ability of decoder-only LLMs in both natural language generation and in-context learning, they can be queried to generate code summaries without fine-tuning by feeding them proper prompts [22–24]. However, there is still a lack of research on how to exploit the strength of LLMs to accurately generate structured code summaries. In this study, we adopt Code Llama [20], an open-source LLM designed for code, as the backbone model of StructCodeSum for empirical study.

E. Public Code Summarization Datasets

One of the most important factors affecting the performance and evaluation of deep learning models is the dataset. Popular code summarization datasets are crawled from GitHub[1] or Stackoverflow[2]. Since the quality of open-source projects

[1]https://github.com/
[2]https://stackoverflow.com/

TABLE I: Public code summarization datasets

Literature	Language	Input	Output	Cross-Project	Duplicates Removed
Iyer et al. [32]	C#	Code snippet	Post title	✗	✗
	SQL	Query	Post title	✗	✗
Hu et al. [34]	Java	Method	The first sentence of the documentation	✗	✗
Hu et al.[11]	Java	Method	The first sentence of the documentation	✗	✗
	Java	API Sequence	The first sentence of the documentation	✗	✗
LeClair et al. [29]	Java	Method	The first sentence of the documentation	✓	✓
Li et al. [54]	Java	Class	The first sentence of the documentation	✓	✓
Barone et al. [55]	Python	Method	Documentation	✓	✗
Husain et al. [30]	Multiple	Method	The first paragraph of the documentation	✗	✓
Lu et al. [56]	Multiple	Method	The first paragraph of the documentation	✗	✓
This work	Java	Method + Contexts	Documentation with content annotations	✓	✓

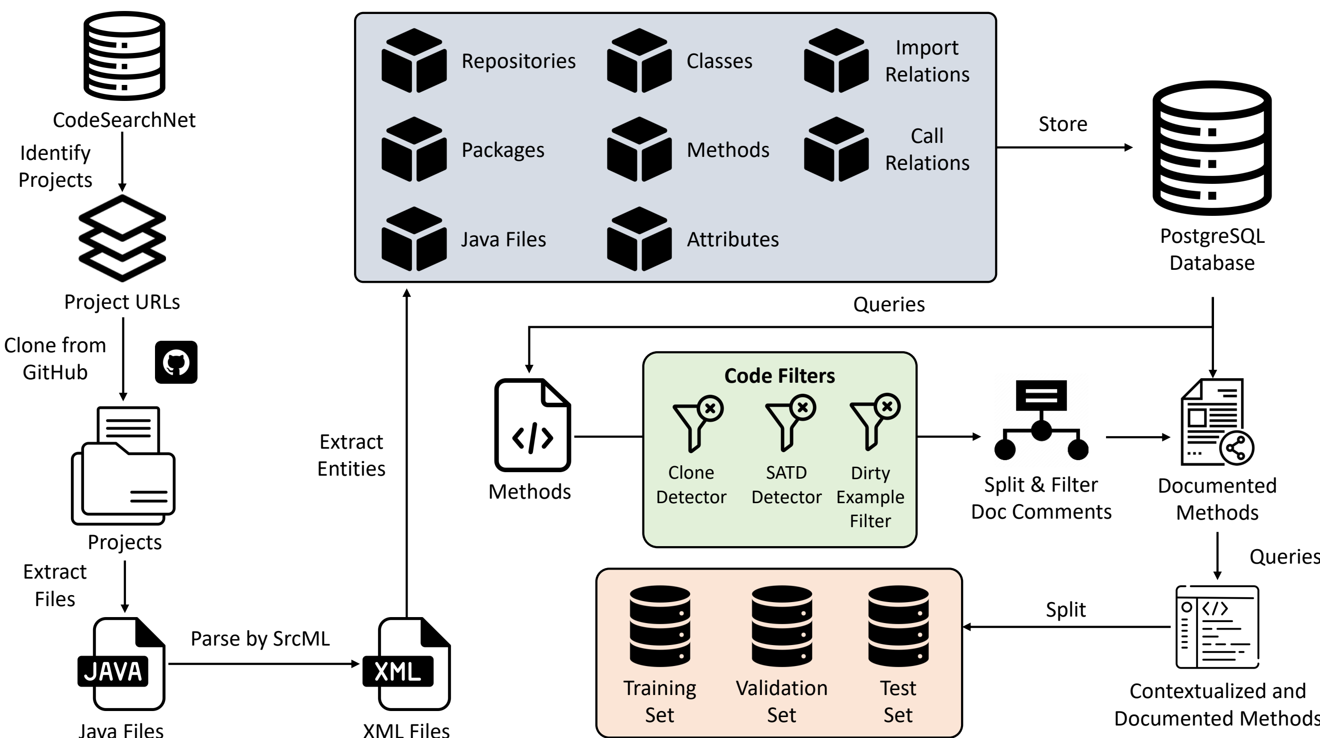


Fig. 2: Overall process to build the dataset

varies, there are many noisy elements in these datasets [53]. Furthermore, the input and output of the examples in these datasets are too simple to train a practical code summarization model, as described in Section I. These motivate us to build a new dataset for code summarization. Table I shows the main characteristics of the previous datasets and ours, where **Cross-Project** denotes whether the training, validation, and test sets are split by project.

III. Dataset Preparation

A. Data Collection & Preliminary Filtering

Our corpus is extracted from the Java projects selected by CodeSearchNet [30], which are originally collected from GitHub according to popularity (stars and forks) and license. CodeSearchNet is one of the most widely-used benchmarks for both code search and code summarization, and has been used for pre-training several popular programming language models [48–50], so we believe this is a good dataset to start with. The overall process of building our dataset is shown in Fig. 2. We first obtain the repository URLs of all Java projects in CodeSearchNet, and then use `git clone` to download the latest versions of the repositories (as of March 1, 2022). After that, we use SrcML [57] to perform static analysis on the source code files and convert them into XML files. Xpath is then used to obtain specific data such as methods, documentation, classes, call relations, and package names. These data are stored in a PostgreSQL database, followed by preliminary filtering of the methods:

1) Remove the methods with no or empty documentation.
2) Remove (near) duplicate methods using the duplication detector proposed by Allamanis [58].
3) Remove the methods containing self-admitted technical

debts (SATD) using the SATD detector proposed by Liu et al. [59].
4) Remove the methods with less than two statements.
5) If a method signature contains the keyword "test", this method is considered as test code and removed.
6) If a doc comment contains the keyword "generated", this method is considered as automatically generated [60] and removed.

For each of the remaining methods, we use SQL queries to obtain its different contexts, including its package name, class signature, class attributes, siblings (neighboring methods within its enclosing class), callees (methods called by it), and callers (methods that call it).

B. Splitting Doc Comments

According to Javadoc guidance[3], a doc comment is made up of several blocks describing different aspects of the code. The first block should be an overall description of the method, starting with a short summary sentence, while each subsequent block should start with a block tag indicating its content type, e.g., `@param`, `@return`, `@exception`. Unfortunately, due to varied project quality, most doc comments on GitHub do not conform to certain specifications, which are usually incomplete such as missing necessary tags. On the other hand, not everything in the documentation is suitable for training and evaluating neural models. Therefore, we design a set of rules to identify different types of elements within doc comments and choose a part of them to build our dataset for structured code summarization.

1) Categorize Untagged Sentences: Given a doc comment, we first split it into blocks by existing Javadoc tags. A block is then split into a sequence of sentences so that we can check their respective content type. For each sentence, we use heuristics to determine which of the following contents it belongs:

- Example code
- SATD description
- Return description
- Exception description
- Usage description
- Summary sentence

Detailed information can be found in Appendix A. To avoid confusion, we call the summary sentence as the **function description** of a method.

2) Preprocessing and Filtering: With the help of existing Javadoc tags and the rules above, we pick the function, usage, parameter, exception, and return descriptions from the code documentation to build the dataset. After that, to make these descriptions more concise, natural, and friendly for evaluation, the following preprocessings are performed:

1) Replace non-integer numbers and hash values with a special token `<num>`.
2) Replace links with a special token `<url>`.
3) Supplementary notes appearing alongside brackets, "e.g." or "i.e." are removed.
4) HTML tags are removed.
5) The identifiers that appear in the doc comments are further split into subtokens according to camelCase and snake_case.
6) Truncate each type of description to the first sentence.
7) The first word of a return description will be removed if it is "return" or "returns".
8) The first word of an exception description will be removed if it is "throw" or "throws".
9) If more than half of the words in a doc comment are non-English, this example will be discarded.

[3]https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

After the above steps, the descriptions containing at least three tokens are considered informative and kept, while the ones with more than 30 tokens are truncated. If we cannot keep any description of a method, this example will be discarded. Note that a method may contain multiple parameters and exceptions so the number of corresponding descriptions can be more than one.

C. Dataset Statistics

Finally, we build a dataset with a total of 620,225 <method, contexts, annotated documentation> triples. The overall statistics of source codes in our dataset are shown in Table II, where # denotes "The number of". To avoid confusion, we denote the method with documentation for prediction as the target method. The count of calls in Table II is the sum of callers and callees of a method. The overall statistics of code descriptions are shown in Table III. We can see from Table III that not all code documentation contains a function description. Specifically, 25.2% of the documentation in our dataset has no proper function description, whose first sentences are identified as other types of descriptions or filtered by the rules above. This suggests that directly using the first sentence in the doc comment as the prediction target is a potential threat to validity. Besides, 10.5% and 2.9% of the return and exception descriptions are untagged and identified by rules, respectively. Table A-I shows the statistics of code lengths, context sizes, and description lengths after preprocessing, where **Std.** denotes standard deviation. The average length of parameter descriptions is the shortest among the five types of descriptions, while that of usage descriptions is the longest. According to the standard deviations, the target method length, the number of callers of the target methods, and the number of methods within the class vary greatly.

Through observations, we find that the exception and usage descriptions are not suitable for model training and evaluation due to both their data size and content. These two descriptions always involve many implementation details of the program, and are unpredictable in their original content. For example, many exceptions can only be figured out during runtime, and a usage description may be a note on corner cases, e.g., "Do not use, for internal use only". Actually, even experienced developers can hardly predict these descriptions only from the program contexts. As a result, exception and usage descriptions are excluded from our experiments, but we still keep them in the released dataset to support future research.

TABLE II: Overall statistics of source codes in our dataset

Target Methods			Contexts of the Target Methods				
# Methods	# Tokens	# Lines	# Projects	# Classes	# Attributes	# Class Methods	# Calls
620,225	40,123,925	6,226,876	4,244	153,659	823,503	2,067,575	1,308,471

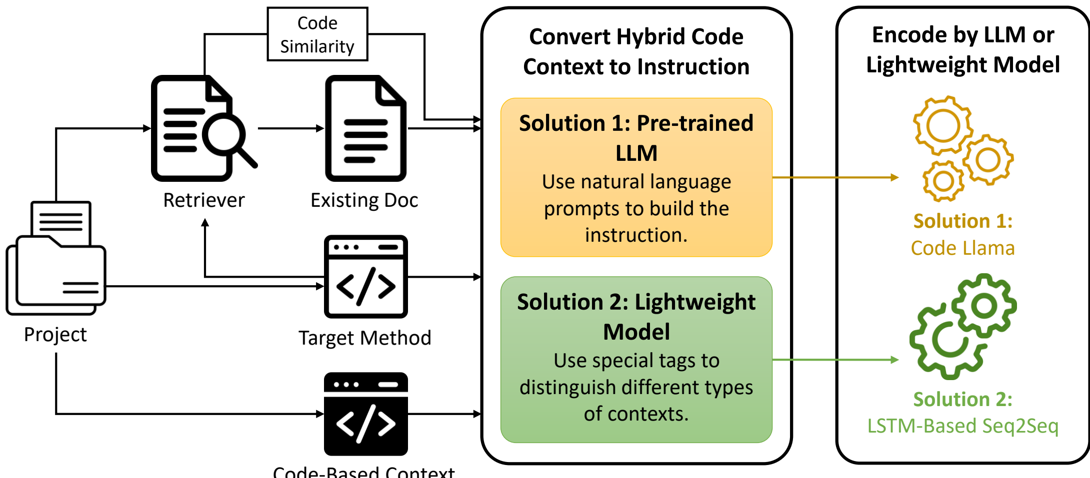


Fig. 3: Input of StructCodeSum

TABLE III: Overall statistics of code descriptions in our dataset

Name	Count
# Documentation comments	620,225
# Function descriptions	464,143
# Parameter descriptions	388,952
# Methods with parameter descriptions	258,313
# Return descriptions (identified by rules)	334,300 (35,156)
# Exception descriptions (identified by rules)	61,359 (1,755)
# Methods with exception descriptions	44,749
# Usage descriptions	21,237
# Total words	11,197,594

We randomly split the dataset into training, validation, and test sets in the proportions of 8:1:1 under different strategies: by target method and by project. The former is a common split setting, called mixed-project splitting. The latter is called cross-project splitting, which makes it impossible for codes from the same project to appear in both training and testing data. Table A-II shows statistics of the dataset split by different strategies. As seen, the counts of descriptions under different settings have no significant difference. We make our dataset publicly available[4].

IV. Proposed Approach

A. Overview

StructCodeSum provides two solutions for generating structured code summaries: an LLM-based approach and a lightweight approach. These two approaches mainly differ in their backbone models where the former adopts Code Llama [20] and the latter is built by LSTM. They have different inputs but share a unified workflow. In general, the contexts fed into the model can be divided into two types: code-based context and documentation context. Code-based context includes the attributes and sibling methods within the same class as the target method, as well as its path. Documentation context denotes the relevant documentation found in the project where the target method is located. These inputs together with the target method are converted to an instruction so that the model can identify different contents, as illustrated in Fig. 3.

The overall training and testing process of StructCodeSum is shown in Fig. 4, where the black and orange arrows denote the information flows during training and testing, respectively. Instead of learning to generate all descriptions of a method at once, StructCodeSum treats the generation of each type of description as a separate task and trains the underlying model through instruction tuning [31], where specific instruction templates are designed for different tasks. During testing, the instructions are used to query the model to produce corresponding descriptions for a given method. Moreover, we use PageRank to select the most important caller of the target method, and generate its function description to form the usage description of the target method.

B. Model Input

1) Code-Based Context: **Path context** refers to the package name of the target method plus the name of its enclosing class. Packages are used to avoid naming conflicts and to organize code files into different directories [61], which are adopted by most programs. In practice, the package name usually relates to the purpose of the classes in the package; for example, it contains the names of function modules and businesses. Therefore, package names may be able to abstract the framework of a program in terms of functionality and design, as well as assist developers in understanding and maintaining software more efficiently [62].

[4]https://github.com/zy-zhou/StructCodeSum

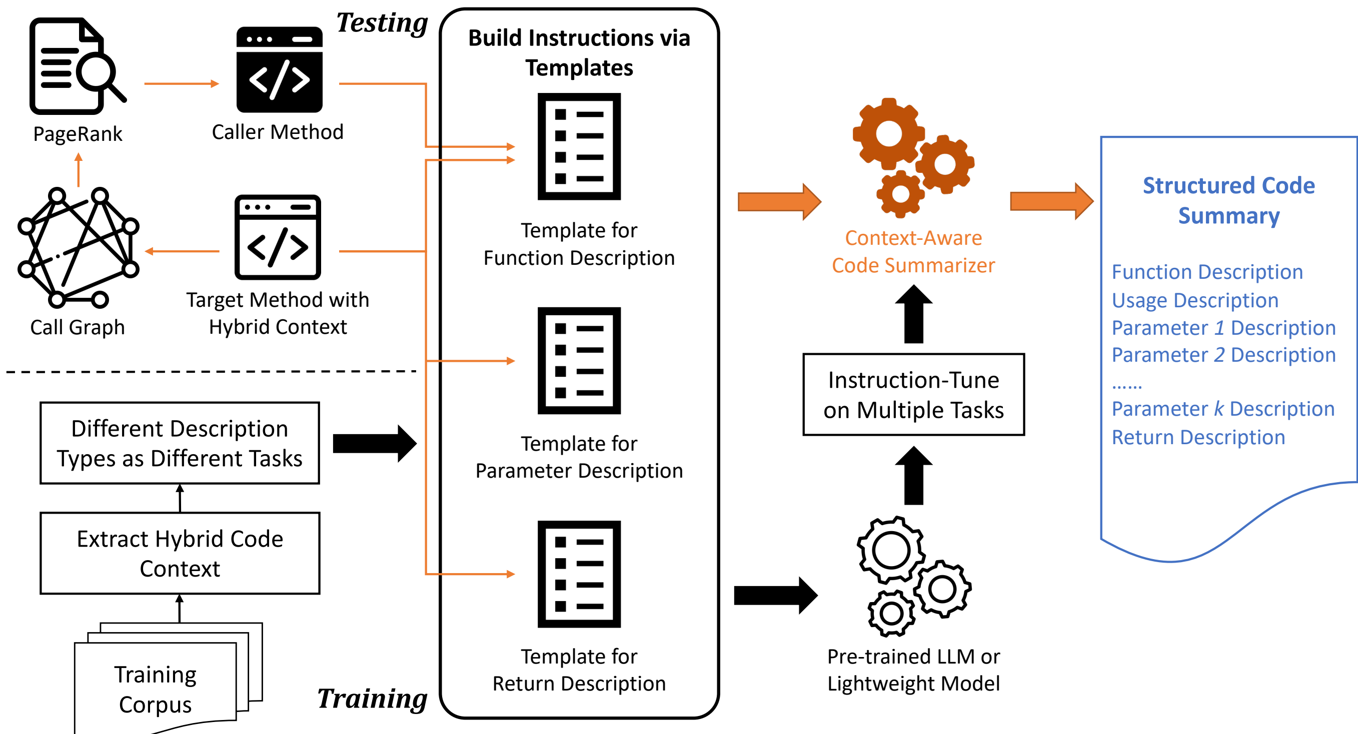


Fig. 4: Training and testing process of StructCodeSum

Class context of the target method includes the attributes and sibling methods within its enclosing class, which is potentially helpful in characterizing the target method. However, it is computationally expensive to feed full of these surrounding contents into the neural model since a class can be very large. On the other hand, there is no guarantee that the complete class context can be obtained in practice. Therefore, we adopt a compromising approach: extracting the names of the attributes and the signatures of the sibling methods.

Since most of the target methods do not have a caller or callee method (the median of the number of both callers and callees is 0 as shown in Table A-I), combining the call context with other types of contexts will cause the imbalance of model input, which may negatively affect model training. Therefore, the call context is not input to the model for training. We will show how to use the call context to generate usage descriptions in Section IV-C2.

2) Documentation Context: The doc comments within the project of the target method are ready-made summaries of its contexts, so they are quite helpful in generating the target summary. However, irrelevant documentation can mislead the model. To solve this issue, we search for the documented method most similar to the target one from its project and use its documentation as the documentation context. The retrieval process is based on Vector Space Model (VSM) [63]. In detail, each method in the project is represented as a vector via Term Frequency-Inverse Document Frequency (TF-IDF). Since the number of methods within a project is limited, the similarity of two methods $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ can be efficiently computed by cosine rather than a search engine:

$$\cos(\boldsymbol{r}^i, \boldsymbol{r}^j) = \frac{\boldsymbol{r}^i \cdot \boldsymbol{r}^j}{\|\boldsymbol{r}^i\|\|\boldsymbol{r}^j\|} \in [-1, 1] \tag{1}$$

where $\boldsymbol{r}^i$ and $\boldsymbol{r}^j$ are TF-IDF vectors of the two (tokenized) methods. After obtaining the most similar method, we further compute a character-level similarity between it and the target method based on text edit distance following [44, 46]:

$$\mathrm{sim}(\boldsymbol{x}^i, \boldsymbol{x}^j) = 1 - \frac{\mathrm{dis}(\boldsymbol{x}^i, \boldsymbol{x}^j)}{\max(|\boldsymbol{x}^i|, |\boldsymbol{x}^j|)} \tag{2}$$

where *dis* denotes text edit distance. This fine-grained similarity will be used by the model instead of the cosine value.

3) Convert Code Contexts to Instructions: LLMs are notable for their ability to understand natural language instructions [21, 31]. Therefore, to build the input for Code Llama, we use natural language prompts to organize the above code contexts along with the target method, as detailed in Appendix B1. For the lightweight model, we instead insert special tags into the subtoken sequence of the input contexts to indicate different contents, as detailed in Appendix B2.

C. Learning to Generate Structured Code Summaries

1) Training by Instructions: A straightforward way for learning to generate structured code summaries is to concatenate different descriptions of a method into a single summary and treat it as the target sequence for training. But this is problematic due to the following reasons. First, a method may not have some types of descriptions either because the documentation is incomplete, or it has no corresponding code

element such as return statement and formal parameter. This will make the model unable to know when to generate a certain type of description. Second, concatenating all descriptions in the documentation leads to much longer target sequences. Since the model recursively takes its previously generated tokens as input, the error in previous steps accumulates and can lead to worse performance when producing long sequences. Therefore, StructCodeSum learns to generate different types of descriptions separately in a multi-task manner, which is achieved by instruction tuning [31]. In this way, the model can better exploit the parallel corpus of code and documentation.

In detail, each training example is initially a triple <method, contexts, description sentence>. It should be noted that a method may come with multiple parameter descriptions, and we assign them to different examples. In order to prompt the model to generate different types of descriptions, the method and its contexts are organized according to different instruction templates. The instruction templates for the LLM-based approach and lightweight approach are elaborated in Appendix B1 and Appendix B2 respectively. Subsequently, the examples of different descriptions are mixed together to train the model via maximum likelihood estimation. Denoting the built instruction and target description as $\hat{\boldsymbol{x}}$ and $\boldsymbol{y}$, the training objective is to minimize the following loss function:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\log p_\theta(y_t^i|y_{<t}^i, \hat{\boldsymbol{x}}^i) \tag{3}$$

where θ is the learnable model parameters, N is the number of training examples, T_i is the length of the i-th target description.

Notably, instruction tuning is originally a fine-tuning technique developed for LLMs but not for un-pretrained small models. Considering the lower capacity of the lightweight model, we further fine-tune it on the corpus of function, parameter and return descriptions respectively with a lower learning rate, and obtain three expert code summarizers for prediction. We find this outperforms either training three expert models from scratch or instruction tuning a single model for use (Section VI-B).

2) Generating Summaries: Given a contextualized target method, we convert it to different instructions and feed it to the trained model. Then we can get different descriptions corresponding to the chosen instruction templates. We first obtain its function description, denoted as $\hat{\boldsymbol{y}}^{fun}$. If it has formal parameter(s), we then generate description $\hat{\boldsymbol{y}}^{par_i}$ for its i-th parameter. If it has return statement(s), we further generate its return description $\hat{\boldsymbol{y}}^{ret}$. If it is explicitly called by other methods in the project, we apply PageRank on its call graph to identify its most important caller following [47]. Given a graph, PageRank calculates a rank r_i to a node i by:

$$r_i = \sum_{j\in B_i}\beta\frac{r_j}{d_j} + (1-\beta)\frac{1}{n} \tag{4}$$

where B_i is the set of nodes which points to i, d_j is the number of outgoing edges from node j, β is a normalization factor below 1, and n is the number of nodes in the graph. From the nodes (methods) that point to the target method, the one with the highest rank is selected, denoted as $\boldsymbol{u}$. Then we generate a function description $\hat{\boldsymbol{y}}^u$ for the selected caller by treating it as the target method. Finally, we build a Javadoc-style code summary for the target method according to the following template:

$\hat{\boldsymbol{y}}^{fun}$.
This method can be used in u^{name} : $\hat{\boldsymbol{y}}^u$.
@param x^{par_1} : $\hat{\boldsymbol{y}}^{par_1}$.
@param x^{par_2} : $\hat{\boldsymbol{y}}^{par_2}$.
......
@param x^{par_k} : $\hat{\boldsymbol{y}}^{par_k}$.
@return $\hat{\boldsymbol{y}}^{ret}$.

where u^{name} is the method name of $\boldsymbol{u}$, x^{par_i} is the name of the i-th target parameter, and k is the number of target parameters.

D. Backbone Models

1) Pre-trained LLM: We adopt Code Llama [20] as the backbone model of our LLM-based approach. It is a decoder-only Transformer pre-trained using an auto-regressive objective. Based on Llama2 [36], it is further pre-trained on 500 billion tokens, with 85% of these tokens consisting of code and 15% of natural language. This enables it to significantly outperform Llama2 on code-related tasks. In particular, Code Llama includes a fine-tuned version which is trained using an additional 5 billion instruction tokens, named CodeLlama-Instruct. This model can perform tasks such as code summarization, refinement, and translation in a zero-shot manner by following input instructions. However, we find that it exhibits worse performance than our lightweight model in utilizing program context through zero-shot inference (Section VI-B). This is the reason why we resort to instruction tuning. In this work, we select the CodeLlama-Instruct-7B version due to our limited computational resources.

During training, given an instruction $\hat{\boldsymbol{x}}$ and a target description $\boldsymbol{y}$, we concatenate them to form the input sequence of Code Llama. We apply Low-Rank Adaptation (LoRA) [64] to efficiently optimize Code Llama towards Eq. 3. LoRA injects trainable rank decomposition matrices into the backbone model with its original pre-trained parameters frozen. Namely, given a pre-trained weight matrix $\boldsymbol{W}_0 \in \mathbb{R}^{d\times k}$, LoRA constrains its update by representing the latter with a low-rank decomposition:

$$\boldsymbol{W}_0 + \Delta\boldsymbol{W} = \boldsymbol{W}_0 + \boldsymbol{B}\boldsymbol{A} \tag{5}$$

where $\boldsymbol{B} \in \mathbb{R}^{d\times r}$, $\boldsymbol{A} \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$. $\boldsymbol{A}$ is randomly initialized but $\boldsymbol{B}$ is initialized by zero. We freeze the original model and apply LoRA to the weight matrices in its self-attention modules and feed-forward layers.

2) Lightweight Model: The lightweight model is an LSTM-based Seq2Seq model. Unlike LLMs, small models may not be capable of handling long inputs. Therefore, we equip it with three encoders for reading different contents in an instruction: a method encoder for the target method, two context encoders for the code-based context and documentation context respectively. Each of them is a bidirectional LSTM (biLSTM) with

an embedding layer, and takes a sequence of (sub)tokens as input. In particular, the initial state of the method encoder is the final state of the code-based context encoder, considering that the representation of the target method should be conditioned on its surrounding context. The encoders have different sets of trainable parameters, except for the shared embedding layer between the method encoder and code-based context encoder. Fig. A-2 illustrates the architecture of the lightweight model. The encoded representations of the target method, code-based context and documentation context are denoted as $\boldsymbol{h}^x$, $\boldsymbol{h}^q$ and $\boldsymbol{h}^d$, respectively.

The decoder is a unidirectional LSTM, which uses $\boldsymbol{h}^*$ and its previous outputs to predict a target summary $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$ one element a time. At time step t, its hidden state $\boldsymbol{s}_t$ is updated by:

$$\boldsymbol{s}_t = \text{LSTM}(\boldsymbol{s}_{t-1}, [y_{t-1}; \boldsymbol{c}_{t-1}]) \tag{6}$$

where $\boldsymbol{c}_{t-1}$ is a context vector computed by the parallel attention mechanism. The last backward hidden states of the method encoder and the documentation context encoder are combined to form the initial decoder state, based on the similarity of the target code $\boldsymbol{x}$ and the retrieved one $\boldsymbol{x}^r$:

$$\boldsymbol{s}_0 = \overleftarrow{\boldsymbol{h}}^x_{-1} + \text{sim}(\boldsymbol{x}, \boldsymbol{x}^r) \cdot \overleftarrow{\boldsymbol{h}}^d_{-1} \tag{7}$$

where *sim* is the similarity metric described in Eq. 2. The parallel attention is the modification of the multiplicative attention in [65]. Specifically, independent attention mechanisms are performed on the three encoded representations. Take the one between $\boldsymbol{s}$ and $\boldsymbol{h}^x$ for example, a context vector $\boldsymbol{c}^x_t$ is computed as a weighted sum of the elements in $\boldsymbol{h}^x$:

$$\begin{aligned} \boldsymbol{c}^x_t &= \sum_{j=1}^{n} \alpha_{tj} \boldsymbol{h}^x_j \\ \alpha_{tj} &= \frac{\exp(\boldsymbol{s}_t^\top \boldsymbol{W}^x_a \boldsymbol{h}^x_j)}{\sum_{k=1}^{n} \exp(\boldsymbol{s}_t^\top \boldsymbol{W}^x_a \boldsymbol{h}^x_k)} \end{aligned} \tag{8}$$

where $\boldsymbol{W}^x_a$ is the parameter matrix and n is the input length of the method encoder. The context vectors for the other two inputs can be obtained accordingly, denoted as $\boldsymbol{c}^q_t$ and $\boldsymbol{c}^d_t$, respectively. Then, a global context vector $\boldsymbol{c}_t$ is computed as:

$$\boldsymbol{c}_t = \boldsymbol{W}_c[\boldsymbol{c}^x_t; \boldsymbol{c}^q_t; \text{sim}(\boldsymbol{x}, \boldsymbol{x}^r) \cdot \boldsymbol{c}^d_t] + \boldsymbol{b}_c \tag{9}$$

where $\boldsymbol{W}_c$ and $\boldsymbol{b}_c$ are parameter matrix and bias. Finally, the conditional probability of generating the next token y_t is:

$$p_\theta(y_t | y_{<t}, \hat{\boldsymbol{x}}) = \text{softmax}(\boldsymbol{W}_s \tanh(\boldsymbol{W}_t[\boldsymbol{c}_t; \boldsymbol{s}_t])) \tag{10}$$

V. Evaluation Setup

A. Model Training and Inference

1) Pre-trained LLM: We set the LoRA parameter r to 4, with a Dropout [66] rate of 0.05. As a result, only 6.23% parameters (420M) of the model are updated. The training hyperparameters are set according to the configurations of LLama2 [36]. The optimizer is AdamW, using a cosine learning rate schedule with an initial learning rate of 2×10^{-5} and a weight decay of 0.1. The batch size is 64 and the sequence length is 4096 tokens. We zero out the loss on the instruction tokens during training, and fine-tune the models for 2 epochs. During inference, greedy decoding and 8-bit quantization are utilized for efficiency. The experiments are conducted on a Linux server with an NVIDIA A100 GPU with 80GB V-RAM, implemented using PyTorch 2.10.

2) Lightweight Model: The subtoken embeddings and LSTM states are set to 256 dimensions. The maximum input length of the code-based context encoder is set to 500. The vocabulary sizes of codes and summaries are limited to 50K and 25K, respectively. We adopt Adam [67] as the optimizer with the learning rate of 4×10^{-4} for the initial instruction tuning and 4×10^{-5} for additional fine-tuning. The batch size is 64. We clip gradient norm by 5 to avoid exploding gradients, and apply Dropout of 0.1 on the embedding vectors and recurrent units. To prevent the models from overfitting, we stop training when their BLEU scores [68] recorded on the validation set do not improve within 4 epochs, and then choose the best models according to the validating BLEU scores. Beam search [69] is adopted during validating and testing, with the beam size and length penalty set to 5 and 1.0 respectively. The experiments are conducted on a Linux server with an NVIDIA Tesla V100 GPU with 16GB V-RAM, implemented using PyTorch 1.10.

B. Automatic Evaluation Metrics

Following existing work [9, 14, 16, 44–46], we evaluate the quality of generated summaries using BLEU [68], ROUGE-L [70], and METEOR [71]. They are also popular in machine translation and text summarization. These metrics are calculated for different description sentences but not for the whole documentation, and we report the scores in percentage. However, recent researches show that the metrics computing only word overlap such as BLEU may not be reliable proxies of human evaluation [72, 73]. Therefore, We further consider SIDE [74], an embedding-based metric designed specifically for code summarization, to measure the model performance in predicting function descriptions. The details of these metrics can be found in Appendix C. Besides, the Wilcoxon Sign-Rank tests are applied to the scores of different approaches to determine if significant performance differences exist between them, and the corresponding p values are reported.

C. Research Questions

We evaluate the proposed framework by investigating the following research questions:

RQ1: How do different types of code contexts affect code summary generation?

RQ2: How effective is instruction tuning for generating different types of descriptions?

RQ3: How does the proposed approach perform generally compared to the state-of-the-art code summarization models?

RQ4: How is the quality of the structured code summaries generated by our approach in practice?

VI. Evaluation Results

A. RQ1: Impact of Different Code Contexts

To answer RQ1, we train context-aware models with different types of code contexts for evaluation, which is achieved

TABLE IV: Model performance of StructCodeSum (LLM) using different contexts

Description	Context	Mixed-project			Cross-project		
		BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
Function	None	20.73	24.15	45.81	13.36	20.58	40.83
	Path	24.67	26.35	49.61	14.21	21.26	41.92
	Attributes	21.30	24.68	46.68	13.38	20.66	40.97
	Siblings	22.01	25.12	47.41	13.72	20.99	41.37
	Documents	41.17	34.11	62.62	23.20	23.76	45.40
	Hybrid	**42.29**	**35.01**	**63.80**	**24.65**	**24.94**	**46.87**
Return	None	31.58	28.51	51.01	16.99	22.16	43.45
	Path	35.37	31.46	55.93	18.82	23.79	45.82
	Attributes	31.96	29.29	51.97	17.01	22.39	43.59
	Siblings	32.93	30.16	53.49	17.68	23.03	44.70
	Documents	49.79	37.77	67.69	27.60	25.57	47.36
	Hybrid	**50.26**	**38.57**	**68.57**	**29.32**	**26.81**	**50.30**
Parameter	None	30.63	27.56	55.16	23.88	22.08	47.83
	Path	33.73	29.59	58.95	25.00	22.73	48.63
	Attributes	30.75	27.80	55.21	23.98	22.00	48.26
	Siblings	31.48	28.29	56.14	24.36	22.35	48.42
	Documents	49.18	37.18	67.38	35.66	28.17	53.87
	Hybrid	**49.71**	**37.59**	**67.44**	**36.98**	**29.01**	**55.11**

TABLE V: Model performance of StructCodeSum (Lightweight) using different contexts

Description	Context	Mixed-project			Cross-project		
		BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
Function	None	22.50	22.84	46.97	11.21	16.95	36.53
	Path	29.83	26.51	52.84	11.22	16.67	35.88
	Attributes	27.59	25.31	50.82	11.19	16.88	36.29
	Siblings	28.15	25.56	51.57	11.35	16.85	36.46
	Docs	37.31	30.37	56.44	**23.49**	**23.20**	45.96
	Hybrid	**41.00**	**32.37**	**60.42**	23.16	23.04	**46.08**
Return	None	35.29	28.30	54.98	16.13	17.66	38.94
	Path	41.84	32.37	61.54	17.34	18.56	40.95
	Attributes	39.40	30.88	59.01	17.02	18.07	40.19
	Siblings	41.11	31.81	60.38	16.63	17.84	39.83
	Docs	47.06	34.82	64.55	**28.65**	**24.94**	**51.14**
	Hybrid	**49.51**	**36.41**	**66.93**	28.11	24.64	50.43
Parameter	None	30.20	25.44	52.08	19.76	19.21	42.14
	Path	36.88	29.05	58.07	19.69	19.12	42.06
	Attributes	34.63	27.69	55.68	19.92	19.56	42.33
	Siblings	36.93	29.25	57.72	19.55	19.09	41.80
	Docs	44.49	32.76	62.04	**33.09**	**26.55**	**53.18**
	Hybrid	**47.10**	**34.47**	**64.81**	33.02	26.47	53.02

by removing other contexts from the input instructions. For simplicity, we directly train different lightweight models on each type of descriptions, i.e., skip the multi-task instruction tuning process, and omit their context encoders and attention modules for the unused contexts. The results of LLM-based approach and lightweight approach are shown in Table IV and V respectively, where *None* denotes the baseline of not using any program context, *Docs* denotes using documentation context only, and *Hybrid* denotes using all types of contexts mentioned in Section IV-B.

In general, the scores on return and parameter descriptions are much higher than on function descriptions. This is because most of the first two descriptions are noun phrases and thus are shorter than the latter. Under the mixed-project splitting, the context-aware models perform significantly better than the baseline models on all descriptions no matter what type of context is used. This illustrates the importance of program contexts for code summarization. Among the contexts other than *Hybrid*, *Docs* is the most effective which boosts the performance of both lightweight model and LLM by more than 10 points of BLEU across all descriptions. Since the documentation context has the same modality as the model output, it provides demonstration to the model and can directly guide summary generation. It is also interesting that as the simplest context, *Path* is quite useful, which contributes even more than the class context. Since a package contains a set of programs with similar functions, businesses, or code components, the summary styles within a package might be similar, but those between different packages vary greatly. Thus, the package or class names can be important prior knowledge, e.g., to prompt the model to generate summaries of specific styles learned from training data. *Hybrid* shows the best performance on all descriptions and metrics ($p < 0.001$), indicating the necessity to mine different contexts for code summarization as well as the rationality of the chosen contexts.

The scores of all models drastically reduced under the cross-project setting. Since different projects can have totally different code patterns and vocabularies (recall that the near-duplicate codes across projects have been removed), it is much harder for the models to generalize to new projects. For LLMs, code-based context positively affects the performance with the most obvious improvement achieved by *Path*, and the

TABLE VI: Effectiveness of instruction tuning on StructCodeSum (LLM)

Description	Context	Training	Mixed-Project			Cross-Project		
			BLEU	METEOR	ROUGE-L	BLEU-S	METEOR	ROUGE-L
Function	None	Zero-Shot	12.75	20.66	40.02	**14.33**	**21.92**	**41.52**
		Instruction	**20.73**	**24.15**	**45.81**	13.36	20.58	40.83
	Hybrid	Zero-Shot	14.95	22.40	42.40	13.58	21.36	41.44
		Instruction	**42.29**	**35.01**	**63.80**	**24.65**	**24.94**	**46.87**
Return	None	Zero-Shot	16.85	22.37	42.88	16.95	**23.14**	43.32
		Instruction	**31.58**	**28.51**	**51.01**	**16.99**	22.16	**43.45**
	Hybrid	Zero-Shot	18.90	22.57	44.59	18.94	24.02	45.59
		Instruction	**50.26**	**38.57**	**68.57**	**29.32**	**26.81**	**50.30**
Parameter	None	Zero-Shot	21.49	23.47	46.40	23.20	**24.82**	**48.65**
		Instruction	**30.63**	**27.56**	**55.16**	**23.88**	22.08	47.83
	Hybrid	Zero-Shot	22.53	23.70	47.49	24.33	25.12	49.67
		Instruction	**49.71**	**37.59**	**67.44**	**36.98**	**29.01**	**55.11**

TABLE VII: Effectiveness of instruction tuning on StructCodeSum (Lightweight)

Description	Context	Training	Mixed-Project			Cross-Project		
			BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
Function	None	Standard	22.50	22.84	46.97	11.21	16.95	36.53
		Instruction	18.62	19.25	40.01	9.98	14.79	33.16
		Instruction (Ft)	**23.53**	**23.51**	**47.92**	**11.33**	**17.01**	**36.73**
	Hybrid	Standard	41.00	32.37	60.42	23.16	23.04	46.08
		Instruction	32.41	26.65	50.55	20.31	20.85	41.86
		Instruction (Ft)	**41.84**	**32.87**	**61.01**	**23.88**	**23.69**	**46.90**
Return	None	Standard	35.29	28.30	54.98	16.13	17.66	38.94
		Instruction	31.92	26.82	52.38	16.64	18.20	40.20
		Instruction (Ft)	**36.21**	**28.94**	**56.35**	**17.64**	**18.61**	**41.24**
	Hybrid	Standard	49.51	36.41	66.93	28.11	24.64	50.43
		Instruction	44.19	33.67	61.37	27.45	24.74	49.12
		Instruction (Ft)	**51.32**	**37.42**	**68.39**	**30.75**	**26.41**	**53.14**
Parameter	None	Standard	30.20	25.44	52.08	19.76	19.21	42.14
		Instruction	27.83	24.32	50.79	20.62	19.88	43.42
		Instruction (Ft)	**31.09**	**26.04**	**53.08**	**20.82**	**20.10**	**43.56**
	Hybrid	Standard	47.10	34.47	64.81	33.02	26.47	53.02
		Instruction	47.25	34.42	64.89	34.39	27.40	54.10
		Instruction (Ft)	**48.54**	**35.25**	**65.97**	**35.01**	**27.69**	**54.68**

performance of *Hybrid* keeps being the best ($p < 0.001$). However, for lightweight models, the comparison results are not consistent with those in the mixed-project setting. Specifically, code-based context can barely boost performance except for return descriptions, and *Hybrid* shows slightly lower performance than *Docs* despite still being much better than the other variants ($p < 0.001$). We attribute this to the highly varied context patterns in different projects. Since the lightweight model has not been pre-trained on massive projects, it may have greater difficulty in using unseen class context under the cross-project setting than the LLM.

B. RQ2: Effectiveness of Instruction Tuning

For the LLM-based approach, we compare it with the zero-shot inference of Code Llama to answer RQ2, as shown in Table VI. We feed the built instructions to Code Llama without training to get its zero-shot results. As seen, the instruction-tuned version of Code Llama is significantly better than the zero-shot one on all metrics when taking hybrid code context ($p < 0.001$). Moreover, it can be found that the zero-shot version is also not comparable with the lightweight model. This suggests that the LLM initially fails to well understand our instructions, but this can be fixed via instruction tuning on sufficient examples.

For the lightweight approach, we consider two baselines to answer RQ2. As shown in Table VII, *Standard* denotes conventionally training models on each type of descriptions (same as *Hybrid* in Table V), *Instruction* denotes multi-task instruction tuning without subsequently obtaining three expert models, and *Instruction (Ft)* denotes the training process described in Section IV-C1. As seen, in most cases, *Instruction* performs worse than *Standard*. This suggests that although a shared lightweight model can perform different tasks simultaneously, it is not as capable as the lightweight models optimized for individual tasks. By contrast, *Instruction (Ft)* shows the best scores on all descriptions and metrics ($p < 0.001$), verifying that it is a better approach to leverage shared knowledge between tasks without losing performance on individual tasks.

C. RQ3: Comparison with State-of-The-Art Models

We compare our approach with 7 representative code summarization models on generating function descriptions, including structure-aware models, context-aware models, retrieval-based models, and fine-tuned CodeBERT. The details of these approaches can be found in Appendix D. In addition, we adopt another zero-shot strategy for Code Llama than the one in Section VI-B as a baseline, named Code Llama (Prompt). Here the zero-shot inference is implemented by prompting Code

TABLE VIII: Performance of our approach compared with baselines on generating function descriptions

Model	Mixed-Project				Cross-Project			
	BLEU	METEOR	ROUGE-L	SIDE	BLEU	METEOR	ROUGE-L	SIDE
CodeNN	18.86	19.92	40.85	74.09	8.55	13.81	30.22	75.05
AST-AttendGRU	18.43	20.83	41.91	81.38	8.48	14.66	30.77	83.25
AST-AttendGRU + FC	20.86	22.03	43.49	80.09	9.74	16.10	33.67	82.53
AST-AttendGRU + PC	17.83	20.12	41.14	81.08	9.78	16.21	33.75	81.93
Rencos	23.57	23.73	48.13	81.92	11.56	17.37	37.25	83.42
Re^{2}Com	29.10	26.08	50.86	79.39	11.00	16.18	34.64	82.08
CodeBERT	27.84	25.82	52.25	72.40	12.73	18.03	38.67	82.57
Code Llama (Prompt)	21.83	23.16	42.40	83.22	23.20	23.76	45.40	84.83
StructCodeSum (Lightweight)	41.84	32.87	61.01	83.20	23.88	23.69	**46.90**	84.74
StructCodeSum (LLM)	**42.29**	**35.01**	**63.80**	**83.92**	**24.65**	**24.94**	46.87	**85.07**

Llama to generate full doc comments rather than different descriptions separately, also given the hybrid code context. The prompt template is slightly modified from the one in Fig. A-1, as shown in Fig. A-3. We use the heuristics described in Section III-B to extract function descriptions from its outputs for evaluation. We do not consider this baseline in Section VI-B because it may fail to include the parameter and (or) return descriptions in its output documentation.

The comparison results are shown in Table VIII. As seen, StructCodeSum (LLM) outperforms all baselines by a large margin on all metrics ($p < 0.001$). Its improvements over Code Llama (Prompt) again verify the efficacy of instruction tuning. On the other hand, StructCodeSum (Lightweight) greatly surpasses all baselines under the mixed-project setting on BLEU, METEOR, and ROUGE-L, and is comparable with the strongest baseline, namely Code Llama (Prompt) under the cross-project setting. Since it contains only 44M parameters, it is quite efficient against LLMs.

Among the baselines, AST-AttendGRU + PC shows only marginal improvements against AST-AttendGRU + FC under the cross-project setting, and fails to outperform the latter and even its base model, namely AST-AttendGRU on most metrics under the mixed-project setting. One possible reason could be its context selection strategy: randomly choose 10 files in the project. According to Table II, the average number of classes in a project in our dataset is 36.2, so random file selection will lead to the uncertainty of the resulting context, i.e., the relation and relevance between the context and the target method will be unknown. On the contrary, the hybrid context we use is based on program dependencies and similarities and is thus more reasonable.

Actually, the context used by AST-AttendGRU + FC is close to our *Siblings* variant in Table V, while its performance is not comparable with ours. The key difference here is their input forms: AST-AttendGRU + FC encodes the sibling methods separately and obtains their method-level vectors for decoding, while we concatenate all sibling signatures for encoding and obtain their token vectors for decoding. Previous work [39] suggests that the code summarization task prefers token-level attention rather than using higher-level attention only, which explains why our encoding method makes more sense. On the other hand, truncating a method to a short and fixed length (25 tokens according to [25], while the average length of our target method is 73.05 tokens recalling Table A-I) could also introduce noise such as partial method body or signature.

TABLE IX: The results of human evaluation

Approach	Naturalness	Informativeness
Human-Written	4.78	4.20
Code Llama (Prompt)	4.55	3.38
StructCodeSum (LLM)	4.61	3.72

It is also worth comparing our *Docs* variant in Table V to the retrieval-based neural models, i.e., Rencos and Re^{2}Com, in that the former can also be viewed as a retrieval-based code summarization model. Under the mixed-project setting, Re^{2}Com achieves the best BLEU and METEOR scores among the baselines, while our *Docs* variant scores much higher. The main difference between Re^{2}Com and our *Docs* variant is that the former retrieves the similar method and exemplar from the training data while we find existing documentation in the current project. Thus, the doc comments found from the code contexts can be more useful than those searched from an external codebase. Under the cross-project setting, the retrieval-based baselines do not show strong competitiveness. This is not surprising because if the retrieved summary is not relevant to the target code, the retrieval-based models may not gain ideal improvements from their non-retrieval-based counterparts [46]. This again indicates that the documented codes in the current project are valuable context.

D. RQ4: Human Evaluation

To answer RQ4, we conduct a human evaluation on the structured code summaries generated by StructCodeSum (LLM), the full method documentation generated by Code Llama (Prompt), as well as the human-written doc comments in the dataset. Following previous studies [15, 32, 46, 47], the code summaries are rated from two unrelated aspects: Informativeness and Naturalness. Informativeness indicates the proportion of important content in the codes that the summary covers, while Naturalness refers to the grammatical accuracy and fluency of the summary. Their scores are both integers and range from 1 to 5. We first use a calculator[5] to compute the minimum example number for evaluation to meet the desired statistical constraints. The confidence level and the margin of error are set at 5%, and the resulting minimum sample size is 382 based on the total size of our cross-project test set. Therefore, we randomly sample 400 methods from the cross-

[5]https://www.calculator.net/sample-size-calculator.html

TABLE X: Definition of token sharing metrics

Definition	**Explanation**
$I_x = \frac{\lvert \text{set}(\boldsymbol{y}) \cap \text{set}(\boldsymbol{x}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the target method.
$I_{path} = \frac{\lvert \text{set}(\boldsymbol{y}) \cap \text{set}(\boldsymbol{q}^{path}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the path context.
$I_{attr} = \frac{\lvert \text{set}(\boldsymbol{y}) \cap \text{set}(\boldsymbol{q}^{attr}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the class attributes.
$I_{sib} = \frac{\lvert \text{set}(\boldsymbol{y}) \cap \text{set}(\boldsymbol{q}^{sib}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the sibling method signatures.
$I_d = \frac{\lvert \text{set}(\boldsymbol{y}) \cap \text{set}(\boldsymbol{d}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the documentation context.
$I_h = \frac{\lvert \text{set}(\boldsymbol{y}) \cap (\text{set}(\boldsymbol{q}) \cup \text{set}(\boldsymbol{d})) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the hybrid context.
$D_{path} = \frac{\lvert (\text{set}(\boldsymbol{y}) - \text{set}(\boldsymbol{x})) \cap \text{set}(\boldsymbol{q}^{path}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the path context but do not appear in the target method.
$D_{attr} = \frac{\lvert (\text{set}(\boldsymbol{y}) - \text{set}(\boldsymbol{x})) \cap \text{set}(\boldsymbol{q}^{attr}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the class attributes but do not appear in the target method.
$D_{sib} = \frac{\lvert (\text{set}(\boldsymbol{y}) - \text{set}(\boldsymbol{x})) \cap \text{set}(\boldsymbol{q}^{sib}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the sibling method signatures but do not appear in the target method.
$D_d = \frac{\lvert (\text{set}(\boldsymbol{y}) - \text{set}(\boldsymbol{x})) \cap \text{set}(\boldsymbol{d}) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the documentation context but do not appear in the target method.
$D_h = \frac{\lvert (\text{set}(\boldsymbol{y}) - \text{set}(\boldsymbol{x})) \cap (\text{set}(\boldsymbol{q}) \cup \text{set}(\boldsymbol{d})) \rvert}{\lvert \text{set}(\boldsymbol{y}) \rvert}$	% Summary tokens that appear in the hybrid context but do not appear in the target method.

project test set, obtain the summaries generated by different approaches, and then equally divide the results into 8 groups.

24 participants with excellent English levels are invited to rate the summaries. They are either developers or post-graduate students of computer science, with 2-5 years of Java development experience and at least 5 years of total programming experience. Specifically, each group is evaluated by 3 participants in the form of a web questionnaire. Fig. A-4 shows an example from the questionnaire. The model names are hidden during evaluation, and the participants are asked to rate the given summaries based on the corresponding codes. For each summary, we use the average score from the 3 raters.

The results of human evaluation are shown in Table IX. As seen, there is a certain gap between the scores of the human-written comments and the full score of 5, indicating that there exist low-quality summaries in the dataset, which is a potential threat to validity. StructCodeSum (LLM) achieves higher scores than Code Llama (Prompt), especially on Informativeness. When applying the Wilcoxon Sign-Rank test to their Informativeness scores, we get $p < 0.05$, denoting significant improvement. The results verify that the structured code summaries generated by StructCodeSum have higher fluency and accuracy, thus being more useful to the developers. We provide output examples from different approaches in Appendix E.

VII. Discussion

A. Statistical Analysis on Summary Token Distribution

To understand the impact of program contexts on code summarization, we examine the content sharing between code summaries and different types of code contexts. We define several metrics to reflect the distribution of the summary (sub)tokens in different contexts, as shown in Table X, where *set* is a function that converts a sequence of (sub)tokens into a set, $\boldsymbol{x}$ is the target method, $\boldsymbol{q}$ is the code-based context, $\boldsymbol{d}$ is the documentation context, $\boldsymbol{y}$ is a code description, and % denotes "The proportion of". For simplicity, the parameter descriptions of a target method are concatenated to a single $\boldsymbol{y}$ when computing these metrics. In addition to the average results on these metrics, we are also interested in the summary token distributions on different ranges of I_x, i.e., different coincidence degrees between the target method and summary. The breakdown results are shown in Table XI, which are reported in percentage.

In the table, the first row of each type of description provides an overview of the token distribution. As seen, the average I_x of the descriptions is around 36%, indicating that most of the tokens in the summary do not appear in the corresponding method (out-of-method summary tokens, named OoM tokens). Among the basic contexts, the documentation context and sibling context share the most tokens with the summary, and the proportions, namely I_d and I_{sib}, are even larger than I_x. By comparison, our hybrid code context is able to cover many more keywords in the summary, with the proportion I_h of up to 70%. It also covers the most OoM tokens, with the proportion D_h of around 40%. This explains why our hybrid context makes sense in the above experiments.

The average results on different I_x ranges show that when I_x goes down, the number of sharing tokens between the summary and code contexts decreases as well. By contrast, the number of OoM tokens caught by the contexts increases as I_x declines. When I_x is below 25%, D_h exceeds 50%, which means half of the tokens in the summary can only be found in the code contexts. This is intuitive: if we cannot infer the function of a method, then we have to dig into its context. Overall, the results in Table XI further demonstrate the crucial role of code contexts in code summarization.

B. Computational Costs

In practice, a noteworthy aspect is the computational costs of different approaches. In terms of model complexity, StructCodeSum (Lightweight) contains only 44M parameters whereas StructCodeSum (LLM) encompasses 7B parameters. This leads to a significant difference in their memory costs and inference speeds. We show their time costs of generating function descriptions on the cross-project test set in Table XII. As seen, the inference speed of StructCodeSum (LLM) is 286 times slower than that of StructCodeSum (Lightweight) despite that the former has better performance (Table VIII). This suggests that they are suitable for different scenarios.

TABLE XI: Distribution of summary tokens in the code contexts with respect to different I_x ranges (in percentage)

Description	I_x **Range**	**Count**	I_x	$I_{path}(D_{path})$	$I_{attr}(D_{attr})$	$I_{sib}(D_{sib})$	$I_d(D_d)$	$I_h(D_h)$
Function	All	464143	36.15	8.87 (2.55)	15.21 (2.18)	39.53 (7.94)	53.90 (32.77)	**70.72 (38.11)**
	[0,25)	143478	13.32	6.25 (4.03)	8.85 (3.53)	23.55 (11.75)	52.53 (44.25)	**63.05 (50.48)**
	[25,50)	191172	34.33	8.89 (2.61)	14.69 (2.10)	38.14 (8.17)	53.61 (33.22)	**69.57 (37.48)**
	[50,75)	99091	56.61	11.46 (1.03)	20.52 (0.99)	54.19 (4.18)	56.15 (23.27)	**78.71 (25.42)**
	[75,100]	30402	88.65	12.70 (0.18)	31.17 (0.17)	75.91 (0.82)	54.82 (6.65)	**88.13 (13.43)**
Return	All	334300	35.43	9.85 (2.35)	14.41 (1.97)	38.41 (8.85)	55.12 (34.04)	**70.76 (39.69)**
	[0,25)	119947	11.27	4.63 (3.32)	6.61 (2.80)	22.08 (13.99)	51.33 (44.37)	**60.61 (51.22)**
	[25,50)	112469	33.81	9.19 (2.90)	13.31 (2.29)	36.92 (8.77)	54.72 (34.68)	**70.03 (39.20)**
	[50,75)	76164	58.62	13.68 (0.78)	21.83 (0.80)	53.34 (3.60)	59.97 (25.04)	**80.84 (26.89)**
	[75,100]	25720	86.48	25.73 (0.11)	33.59 (0.15)	76.87 (0.71)	60.16 (9.68)	**91.48 (16.78)**
Parameter	All	258313	38.52	8.10 (1.45)	14.64 (2.03)	42.85 (8.36)	61.22 (36.69)	**77.06 (41.07)**
	[0,25)	64791	12.99	5.29 (2.70)	9.16 (3.56)	24.29 (12.56)	61.53 (52.28)	**69.86 (57.41)**
	[25,50)	104606	33.87	7.40 (1.51)	13.04 (2.15)	39.12 (9.05)	59.84 (37.84)	**73.96 (41.86)**
	[50,75)	72345	57.66	10.44 (0.55)	20.18 (0.88)	57.54 (5.20)	62.65 (26.80)	**84.55 (29.35)**
	[75,100]	16571	84.15	13.26 (0.12)	21.92 (0.25)	74.79 (1.36)	62.59 (11.65)	**92.05 (16.83)**

TABLE XII: Computational costs of StructCodeSum (LLM) and StructCodeSum (Lightweight)

	Parameters	**Tokens / Second**	**Methods / Second**	**Test Time (Minutes)**
StructCodeSum (Lightweight)	44M	2469	238.76	2.35
StructCodeSum (LLM)	7B	6.75	0.87	672.15

If hardware resource is not a problem and higher summary quality is required, the former is preferred. However, if hardware resource is limited or the number of methods awaiting documenting is large, the latter is preferred.

Literally, the context of a method can be all source code files in the project where it resides. Unfortunately, a project may be very large and thus encoding the whole context is not feasible due to time or resource constraints. Also, the project size can easily exceed the context windows of LLMs. For example, the Apache Hadoop project[6] contains more than 31M tokens. Therefore, the hybrid code context we choose is highly efficient for boosting model performance. Another advantage of our simplified context is that it does not require full implementations of the components associated with the target method, e.g., only signatures of the sibling methods are needed, which is useful for projects under development.

VIII. Conclusion

In this work, we first release a code summarization dataset with various code contexts and summary content annotations. Then, we propose a deep learning method, StructCodeSum, for generating structured code summaries from hybrid code context. Given a target method, StructCodeSum predicts its function description, return description, parameter description, and usage description through hybrid code context, and ultimately builds a Javadoc-style code summary. The hybrid code context consists of path context, class context, documentation context, and call context of the target method. StructCodeSum provides both an LLM-based approach and a lightweight approach, which are suitable for different scenarios. Both of them significantly outperform representative baselines. Extensive experimental results on the constructed dataset indicate that:

1) The proposed hybrid code context significantly improves the model performance on generating different code descriptions. Statistical analysis shows that the hybrid context covers more than 70% of the tokens in the code summary while the target code itself only covers about 36%.
2) When generating function descriptions, StructCodeSum significantly outperforms the state-of-the-art code summarization approaches including structure-aware models, context-aware models, retrieval-based models, and pre-trained language models.
3) The quality of the structured code summaries generated by our approach is better than the documentation generated by Code Llama.

However, there are limitations to this work. The heuristics for categorizing the content within doc comments are based on massive observations and are impossible to cover all patterns in the documentation. Therefore, they cannot guarantee the resulting annotations are correct. On the other hand, the documentation styles of different programming languages can vary, e.g., the specification of Python docstring is different than Javadoc. It is non-trivial to better identify different elements within code documentation so that code summarization models can be effectively guided to generate more useable code summaries. Moreover, since our dataset is collected from GitHub, the documented methods in our test set may have been exposed to CodeBERT and Code Llama during their pre-training stages, which is a threat to validity.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (62372174, 62276097), the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality (22010504100), the Computational Biology Program of Shanghai Science and Technology Commission (23JS1400600), the Research Project Funding of Shanghai

[6]https://github.com/apache/hadoop

Data Exchange Corporation, and Shanghai Engineering Research Center of Smart Energy.

References

[1] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer-an investigation of how developers spend their time," in *2015 IEEE 23rd international conference on program comprehension*, pp. 25–35, IEEE, 2015.

[2] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[3] G. Garousi, V. Garousi-Yusifoğlu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and usefulness of technical software documentation: An industrial case study," *Information and software technology*, vol. 57, pp. 664–682, 2015.

[4] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 826–831, IEEE, 2018.

[5] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: Code annotation for code retrieval with reinforcement learning," *arXiv preprint arXiv:1904.00720*, 2019.

[6] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," *arXiv preprint arXiv:2002.10198*, 2020.

[7] A. T. Nguyen and T. N. Nguyen, "Automatic categorization with deep neural network for open-source java projects," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 164–166, IEEE, 2017.

[8] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 795–806, IEEE, 2019.

[9] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and S. Y. Philip, "Improving automatic source code summarization via deep reinforcement learning," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 397–407, IEEE, 2018.

[10] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, pp. 2179–2217, 2020.

[11] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 2269–2275, 2018.

[12] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[13] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, pp. 184–195, 2020.

[14] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[15] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 349–360, 2020.

[16] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, S. Y. Philip, and G. Xu, "Reinforcement-learning-guided source code summarization using hierarchical attention," *IEEE Transactions on software Engineering*, vol. 48, no. 1, pp. 102–119, 2020.

[17] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. Lyu, "Code structure–guided transformer for source code summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–32, 2023.

[18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[19] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

[20] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[21] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

[22] 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.

[23] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5, 2022.

[24] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[25] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *IEEE/ACM International Conference on Mining Software Repositories*, Association for Computing Machinery, Inc, 2020.

[26] B. Liu, T. Wang, X. Zhang, Q. Fan, G. Yin, and J. Deng, "A neural-network based code summarization approach by using source code and its call dependencies," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pp. 1–10, 2019.

[27] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 253–264, IEEE, 2021.

[28] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[29] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 3931–3937, 2019.

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[31] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[32] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *54th Annual Meeting of the Association for Computational Linguistics 2016*, pp. 2073–2083, Association for Computational Linguistics, 2016.

[33] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, pp. 2091–2100, PMLR, 2016.

[34] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, pp. 200–210, 2018.

[35] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[37] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.

[38] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," *arXiv preprint arXiv:2012.14710*, 2020.

[39] Z. Zhou, H. Yu, G. Fan, Z. Huang, and X. Yang, "Summarizing source code with hierarchical code representation," *Information and Software Technology*, vol. 143, p. 106761, 2022.

[40] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 184–195, IEEE, 2021.

[41] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," *arXiv preprint arXiv:2108.12987*, 2021.

[42] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[43] R. Xie, W. Ye, J. Sun, and S. Zhang, "Exploiting method names to improve code summarization: A deliberation multi-task learning approach," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 138–148, IEEE, 2021.

[44] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1385–1397, 2020.

[45] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 155–166, IEEE, 2021.

[46] Z. Zhou, H. Yu, G. Fan, Z. Huang, and K. Yang, "Towards retrieval-based neural code summarization: A meta-learning approach," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 3008–3031, 2023.

[47] A. Aghamohammadi, M. Izadi, and A. Heydarnoori, "Generating summaries for methods of event-driven programs: An android case study," *Journal of Systems and Software*, vol. 170, p. 110800, 2020.

[48] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[49] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[50] Y. Wang, W. Wang, S. Joty, and 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*, pp. 8696–8708, 2021.

[51] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (Online), pp. 2655–2668, Association for Computational

Linguistics, June 2021.

[52] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[53] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, “Are we building on the rock? on the importance of data preprocessing for code summarization,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 107–119, 2022.

[54] M. Li, H. Yu, G. Fan, Z. Zhou, and J. Huang, “Classsum: a deep learning model for class-level code summarization,” *Neural Computing and Applications*, vol. 35, no. 4, pp. 3373–3393, 2023.

[55] A. V. M. Barone and R. Sennrich, “A parallel corpus of python functions and documentation strings for automated code documentation and code generation,” *arXiv preprint arXiv:1707.02275*, 2017.

[56] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.

[57] M. L. Collard, M. J. Decker, and J. I. Maletic, “srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration,” in *2013 IEEE International conference on software maintenance*, pp. 516–519, IEEE, 2013.

[58] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

[59] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Satd detector: A text-mining-based self-admitted technical debt detection tool,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 9–12, 2018.

[60] K. Shimonaka, S. Sumi, Y. Higo, and S. Kusumoto, “Identifying auto-generated code by using machine learning techniques,” in *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pp. 18–23, IEEE, 2016.

[61] M. Olsson, *Java Quick Syntax Reference*. Berkeley, CA: Apress, 2018.

[62] H. Yamada and A. Hazeyama, “A support system for helping to understand a project in software maintenance using the program package name,” in *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pp. 411–416, IEEE, 2013.

[63] G. Salton, A. Wong, and C.-S. Yang, “A vector space model for automatic indexing,” *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[64] E. J. Hu, yelong shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “LoRA: Low-rank adaptation of large language models,” in *International Conference on Learning Representations*, 2022.

[65] M.-T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” *arXiv preprint arXiv:1508.04025*, 2015.

[66] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[67] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[68] K. Papineni, S. Roukos, T. Ward, and 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*, pp. 311–318, 2002.

[69] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.

[70] L. Chin-Yew, “Rouge: A package for automatic evaluation of summaries,” in *Proceedings of the Workshop on Text Summarization Branches Out, 2004*, 2004.

[71] M. Denkowski and A. Lavie, “Meteor universal: Language specific translation evaluation for any target language,” in *Proceedings of the ninth workshop on statistical machine translation*, pp. 376–380, 2014.

[72] D. Roy, S. Fakhoury, and 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*, pp. 1105–1116, 2021.

[73] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic similarity metrics for evaluating source code summarization,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 36–47, 2022.

[74] A. Mastropaolo, M. Ciniselli, M. Di Penta, and G. Bavota, “Evaluating code summarization techniques: A new metric and an empirical characterization,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE ’24, (New York, NY, USA), Association for Computing Machinery, 2024.

[75] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, and B. Xu, “Mat: A simple yet strong baseline for identifying self-admitted technical debt,” *arXiv preprint arXiv:1910.13238*, 2019.

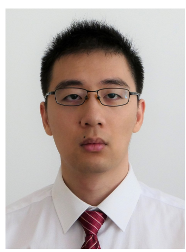
[76] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, pp. 1–10, 2020.

[77] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, “Magicoder: Source code is all you need,” *arxiv preprint arxiv:2312.02120*, 2023.

[78] B. Chen and C. Cherry, “A systematic comparison of

smoothing techniques for sentence-level bleu," in *Proceedings of the ninth workshop on statistical machine translation*, pp. 362–367, 2014.

[79] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 6000–6010, 2017.



Ziyi Zhou received his Ph.D. degree in computer science from East China University of Science and Technology in 2023. He is currently a postdoctoral with the Institute of Natural Sciences, Shanghai Jiao Tong University. His research interests include program comprehension, natural language processing, and deep learning for software engineering.

Mingchen Li is currently working toward the Ph.D. degree with the School of Information Science and Engineering, East China University of Science and Technology. He received the B.E. degree from the Department of Mathematics, East China University of Science and Technology in 2020. His research interests include software engineering, code summarization and artificial intelligence.

Huiqun Yu received the B.S. degree from Nanjing University in 1989, M.S. degree from East China University of Science and Technology (ECUST) in 1992, and Ph.D. degree from Shanghai Jiao Tong University in 1995, all in computer science. He is currently a Professor of computer science with the Department of Computer Science and Engineering at ECUST. From 2001 to 2004, he was a Visiting Researcher in the School of Computer Science at Florida International University. His research interests include big data, software engineering, and trustworthy computing. He is a member of the ACM, a senior member of the IEEE, and a senior member of the China Computer Federation.

Guisheng Fan is currently an Associate Professor with the Department of Computer Science and Engineering, where he teaches requirement engineering, software testing, and cloud computing. His research interests include intelligent software, formal methods for complex software system, and techniques for analysis of software architecture.

Penghui Yang received the master's degree from East China University of Science and Technology in 2024. She is currently an engineer in the field of software engineering, responsible for quality assurance. Her main research interests include automatic code summarization and program comprehension.

Zijie Huang received his B.S. and M.E. degrees from Shanghai Normal University in 2016 and 2020, and the Ph.D. degree from the East China University of Science and Technology in 2024, all in computer science. He is currently a postdoctoral researcher with the Shanghai Development Center of Computer Software Technology. His research interests include open source software quality assurance and empirical software engineering.

APPENDIX

A. Categorize Untagged Sentences

Split block into sentences. A block is split into a sequence of sentences by finding periods. Considering that the documentation may contain source codes, we determine a "." as a period if:

- It is immediately followed by a whitespace character or an HTML tag.
- It is not within a pair of brackets.
- It is not within abbreviations "e.g." and "i.e.".

Identify example codes and SATD descriptions. The first things we have to remove from a raw doc comment are code examples and SATD descriptions, in that generating them is currently beyond the scope of code summarization and they will badly affect model evaluation. In detail, we remove a sentence from a doc comment if it contains any of the following patterns (ignoring letter case, same for the other rules below):

- It contains the keyword "see".
- It starts with the keywords "for instance", "for example", or "example:".
- It defines explicit code style via `<code>...</code>` or {`@code`}, and the wrapped code is longer than 30 characters.
- It contains `<pre>,<blockquote>` tags.
- It contains the SATD keywords "TODO", "FIXME", "bug", "compatibility", "hack", or "xxx" [75, 76].

It is hard to say that the untagged block is an overall method description because it may be some specific comments missing tags. In practice, comments for `@return` and `@exception` are likely to be mixed up with the method description rather than following explicit tags. Since tag comments are preferred to be non-redundant with the overall method description, it is reasonable to tell them apart.

Identify return descriptions. We determine an untagged sentence as a return comment if it contains any of these patterns:

- It starts with "return".
- It contains the keywords "this returns", "this method returns", "this function returns", "will return", "will not return", "won't return", "will be returned", "will not be returned" or "won't be returned".

Identify exception descriptions. Similarly, we determine an untagged sentence as an exception comment if it contains any of these patterns:

- It starts with "throw".
- It contains the keywords "this throws", "this method throws", "this function throws", "will throw", "will not throw", "won't throw", "will be thrown", "will not be thrown" or "won't be thrown".

Identify usage descriptions. Since usage comments or tips usually reflect detailed business and implementation logic and are very hard to predict from the contexts, e.g., "Must be called from the UI thread", we also separate them from the general method descriptions. We determine an untagged sentence as a usage comment if it contains any of these patterns:

- It starts with "used", "called" or "invoked".
- It contains the keywords "be used", "be called", "be invoked", "works only", "only works", "cannot", "do not", "is valid only", "is only valid" or "please".

Identify summary sentence. If the first sentence in the untagged block is not removed or identified as a specific comment, it will be considered as the summary sentence.

B. Instruction Templates

1) Pre-trained LLM: Figure A-1 illustrates the instruction template for Code Llama, which is designed according to the OOS-Instruct dataset [77]. It is divided into two sections: **System Prompt** and **User Prompt**. Overall, **System Prompt** aims

TABLE A-I: Statistics of code lengths, context sizes, and description lengths after preprocessing

Name	**Mean**	**Std.**	**Median**	**75th Percentile**	**85th Percentile**	**95th Percentile**
# Tokens in the target method	73.05	88.38	34	87	139	286
# Attributes in a class	4.99	5.79	3	6	9	17
# Methods in a class	12.26	11.33	9	16	22	39
# Callees in the target method	1.02	2.30	0	1	2	4
# Callers of the target method	1.09	8.13	0	1	2	3
# Words in a function description	9.89	5.61	8	12	15	21
# Words in a parameter description	7.30	4.55	6	9	11	16
# Words in a return description	8.70	5.18	7	12	14	18
# Words in an exception description	8.65	4.83	7	10	13	18
# Words in a usage description	15.62	6.99	14	20	24	30

TABLE A-II: Statistics of the dataset split by different strategies

Splitting Strategies		# Target Methods	# Function Descriptions	# Parameter Descriptions	# Return Descriptions
Mixed-project	Training	496,180	371,212	310,610	267,530
	Validation	62,022	46,307	39,312	33,351
	Testing	62,023	46,624	39,030	33,419
Cross-project	Training	496,467	371,305	306,813	271,831
	Validation	62,068	46,483	41,547	31,883
	Testing	61,690	46,355	40,592	30,586

```
System Prompt (Fixed)
You are a helpful, precise, detailed, and concise artificial intelligence assistant with deep expertise in Java programming languages. You excel at understanding and summarizing code effectively. In this task, you are required to read the provided Java method and its context. Based on the given requirements, generate a specific type of summary. This summary can be a description of the method's functionality, a description of the return value, or a description of a parameter.
Requirements:
The generated summary should accurately reflect the object to be described.
The generated summary should be concise and easy to understand.
If context is provided, you need to understand the code based on the context and generate a description.
Below is a method and its context. Your task is to provide the summary based on its target and format.

User Prompt
### Context:
#### Sibling methods signatures:
<sibling method signature - 1>
<sibling method signature - 2>
…
#### Class Attributes:
<attribute - 1>
<attribute - 2>
…
#### Path:
Project name : <project name>
Package name : <package name>
Class name : <class name>
Class Signature : <class signature>
#### Relevant documentation (Relevance level : {low, medium, high, very high})
<Relevant document>
### Method:
<code>
### Summary target:
{Functionality description, Return value description, Parameter description of <parameter name>}
### Response (one sentence):

Output
<summary>
```

Fig. A-1: Instruction template for Code Llama

to help Code Llama understand the task and **User Prompt** is the core part of an instruction. Besides, **Output** is the summary to be generated, i.e., the ground truth description.

In **User Prompt**, the subsection **Context** consists of the hybrid code context introduced in Section IV-B, where the red placeholders are replaced by specific contents. In particular, for the documentation context, the similarity calculated by Eq. 2 is mapped to words indicating the relevance level, where the mapping thresholds are set to 0.33, 0.48, and 0.63 (this is statistically derived from the training data). The placeholder <code> is replaced by the target method. **Summary target** indicates which kind of description should be generated. The marker "Response:" indicates the start of generation. The placeholder <summary> is replaced by the target description during training and omitted during inference. Code Llama has a built-in tokenizer, so the input codes are not tokenized before being fed to the model.

2) Lightweight Model: For the lightweight model, the input codes are tokenized and further split into subtokens according to camelCase and snake_case. In particular, given a package name, we tokenize it into a token sequence by ".", and then append the tokenized class name to it. The tokenized code-based contexts are concatenated to a single subtoken sequence together with different tags indicating their types. Formally, the subtoken sequences of path $\boldsymbol{q}^{path}$, attributes $\boldsymbol{q}^{attr}$ and siblings $\boldsymbol{q}^{sib}$ form a input sequence $\boldsymbol{q} = (\texttt{<path>}, \boldsymbol{q}^{path}, \texttt{<attr>}, \boldsymbol{q}^{attr}, \texttt{<sib>}, \boldsymbol{q}^{sib})$, where the attributes and signatures within $\boldsymbol{q}^{attr}$ and $\boldsymbol{q}^{sib}$ are separated by ";".

Special tags are also inserted into the tokenized documentation context to separate different descriptions. Assuming that the retrieved method has a function description $\boldsymbol{d}^{fun}$, a sequence of parameter descriptions $\boldsymbol{d}^{par}$, a return description $\boldsymbol{d}^{ret}$, a sequence of exception descriptions $\boldsymbol{d}^{exc}$ and a usage description $\boldsymbol{d}^{use}$, the concatenated input will be $\boldsymbol{d} =$

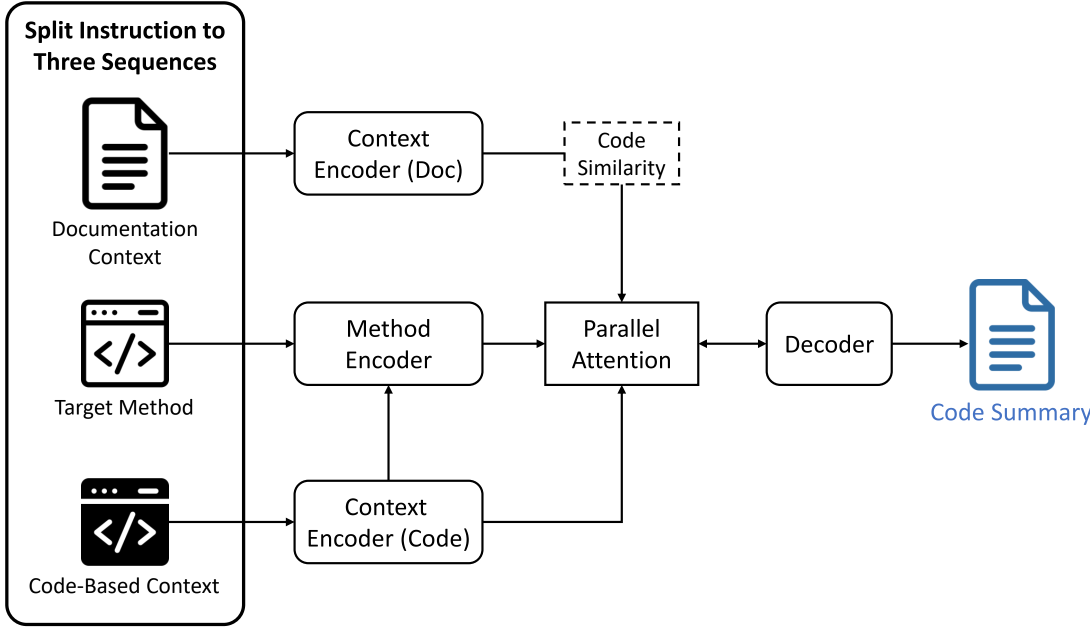


Fig. A-2: Model architecture of StructCodeSum (Lightweight)

$(\texttt{<fun>}, \boldsymbol{d}^{fun}, \texttt{<par>}, \boldsymbol{d}^{par}, \texttt{<ret>}, \boldsymbol{d}^{ret}, \texttt{<exc>}, \boldsymbol{d}^{exc}, \texttt{<use>}, \boldsymbol{d}^{use})$, where the descriptions within $\boldsymbol{d}^{par}$ and $\boldsymbol{d}^{exc}$ are split by ";". In practice, documented methods sometimes do not exist in a project. In this situation, $\boldsymbol{d}$ will be a zero-padded sequence, and the input similarity will be set to 0.

We insert a task tag (<fun>, <par>, or <ret>) at the beginning of each target method to prompt the model to generate different descriptions. To make the model generate the desired parameter descriptions, we insert the tokenized parameter name before the target method. Namely, when generating a parameter description, the input of the method encoder will be $\boldsymbol{x}' = (\texttt{<par>}, \boldsymbol{x}^{par}, \texttt{<par>}, \boldsymbol{x})$, where $\boldsymbol{x}^{par}$ is the subtoken sequence of the target parameter.

C. Automatic Metrics

BLEU [68]: It calculates the weighted geometric mean of n-gram precisions between the generated sentence and the reference for different n, and applies brevity penalty on short predictions:

$$\begin{aligned} BLEU &= BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \\ BP &= \begin{cases} 1 & c > r \\ \exp\left(1 - \frac{r}{c}\right) & c \leq r \end{cases} \end{aligned} \tag{11}$$

where p_n is the n-gram matching precision of length n subsequences, c is the length of candidate sentence, r is the reference length and BP refers to brevity penalty. We compute BLEU for $N = 4$ with w_n set to uniform weight $\frac{1}{4}$. Considering that higher order n-grams may not overlap, we smooth the BLEU score with NIST smoothing (smoothing method 3 in [78]).

ROUGE-L [70]: It is a common metric for text summarization. It computes the similarity between a generated summary Y and a reference X based on the length of the longest common subsequence. Suppose the lengths of X and Y are r and c respectively, it is computed as:

$$\begin{aligned} P_{lcs} &= \frac{LCS(X,Y)}{c} \\ R_{lcs} &= \frac{LCS(X,Y)}{r} \\ F_{lcs} &= \frac{(1+\beta^2) \cdot P_{lcs} \cdot R_{lcs}}{R_{lcs} + \beta^2 \cdot P_{lcs}} \end{aligned} \tag{12}$$

where LCS calculates the length of the longest common subsequence of two sentences. In our experiments, F_{lcs} is reported and β is set to 1.

System Prompt (Fixed)

You are a helpful, precise, detailed, and concise artificial intelligence assistant with deep expertise in Java programming languages. You excel at understanding and summarizing code effectively. In this task, you are required to read the provided Java method and its context. Based on the given requirements, generate a JavaDoc for the method.
Requirements:
The generated JavaDoc should accurately reflect the object to be described.
The generated JavaDoc should be concise and easy to understand.
If context is provided, you need to understand the code based on the context and generate a description.
Below is a method and its context. Your task is to generate the JavaDoc of the method.
Context:

User Prompt

Sibling methods signatures:
<sibling method signature - 1>
<sibling method signature - 2>
…
Class Attributes:
<attribute - 1>
<attribute - 2>
…
Path:
Project name : <project name>
Package name : <package name>
Class name : <class name>
Class Signature : <class signature>
Relevant documentation (Relevance level : {low, medium, high, very high})
<Relevant document>
Method:
<code>
Summary target:
JavaDoc
Response (JavaDoc format):

Fig. A-3: Prompt template for the baseline Code Llama

METEOR [71]: Based on BLEU, it takes recall into account and applies synonym matching. It computes a parameterized harmonic mean of unigram precision P and recall R with a penalty:

$$\begin{aligned} METEOR &= (1 - Pen) \cdot F_{mean} \\ F_{mean} &= \frac{PR}{\alpha P + (1-\alpha)R} \\ Pen &= \gamma \cdot \left(\frac{ch}{m}\right)^{\beta} \end{aligned} \tag{13}$$

where m is the number of unigram matches and ch is the number of chunks. We use the optimized parameter settings for the English target described in [71]: α = 0.85, β = 0.2 and γ = 0.6.

SIDE [74]: Different from the above metrics, SIDE measures the suitability of the generated summary for the target method independently from the original human-written comment. It leverages contrastive learning to train a BERT-like model such that the paired codes and summaries are close in its learned embedding space while the unpaired ones are set apart. Subsequently, the cosine similarity between the embeddings of the generated summary and the target method is used to measure the summary quality. Since SIDE treats the first sentence of the Javadoc as the summary during training, it can only be adopted to evaluate the function description.

All these metrics are computed on the summary level. The implementations of BLEU and METEOR are from NLTK 3.5[7] and the implementation of ROUGE-L is from py-rouge[8].

D. Baselines

CodeNN [32] is the first neural code summarization model. It uses LSTM with attention to generate summary directly from token embeddings of source code.

AST-AttendGRU [8] is a typical model with multiple inputs, which has one GRU encoder for the token sequence of code and another one for its flattened AST, namely SBT sequence [34]. During decoding, the attentional contexts of both encoders are fused to predict a target word. Similar combination strategies are adopted in [9, 11, 13].

[7]https://www.nltk.org/
[8]https://pypi.org/project/py-rouge/

AST-AttendGRU + FC [25] enhances AST-AttendGRU with file context. The file context mentioned in [25] refers to other methods in the same file as the method to be summarized. In the case of Java files, this is usually equivalent to the sibling methods within the same class as the target method. In detail, based on AST-AttendGRU, it encodes these methods using an extra GRU encoder, and makes the decoder attend to the file context during summary generation. For efficiency, the authors truncate each method in the context to the first 25 words.

AST-AttendGRU + PC [27] improves AST-AttendGRU with project-level context, with an extra project-level encoder to create a vectorized representation of randomly selected code files in a project. The project-level encoder first encodes the methods in the selected files (the maximum file number is set to 10 according to the paper) to method vectors via an RNN, and then uses another RNN to aggregate the method vectors to file vectors. During decoding, the decoder attends to the file vectors to generate target words.

Rencos [44] is the first work that combines retrieval-based and NMT-based methods in code summarization. Given a target code snippet, it retrieves the two most similar code snippets from the training set based on syntax and semantics respectively, and then feeds them into a trained code summarizer. The code summarizer translates the three codes simultaneously and combines their output probabilities according to their similarities.

Re^{2}Com [15] is another retrieval-based model proposed recently. Compared with Rencos, it further considers the existing comment of a similar code snippet and uses it as an exemplar. It builds an encoder-decoder model with four biLSTM encoders for the given code snippet, its SBT sequence, its similar code, and the exemplar, respectively. The context vectors of the target code and exemplar are fused according to the code similarity.

CodeBERT [48] is a pre-trained model for programming language and natural language based on Transformer [79] architecture. It is pre-trained on CodeSearchNet with both bimodal data, which refers to parallel data of code-documentation pairs, and unimodal data, which stands for unpaired codes and natural language texts. Its training objective includes masked language modeling and replace token detection. Code summarization can be one of its downstream tasks.

Since previous neural models mainly focus on generating a brief function description of code rather than a structured code summary, we limit the comparison to function descriptions. In detail, these baselines are trained and evaluated on the examples that have function descriptions in our dataset, with the same splitting and preprocessing as ours. The hyperparameters of these baselines are kept the same as those presented in their papers. For a fair comparison, we enhance CodeNN with parametric multiplicative attention and input feeding mechanism [65], which is also adopted by our approach. We also replace the GRUs in AST-AttendGRU and its variants with LSTMs based on their original implementations, and their encoders are set to be bidirectional instead of unidirectional. These modifications greatly improve their performances. All these baselines adopt beam search during decoding, with the same settings as ours. For the pre-trained CodeBERT, we equip it with a Transformer decoder and then fine-tune it on our training data.

E. Output Examples

In this section, we provide several output examples from different approaches on the test set to examine their actual quality, as shown in Table A-III and Table A-IV, where Code Llama (No Context) is similar to Code Llama (Prompt) in Section VI-C but without any code context in the prompts. In each example, we show the human-written doc comment as well as part of the code context, and the formatting of the code summaries is adjusted for a better look.

The comment in Example 1 contains an OoM keyword "transport". This token appears in the package name and class name of the target method rather than itself. Since CodeBERT and Code Llama (No Context) do not use code context, they fail to generate this token. On the contrary, StructCodeSum generates an accurate summary. In Example 2, there is an OoM keyword "observation sequence" in the comment, where CodeBERT and Code Llama (No Context) also fail to generate it. Notably, the method in this example has a parameter and return statement, but there is no corresponding description in the human-written comment. Since "observation sequence" is covered in the documentation context, StructCodeSum correctly captures it. Meanwhile, the usage description, parameter description, and return description generated by StructCodeSum are all consistent with the target method and can serve as a supplement to the original comment. In Example 3, the most important caller of the target method is `convertFieldValue`. StructCodeSum accurately predicts its summary as a usage description of the target method. Moreover, in both Example 3 and 4, StructCodeSum generates better parameter descriptions than Code Llama (No Context).

From these examples, we can see that compared with CodeBERT and vanilla Code Llama, StructCodeSum can generate different types of code descriptions with better quality through hybrid code context, and further produce standardized and structured code summaries.

Human Evaluation

Target Method (#4526174)

```
public static byte[] randomBytes(int size) {
    byte[] bytes = new byte[size];

    for (int index = 0; index < size; index++) {
        bytes[index] = (byte) randomNumber(0, 255);
    }

    return bytes;
}
```

Generated JavaDoc

```
/**
* create an array of random bytes.
* @param size the number of random bytes to create.
* @return an array of random bytes.
*/
```

Naturalness Score (Naturalness refers to the grammaticality and fluency of the summary.)

- ○ 1 - The result is totally not readable.
- ○ 2 - The result is hard to read but part of its meaning can be guessed.
- ○ 3 - The result contains several grammatical errors or repeated phrases but they do not affect understanding
- ○ 4 - The result contains only one repeated phrase or minor grammatical error and is easy to read.
- ◉ 5 - There is no grammatical error or repeated phrase in the result and it is as smooth as from human.

Informativeness Score (Informativeness refers to the amount of key content carried over from the input code to the generated comments, ignoring fluency of the text.)

- ○ 1 - The result cannot reflect any functionality of the code snippet and is totally misleading.
- ○ 2 - The result shares some information with the code but the rest part is inconsistent with the actual functionality of the code.
- ○ 3 - The result covers the very basic functionality of the code (such as actions) but loses specific information and seems indifferent.
- ○ 4 - The result provides a specific description of the code.
- ◉ 5 - The result is able to summarize the code precisely.

Submit Evaluation

Fig. A-4: An example from the questionnaire for human evaluation

TABLE A-III: Examples of generated summaries

<table>
<tr><th colspan="3">Example 1</th></tr>
<tr><td>Source Code</td><td colspan="2">@Override
public void shutdown() {
runner.set(false);
try {
threadA.join();
if (threadB != null)
threadB.join();
} catch (Exception e) {}
CloseHelper.quietClose(driver);
try {
Thread.sleep(500);
} catch (Exception e) {}
}</td></tr>
<tr><td rowspan="2">Context</td><td>Package Name</td><td>org.nd4j.parameterserver.distributed.transport</td></tr>
<tr><td>Class Name</td><td>BaseTransport</td></tr>
<tr><td rowspan="4">Summary</td><td>Human-Written</td><td>This method stops transport system.</td></tr>
<tr><td>CodeBERT</td><td>Shutdown the client.</td></tr>
<tr><td>Code Llama (No Context)</td><td>Shutdown the thread.</td></tr>
<tr><td>StructCodeSum (LLM)</td><td>Shutdown the transport.</td></tr>
<tr><th colspan="3">Example 2</th></tr>
<tr><td>Source Code</td><td colspan="2">private int[] translate(T[] o) {
return Arrays.stream(o).mapToInt(ordinal).toArray();
}</td></tr>
<tr><td rowspan="2">Context</td><td>Similar Code</td><td>public double p(T[] o)</td></tr>
<tr><td>Documentation Context</td><td>Returns the probability of an observation sequence.
@param o an observation sequence.
@return the probability of this sequence.</td></tr>
<tr><td rowspan="4">Summary</td><td>Human-Written</td><td>Translates an observation sequence to internal representation.</td></tr>
<tr><td>CodeBERT</td><td>Translate an array.</td></tr>
<tr><td>Code Llama (No Context)</td><td>Translates the given array of objects to an array of integers.
@param o the array of objects to be translated.
@return the translated array of integers.</td></tr>
<tr><td>StructCodeSum (LLM)</td><td>Translate the observation sequence to an integer sequence.
This method can be used in {@code p}: computes the p value for a given vector.
@param o an observation sequence.
@return the integer representation of the observation sequence.</td></tr>
</table>

TABLE A-IV: Examples of generated summaries (continued)

<table>
<tr><th colspan="3">Example 3</th></tr>
<tr><td>Source Code</td><td colspan="2">protected P upConvertPayload(GenericRecord inputRecord) throws
DataConversionException {
try {
    Schema payloadSchema = getPayloadSchema(inputRecord);
    latestPayloadReader.setSchema(payloadSchema);
    byte[] payloadBytes = getPayloadBytes(inputRecord);
    Decoder decoder = DecoderFactory.get().binaryDecoder(payloadBytes, null);
    return latestPayloadReader.read(null, decoder);
    } catch (Exception e) {
        throw new DataConversionException(e);}}
}</td></tr>
<tr><td>Context</td><td>Call Context</td><td>protected Object convertFieldValue(Schema outputSchema, Field field, GenericRecord inputRecord, WorkUnitState workUnit)</td></tr>
<tr><td rowspan="4">Summary</td><td>Human-Written</td><td>Convert the payload in the input record to a deserialized object with the latest schema.
@param inputRecord the input record.
@return the schema'ed payload object.</td></tr>
<tr><td>CodeBERT</td><td>Convert the payload.</td></tr>
<tr><td>Code Llama (No Context)</td><td>Converts the payload of the input record to the latest schema.
@param inputRecord the input record.
@return the converted payload.</td></tr>
<tr><td>StructCodeSum (Lightweight)</td><td>Deserialize payload using payload schema.
This method can be used in {@code convertFieldValue}: convert a field value to a java object.
@param inputRecord the input record from which to deserialize payload.
@return the converted payload.</td></tr>
<tr><th colspan="3">Example 4</th></tr>
<tr><td>Source Code</td><td colspan="2">public boolean hasStickyHeader(View itemView, int orientation, int position) {
    int offset, margin;
    mDimensionCalculator.initMargins(mTempRect1, itemView);
    if (orientation == LinearLayout.VERTICAL) {
        offset = itemView.getTop();
        margin = mTempRect1.top; } else {
        offset = itemView.getLeft();
        margin = mTempRect1.left;
    }
    return offset <= margin && mAdapter.getHeaderId(position) >= 0;
}</td></tr>
<tr><td rowspan="4">Summary</td><td>Human-Written</td><td>Determines if a view should have a sticky header.
@param itemView given by the RecyclerView.
@param orientation of the Recyclerview.
@param position of the list item in question.
@return True if the view should have a sticky header.</td></tr>
<tr><td>CodeBERT</td><td>Checks if a sticky header exists in the adapter.</td></tr>
<tr><td>Code Llama (No Context)</td><td>Checks if the item at the given position has a sticky header.
@param itemView the item view.
@param orientation the orientation of the list.
@param position the position of the item.
@return true if the item has a sticky header, false otherwise.</td></tr>
<tr><td>StructCodeSum (LLM)</td><td>Determines if an item is obscured by a sticky header
@param itemView the view to check for a sticky header.
@param orientation linear layout, vertical or horizontal.
@param position the position of the item view.
@return true if the item view has a sticky header.</td></tr>
</table>