Function Call Graph Context Encoding for Neural Source Code Summarization

Aakash Bansal, Zachary Eberhart, Zachary Karas, Yu Huang, and Collin McMillan

Abstract—Source code summarization is the task of writing natural language descriptions of source code. The primary use of these descriptions is in documentation for programmers. Automatic generation of these descriptions is a high value research target due to the time cost to programmers of writing these descriptions themselves. In recent years, a confluence of software engineering and artificial intelligence research has made inroads into automatic source code summarization through applications of neural models of that source code. However, an Achilles' heel to a vast majority of approaches is that they tend to rely solely on the context provided by the source code being summarized. But empirical studies in program comprehension are quite clear that the information needed to describe code much more often resides in the context in the form of Function Call Graph surrounding that code. In this paper, we present a technique for encoding this call graph context for neural models of code summarization. We implement our approach as a supplement to existing approaches, and show statistically significant improvement over existing approaches. In a human study with 20 programmers, we show that programmers perceive generated summaries to generally be as accurate, readable, and concise as human-written summaries.

Index Terms—automatic documentation generation, source code summarization, neural networks, context-aware models.

1 Introduction

A summary of source code is a short description of that code in natural language. Even very brief summaries e.g., "creates connection to game server" help programmers comprehend source code without having to read the code itself. These summaries form the backbone of documentation for programmers, such as the navigable HTML files generated by JavaDocs and Doxygen [1]. The task of automatically writing this part of documentation has become known as source code summarization [2], and has been a holy grail of software engineering research for decades [3], [4].

The workhorse of almost all recent research into code summarization is the attentional encoder-decoder neural architecture. The inspiration for using models of this architecture derives from machine translation in NLP, in which sentences in one natural language (e.g., French) are translated into another (e.g., English). When provided sufficient training data samples (usually well into the millions), the encoder portion of the model learns a representation of one language, and the decoder learns the other. The representations are combined via an attention network or other mechanism. Then if the encoder is provided a sentence in one language, the decoder can be used to help predict an output sentence in the other language. This is a tidy solution for machine translation because the information needed to write a sentence in one language tends to exist in translated sentences in other languages – the encoder usually has access to all the information it needs to represent the sentence for the decoder.

At a high level, almost all recent approaches to code summarization are essentially encoder-decoder neural models in which the input to the encoder is the source code and the output from the decoder is the natural language description. The encoder must learn a representation of the code suitable for the decoder to write a description. The typical direction for research is to create ever more complex models of the input source code via the encoder, with the aim to learn better representations for predicting a code summary via the decoder.

But applications of the metaphor of machine translation only extend so far for code summarization. Empirical studies in program comprehension are quite clear that not all of the information necessary to understand a section of source code exists within that source code itself [5], [6], [7], [8], [9], [10]. The implication for code summarization research is that there is a ceiling at which even a "perfect" encoder model could not lead to an accurate summary, because the information needed to write that summary is not in the piece of code being summarized.

One potential answer to this problem is also evident in program comprehension literature: the Function Call Graph. The nodes in this graph are the subroutines in a program. The edges are call relationships among the subroutines (usually directed from one function to another). Existing empirical studies have shown that most of the information that human programmers need to understand a function appears within two "hops" in this graph – e.g., a function's caller and the caller's callers [11]. This information forms the *context* that a human needs to understand the code. A hope for neural approaches to code summarization is to provide the encoder with this same information, so that it can learn to understand software more like a human would.

- *Manuscript received —- —- —-. This work is supported in part by the NSF CCF-2100035. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.*
- *The authors are with the Department of Computer Science and Engineering, University of Notre Dame, IN 46556.*
  E-mail: abansal1,zeberhar, and cmc @nd.edu
  Department of Computer Science, University of Vanderbilt, TN 37235.
  E-mail: z.karas,yu.huang @vanderbilt.edu
- *This paper has supplementary downloadable multimedia material available at https://github.com/projcon/callcon2021 provided by the authors.*

Recently, Bansal *et al.* [12] achieved significant improvements using methods from other files in the project. These methods were randomly selected and modeled as "project context". This paper is an extension of that project context paper published at ICPC 2021, except that now we eliminate the random factor. In addition, we clearly define the relationship between the query method and every method in the context. We observe that randomly selected files and methods in project context could improve or hinder results based on the selection. Therefore, we chose a call graph to define a fixed set of methods. See Table 1 and Section 2.1.

In this paper, we present an approach for encoding the function call graph context for neural approaches to source code summarization. Our approach is an augmentation to, rather than a competitor to, existing techniques. Essentially the approach we take is to 1) extract all functions within two hops of the given subroutine in the call graph, 2) create vectorized representations of these functions using a recurrent neural network, 3) use a graph neural network to propagate information among these representations, and 4) use an attention mechanism to highlight the most important functions in the call graph context. The result is a context vector of the call graph that can be appended to the code vector created by existing code summarization approaches.

We implement our approach and augment a baseline neural model for code summarization. We perform an experiment on the dataset drawn from large software repositories. We show marked improvement over the baselines in almost all cases. More importantly, we observe that this improvement is orthogonal to the improvements made by more complex representations of the subroutines being summarized themselves. While better representations of the code being summarized are helpful, our approach is helpful in a different way. We release a complete package necessary for replication in our online appendix (see Section 10).

2 Background & Related Work

This section discusses key background technologies and related work, such as source code summarization and neural encoder-decoder model designs.

2.1 Source Code Summarization

The term "source code summarization" was coined around 2009 by Haiduc *et al.* [37] for the task of generating short descriptions of source code. The word "summarization" referred to the underlying technologies borrowed from the Natural Language Processing research community used to summarize natural language documents. At the time, these were dominated by keyword extraction techniques, such as ranking the top-n words in a document using tf/idf or a similar metric. A widely-accepted practice was to use the *context* around source code to help this process [38], where context was defined as a set of functions in the Function Call Graph surrounding the code being described [39].

This line of research was largely put on ice around 2017, with the introduction of neural models of source code and encoder-decoder architectures (e.g., seq2seq, graph2seq) [3]. Figure 1 depicts this history. Column I in the figure groups techniques based on IR, manual feature design, and other heuristics. Column N refers to papers in which the underlying model is based on a neural architecture. Column G means the code is represented via graph or graph-like features such as the AST. Column T means the model is Transformer-based. Column C means the intellectual merit of the paper is in using the code context.

Figure 1 shows an important pattern, that while neural models have succeeded IR and template-based solutions, the use of *context* is ripe for a resurgence of research interest. Between 2017 and 2019, many papers achieved big gains from the big data input. Their efforts were focused on how to pre-process the data for use in existing neural models (an exemplar in this category is the SBT by Hu *et al.* [23] technique for linearizing an AST, which has been validated by third parties [30]). Since then, two complementary strategies have emerged to best improve performance of code summarization: 1) better models of the code itself, such as by Zügner *et al.* [35] and Liu *et al.* [36], and 2) models that include context information, such as by Haque *et al.* [34].

More recently, retrieval-based techniques have also been employed to use contextual information. In 2020, Wei *et al.* [40] introduce Re2Com, a technique to find similar functions in a database and use corresponding summaries as a secondary input to neural network. In 2021, Li *et al.* [41] introduced a technique to retrieve summaries of similar functions and used them as a template. They proposed a module to edit these summaries with new information from the target function. These approaches are important, given the wide re-use of source code in online repository. However, our dataset and use-case is different, in that we remove duplicate methods and doc-strings from our training set to prevent data leaks. Our approach does not rely on the availability of documented code and summary inputs to the model during prediction. We believe retrieval-based techniques have different application conditions from ours, and thus, do not serve as baselines.

	I	N	G	T	C
McBurney (2016) [13]	x				x
Zhang *et al.* (2016) [14]	x				x
Iyer *et al.* (2016) [15]		x			
Rodeghero *et al.* (2017) [16]	x				x
Fowkes *et al.* (2017) [17]	x				
Badihi *et al.* (2017) [18]	x				
Loyola *et al.* (2017) [19]		x			
Lu *et al.* (2017) [20]		x			
Jiang *et al.* (2017) [21]		x			
Hu *et al.* (2018) [22]		x			
Hu *et al.* (2018) [23]		x	x		
Allamanis *et al.* (2018) [24]		x	x		
Wan *et al.* (2018) [25]		x			
Liang *et al.* (2018) [26]		x			
Alon *et al.* (2019) [27], [28]		x	x		
Gao *et al.* (2019) [29]		x			
LeClair *et al.* (2019) [30]		x	x		
Nie *et al.* (2019) [31]		x			
Haldar *et al.* (2020) [32]		x			
Ahmad *et al.* (2020) [33]		x		x	
Haque *et al.* (2020) [34]		x			x
Zügner *et al.* (2021) [35]		x	x		
Liu *et al.* (2021) [36]		x	x		
Bansal *et al.* (2021) [12]		x			x
(This Paper)		x	x		x

Fig. 1. Snapshot of the past five years in source code summarization. Column I stands for IR-based techniques. N means neural network-based. G means the code is modeled as a graph. T means Transformer designs. C means learning chiefly from code context.

2.2 Project Context

In 2021, we [12] proposed an approach that improves source code summarization using contextual information from other files in the project. However, there was a random element in the selection of the contextual information. We selected n methods at random from other files in the project to model "project context". We view that work as a proof of concept that shows the potential for improvement using out-of-file context. After that project, we observed that this random factor leads to high variance in terms of metric score improvements. Table 1 shows how different random selections can impact the gains made using "project context". The v2 selection achieved much lower BLEU scores compared to random selections. Whereas, the v3 selection achieved a higher BLEU score than the selection published in that work (v1). To eliminate this random element, we posit the function call graph offers a logical solution. As part of the call graph, selected methods from the project have a clearly defined relationship with the method being summarized. There is a flow of data through function parameters and return values between nodes in the call graph. We design our approach to learn this relationship between methods in the context and the target function to be summarized. Note, this paper uses a different dataset than the one used in Table 1, explained in Section 4.

TABLE 1
BLEU for reproduction of the project context paper [12] with different random seed values for selection of methods to include in the context.

Projcon	BLEU				
Models	A	1	2	3	4
attendgru	15.87	36.22	18.89	11.55	8.03
projconv1	17.19	37.34	20.20	12.71	9.10
projconv2	16.36	36.15	19.29	12.07	8.52
projconv3	17.77	37.88	20.71	13.24	9.59

2.3 Encoder-Decoder Neural Models

The workhorse of almost all neural source code summarization approaches is the encoder-decoder model architecture. This architecture consists of two learned representations of paired inputs of data. The idea was initially proposed for use in machine translation, where an "encoder" would generate a vector representation of a sentence in e.g., French, while a "decoder" would generate a representation of the same sentence in e.g., English [42]. A key improvement to the original model design is the addition of "attention" around 2014 by Bahdanau *et al.* [43]. The purpose of attention is to connect features in the encoder representation to features in the decoder representation. Usually in machine translation, this means connecting a word in one language to another e.g., "ami" in French to "friend" in English.

The basic structure of the encoder-decoder model has found uses in many language generation tasks, such as image captioning [44], question answering [45], and code summarization (see above). While uses of the encoder-decoder architecture are far too common to be covered in one paper, notable surveys include: [3], [46], [47], [48]. This paper is in the same vein as this related work, except that we focus on encoding function call graph context rather than details about the source code being summarized itself. In this way, this paper may be viewed as bordering image captioning in addition to machine translation, as we seek to locate features in a context with a much broader scope than the text that is to be generated. In translation, the encoder sentence is usually expected to contain the features necessary to translate it. In image captioning, often artifacts such as surrounding text in a webpage are considered.

2.4 Function Call Graph Context

The Function Call Graph is a key abstraction of code context used in software engineering literature for decades. The graph itself consists of nodes, which are the functions (or methods, subroutines) in a program, and edges, which are the call relationships among the functions. It has long been observed that the behaviors of a program, from a human perspective, tend to be defined by these relationships [39], [49], [50]. To take a classic example, the behavior of booking a single passenger on a single flight in airline software is unlikely to be implemented by just one function – there is a constellation of functions in the call graph that would implement this feature [51]. Abstracting a program as functions and function calls is one of the key components of human programmers' mental models of program behavior [52], [53], and a mainstay of software engineering research.

In this paper, we define the **call context** of a subroutine as the nodes that fall within two edges from the subroutine in the program call graph. This scope includes the callers of a subroutine and that caller's callers. Plus, it includes the functions that a subroutine calls, plus the functions those functions call. Our definition of call context is in line with related work, which has repeatedly shown that human programmers almost always find the information they need within two edges in the function call graph [51], [54], [55], [56], [57]. This scope, while "only" encompassing two hops in the call graph, turns out to cover an average of 8% of a typical program in our subset. For example, in our dataset of 190k Java methods, projects have a median of 170 methods, and the mean call context of a method includes about 14 methods.

3 Approach

This section describes our approach. Essentially we extract the call context, and then use a neural model to learn a representation of this context to predict summaries.

3.1 Modeling Call Context

The first step in our approach is to model the call context. We define call context in Section 2.4 based on related work. However, in practice, hardware and software limitations mean not all information from all functions in this context can be included. We extract the call context with a limiting hyperparameter b (breadth) to indicate the maximum number of calls per function. If a function has more than b calls, we include only the first b calls that function makes. The value of b is a delicate balance between maximizing the number of functions in the context, while preventing a single function from "taking over" the context by making too many calls. The maximum value of b we are able to test is 5, limited by the largest graph we can fit on the GPU memory available to us.

Consider the example call context in Figure 2. The function `setRadius()` is the target, and is part of its own call

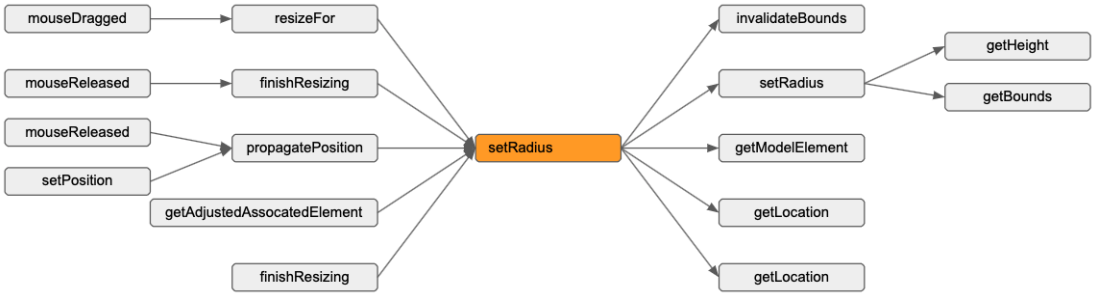

Fig. 2. A depiction of the call context for an example function named `setRadius()`. We define call context as the functions within two hops in the program function call graph (see Section 2.4).

context. To build the rest of the call context, we take the first b functions that `setRadius()` calls. Then, we take the first b calls that those functions make. These functions are the "right side" of the call context in Figure 2. Then to make the "left side", we add a maximum of b functions in which `setRadius()` is within the first b calls. Then we add the b functions that call each of those functions. The maximum number of functions in the call context is then $2*(b^2+b)+1$.

We chose b=5, implying a maximum of 61 functions in the call context. As we will note in Dataset Preparation (Section 4), the mean number of calls per function in the dataset is 2.7, and only around 25% made more than five calls. Only four functions in the dataset had the maximum of 61 functions in the call context.

3.2 Neural Model

The heart of our prediction model is a graph neural network (GNN) that creates a vectorized representation of the functions in the call context. We use a recurrent neural network (RNN) to create a vector representation for the initial state of each function in the call context. Then we use a GNN to propagate information among these functions based on their function calls. We combine this call context information with information from a standard encoder-decoder model to predict a summary for the function.

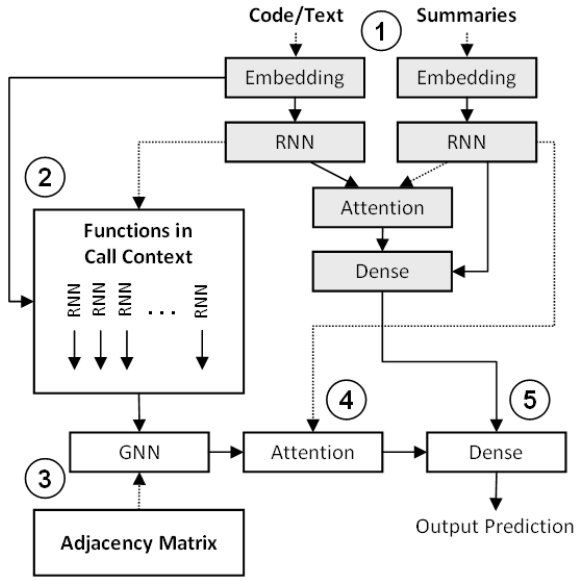

Fig. 3. The architecture of our approach. White areas indicate novel additions for this paper. Gray areas indicate components of the model loaded from a baseline, prior approach. Solid arrows indicate information flow over which back propagation is allowed. Dashed arrows indicate information flow without back propagation.

An overview of the neural model underpinning our approach is in Figure 3. In general, our model is based on an encoder-decoder architecture like most approaches to neural code summarization. What is novel is that we add components to the encoder to help the model learn from call graph context (see our definition of call context in Section 2.4). The gray components in Figure 3 (area 1) indicate a standard encoder-decoder model in which the encoder's input is the source code of the function and the decoder learns to represent the summaries. This encoder-decoder model is the foundation of almost all neural source code summarization techniques (see Section 2.1), and we continue to use it in our approach.

The white components in Figure 3 indicate novel contributions of this paper. The purpose of these components is to create a vectorized representation of the call context of a function. We combine this representation with the standard encoder-decoder model. This works as follows:

In **area 1**, we obtain the source code for a target function to summarize. That code is the input to the standard "gray" encoder. We represent this part as:

$$C' = G_2(E_2(C)) \tag{1}$$

$$T' = G_1(E_1(T)) \tag{2}$$

$$T'' = SoftmaxActivation(\sum_{i=1}^{m} C'_i T'_i) \tag{3}$$

$$T_c = \sum_{i=1}^{m} T''_i T'_i \tag{4}$$

$$T_a = T_c \oplus C' \tag{5}$$

Here, G_1 and G_2 denote pertained RNNs for the function (T) and comment (C) tokens respectively. E_1 and E_2 denote the word embeddings for the function and comment tokens respectively. The $\oplus$ symbol denotes a concatenation operation, i and j are iterative variable.

In **area 2** we encode the source code for every function in the call context of the target function. We use an RNN to create a representation of the source code for each function in this context. We use the same word embedding and vocabulary as the standard encoder, and the initial state of each RNN is the final state of the RNN from the standard encoder. This operation is represented as:

$$N = GRU(E_1(G_n)) \tag{6}$$

Technically, the call context becomes an mxn matrix where m is the number of functions in the context and n is the vector size of the RNN's representing each function.

In **area 3** we obtain the edges among each function in the call context and store these edges as an adjacency matrix. When we create the adjacency matrix, we treat the call context edges as undirected. Otherwise, a GNN would propagate information from the caller functions to the target, but not from the callee functions. E.g., functions on the left side of the target function in Figure 2 would propagate information to the target, but not functions on the right. In our approach, information from any functions can propagate to any other functions within the call context.

We use a convolutional GNN to propagate information among the functions in the call graph, based on the adjacency matrix. The GNN we use is of our own implementation. It is a faithful re-implementation of the GNN used by graph2seq [58] and also used successfully for modeling abstract syntax trees [59]. We represent a single "hop" of GNN propagation as:

$$N_{new} = ReLu(\sum_{k=1}^{m}(\sum_{x=1}^{m} E_{jx}N_{ix})_{jk}W_{ik}) \tag{7}$$

$$N = N_{new} \tag{8}$$

Here, N is the state of the context at the beginning and end of a "hop". E and W denote the edge adjacency matrix and a randomly initialized weight matrix respectively. The output of the GNN is a matrix with the same mxn shape as the call context. The content of this matrix is similar as well, except that the GNN propagates information among the nodes, so that nodes near each other in the graph become more similar to each other. In our view, this propagation is likely to create a good representation of code context because the edges represent actual information flow in the program.

In **area 4** we compute attention between the decoder and the post-GNN call context. The decoder represents words in the summary, while the entries in the call context represent functions in that context. Some words in the summary may have more relevance to some functions than others. For example, the word "record" in a summary may have high relevance to functions related to audio/video files. To capture this relevance, we compute attention between the words in the decoder and the functions in the call context. We represent this part as:

$$C'' = SoftmaxActivation(\sum_{i=0}^{m} C'_i N_i) \tag{9}$$

$$C_c = \sum_{i=0}^{c} mC''_i N_i \tag{10}$$

Here cm is the number of words in the summary, which is 13 for our experiment. Our attention mechanism is identical to the one described by Luong *et al.* [60] and used extensively in code summarization research [23], [30]. The difference is that we compute attention to functions in call context rather than only to words in the target function itself.

In **area 5**, the final step is to combine the prediction from the standard encoder-decoder model (the "gray" part) with the output prediction from the call context.

$$O = DenseReLu(C_c) \tag{11}$$

$$O = O \oplus T_a \tag{12}$$

$$Cn = DenseSoftmax(O) \tag{13}$$

Here, a dense layer is calculated after attention in both the standard encoder-decoder model and the call context. We concatenate the output from these dense layers into a single vector, and then send that vector to another dense layer which serves as the output layer.

3.3 Hyperparameters

Table 2 lists hyperparameters of our neural network. Due to the high expense of computation time in training large neural models, a grid search for optimal hyperparameters is not currently feasible. However, satisfactory values for many parameters are available in related literature: 1) we use the vocabulary sizes (v_d and v_e) from recommendations by LeClair *et al.* [61], 2) we use a GRU with a vector size of 100 as recommended for modeling functions by Haque *et al.* [34], and 3) as mentioned above, we use a convolutional graph neural network to propagate information among the functions in the call graph inspired by related work. We discussed the values of b and m above in Section 3.1 among other parameters, and we decided on $h = 1$, empirically through RQ4 in Section 6.4.

TABLE 2
Hyperparameters of our neural network

Parameter	Value	Description
v_d	10000	decoder vocab size
v_e	75000	encoder vocab size
b	5	maximum calls per function
m	61	maximum functions in call context
n	100	embedding vector size
cm	13	number of tokens in the summary
RNN	GRU	type of RNN
GNN	conv	type of GNN
h	1	hops in GNN

3.4 Input/Output Details

There are two key components of the input/output details: 1) preprocessing, and 2) training procedure. For preprocessing, note that the source code of a function and the summary of that function are both inputs to the model during training. We used the preprocessed summaries using techniques by LeClair *et al.* [61]. We truncated to 13 words, dropped to lower case, and removed punctuation. For source code, the paper preprocessed by removing non-word characters, splitting by camel case and underscore, and dropping to lower case. We used the same preprocessing, except that we did not remove non-word characters, and we replaced newlines with a `NL` special token. We found in pilot studies that these newlines and other tokens (brackets, periods, etc.) led to better predictions.

Our training procedure is teacher forcing [62]. Essentially what teacher forcing does is train the model to predict summaries one word at a time, while providing the answer at each step during training. A comprehensive discussion of teacher forcing is beyond the scope of this paper, as it is the most common means by which neural code summarization algorithms are trained [23], [30], [34], [63].

3.5 Hardware/Software Details

Our implementation and experimental hardware includes a Xeon E5-1650v4 CPU, two Quadro P5000 GPUs with 16GB of Video memory each, and 128GB of system memory.

Our software versions for reproduction include CUDA 11.2, Tensorflow 2.9, Python 3.10, Pandas 1.4, NLTK 3.6, Debian Experimental Release (March 2021 Snapshot).

4 Dataset Preparation

We curated our dataset from a larger one that was used in the project context paper we extend [12] and used to generate Table 1. To build our dataset, we extracted call graphs to create the call context and adjacency matrices required for our approach. However, we observed that a large percent of the functions in the dataset used for the project context paper are quite small and tend to involve rewriting the words available in a subroutine's signature. For example, a method `playMidiFile()` may have a summary like "plays a midi file." This observation is further corroborated by related work that uses the same dataset [64]. While these short methods are interesting targets for code summarization and automatic documentation generation in general, **we view call context as a way to help write summaries for longer sections of code that may make several function calls.** A function that is very short and makes no function calls will have a limited call context and is less likely to benefit from call context. Therefore, we prepared a subset of the published dataset.

Our dataset originates from the one published by LeClair *et al* [61]. We selected this dataset because it follows accepted practice in the field, such as splitting training/validation/test sets by project. Then, we selected the largest 10% of Java methods from the dataset, where we define "largest" by the number of tokens in each method. Our reasons for using this threshold are two fold. First, this threshold led to approximately 200k subroutines in the dataset, which is near the upper limit of our resources for extracting call graphs. We used `srcml` [65] to extract the call graph for every project in the dataset, and then subdivided these graphs into call context function and adjacency matrices. Due to high I/O requirements, parallelization of this process has limited benefits, and even 200k functions took approximately two weeks of compute time. The second reason we used this threshold is because it favors the larger subroutines, versus, for example, a random selection.

Due to filtering, the number of samples in the dataset decreases while the size of those samples increases. The average number of tokens increases from 27 in original set to 122 in our dataset. The number of methods in the call graph has a median of 12 and mean of 14.2. Roughly 32% of methods were called by more than 5 methods, as well as 24% called more than 5 methods. Due to resource constraints explained in 3.1 the graph is limited to a breadth of 5. In general, a vast majority of the methods chosen via the size threshold both call (93%) and are called (also 93%) by at least one other method.

5 Quantitative Experiment

This section describes our quantitative experiment involving computed metrics over our dataset. This experiment is distinct from our qualitative experiment in Section 7.

5.1 Research Questions

The research objective of this experiment is to measure the effect of call context on the prediction quality of neural code summarization, in a reproducible manner and over large datasets. We ask the following Research Questions (RQs):

- **RQ$_1$** What is the difference between our approach and recent baselines, as measured by automated metrics?
- **RQ$_2$** Are the gains orthogonal, as measured by ensembles of models to generate summaries?
- **RQ$_3$** How does our model compare to the baselines when the summary includes words from the call graph?
- **RQ$_4$** What is the effect of breadth b and hops h on performance, as measured by automated metrics ?

The rationale behind RQ$_1$ is two fold. First, automated metrics are inexpensive, so performance over several thousand subroutines may be computed. This evaluation of a large set reduces the risk of inadvertently "cherry picking" a set for which one model works better than another (as may happen in a human evaluation with only a few dozen randomly selected samples). Second, it makes studies reproducible – the datasets are available via our online appendix, and automated metrics are well-defined in the literature. So far, a vast majority of approaches use this type of evaluation.

The rationale behind RQ$_2$ is that automated metrics are measured as an average over the whole test set, but some models may excel over a subset more than the other. Recent work by Bansal *et al.* [12] and LeClair *et al.* [66] uses these to capture orthogonal gains and generate better summaries using ensembles.

The purpose of RQ$_3$ is to explore "how" the call graph helps generate better summaries. One possibility is that there are unique words in the methods that are part of the call graph that do not exist in the target method. The reference summaries, written by human programmers, could contain identifiers and other words from the caller and callee methods. We ask this RQ to measure performance of our approach over these niche subsets.

The rationale behind RQ$_4$ is to find out how our design choices affect the model performance. First, our graph layer is inspired by LeClair *et al.* [59]. They found that the value of h produces diminishing returns in terms of performance for AST graphs, but we do not know if this is true for call graphs. Second, we chose $b = 5$ because that is the maximum number we could fit on our GPU. While we cannot test out higher values of b, it may be that a smaller graph performs better. We ask this RQ to quantify the level to which these design choices affect performance of our approach.

5.2 Methodology

Our methodology is based on the accepted practice followed by most papers on neural source code summarization techniques. First, as detailed in Section 4, we prepare our dataset. The training, validation, test split is approximately 80%, 10%, 10%, though because these datasets were split using a "by project" procedure to reduce biases (and because we filter for larger functions), the split percentages are only approximate. The second step is to train each baseline (plus our approach) using the training set.

We trained for a maximum of 20 epochs, and then chose the model at the epoch that achieved the highest validation set accuracy. Then we used that model to predict summaries for subroutines in the test set. Finally, we computed METEOR [67], USE [68], ROUGE [69], and BLEU [70] scores for predictions against the reference summaries for those subroutines. Recently, Roy *et al.* [71] evaluated several metrics for source code summarization and recommended METEOR as an alternative to BLEU. Haque *et al.* [68] found that sentence encoder based metrics correlate better to human similarity ratings compared to n-gram based metrics such as METEOR and BLEU. They recommended a Universal Sentence Encoder [72] based metric we report as USE. We report BLEU-A and ROUGE-LCS scores to be consistent with literature and our previous work with project context that this paper extends. We use the python NLTK version 3.6 implementation of these metrics.

5.3 Baselines

Our experiment includes five baselines. We created faithful reimplementations of each baseline in our own experimental framework in order to reduce experimental variables. While many papers do release reproducibility packages, there are slight differences such as preprocessing/input changes, different vector sizes or RNN types (e.g., LSTM vs. GRU), pretrained word embeddings, etc. Therefore, output of the models could vary due to implementation differences, while we aim to measure the effect of call context only.

`code2seq` This approach represents a family of approaches that use paths in the AST to represent code, as introduced by Alon *et al.* [63]. This model is consistently a strong performer in experiments in various papers [34].

`ast-attendgru-fc` This approach is the "file context" baseline introduced by Haque *et al.* [34]. It works by modeling each subroutine in the same file with an RNN, then computing attention between the output summary words and the RNN output vectors for each of those subroutines. It is similar to this approach in that other subroutines are modeled for prediction, but it is different in that it does not consider any relationships between subroutines.

`codegnngru` This approach represents a family of papers that model source code as an abstract syntax tree (AST). This approach is the best configuration reported by LeClair *et al.* [59], which uses a GNN to encode the AST only, not any external context. GNNs are a growing area of investigation for modeling source code [24], [35], [73].

`transformer` This approach is essentially a vanilla transformer-based seq2seq model, as described by Ahmad *et al.* [33]. Transformer-based models have found strong acceptance in the NLP research community and are beginning to be tested for code summarization.

`HANcode` This approach is the newest baseline that uses a Hierarchical Attention Network designed for source code summarization. This approach is a non-ensembled version of the best performing approach proposed by Zhou *et al.* [74], who recommend an ensemble with an AST based approach for best performance.

5.4 Threats to Validity

This experiment carries threats to validity similar to most studies of neural code summarization. The key threats are the datasets and the automated metrics. Different conclusions may arise with different random splits, since these splits affect what is in the training set. We attempted to mitigate this threat by using a project based split filtered by length. The other key threat are the metrics BLEU and ROUGE. These metrics compute word overlap, which is only one way of measuring similarity. We attempt to mitigate this threat by also conducting a qualitative experiment with human experts.

6 Quantitative Study Results

This section includes our answers to RQ1, RQ2, and RQ3 based on the data collected in the quantitative experiment.

6.1 RQ_1: Performance using Automated Metrics

We found that compared to our baselines, `callcon` achieved the highest performance as measured by automated metrics. In Table 3 the top sub-table shows the METEOR, USE, ROUGE-LCS, and BLEU scores over our test set for each of the model configurations. For METEOR and USE scores we also performed a paired T-test for statistical significance where P-value of less than 0.05 indicates rejection of the null hypothesis and hence strongly suggests statistical significance. This test was conducted compared to our approach – therefore, the values are blank for our approach. We found that `callcon` achieved highest scores in all but one sub-metric, P score for ROUGE-LCS.

TABLE 3
Automated metric scores for baselines, our approach, and ensembles. Top subtable reports scores for standalone models where the T-tests for METEOR and USE compare our approach with each baseline. Bottom subtable reports scores for ensembles of our approach and baselines where the T-tests compare ensemble with the baseline.

Models	METEOR			USE			ROUGE-LCS			BLEU-A
	Score	T-test	P-val	Score	T-test	P-val	P	R	F1	Score
code2seq	30.44	18.39	0.01	46.68	28.31	0.01	50.42	42.91	44.75	16.52
ast-attendgru-fc	29.51	23.01	0.01	45.69	32.74	0.01	49.54	41.57	43.70	15.55
codegnngru	32.27	8.90	0.01	49.42	14.67	0.01	51.15	44.49	46.07	17.94
transformer	33.24	3.46	0.01	51.87	0.49	0.31	**51.81**	45.58	46.97	18.53
HANcode	27.19	34.54	0.01	41.21	51.90	0.01	43.80	38.80	39.85	14.71
callcon	**33.80**	-	-	**51.95**	-	-	51.41	**46.12**	**47.16**	**19.54**

Ensembles	METEOR			USE			ROUGE-LCS			BLEU-A
	Score	T-test	P-val	Score	T-test	P-val	P	R	F1	Score
code2seq+callcon	34.31	25.22	0.01	52.53	35.67	0.01	53.66	46.75	48.46	19.83
ast-attendgru-fc+callcon	34.33	30.45	0.01	52.53	40.73	0.01	53.47	46.58	48.30	19.77
codegnngru+callcon	34.77	17.88	0.01	53.04	24.82	0.01	**53.91**	47.00	48.68	20.15
transformer+callcon	**35.09**	14.31	0.01	**53.84**	15.70	0.01	53.83	**47.43**	**48.92**	**20.22**
HANcode+callcon	33.79	40.22	0.01	51.25	53.99	0.01	51.96	45.98	47.32	19.42

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3279774

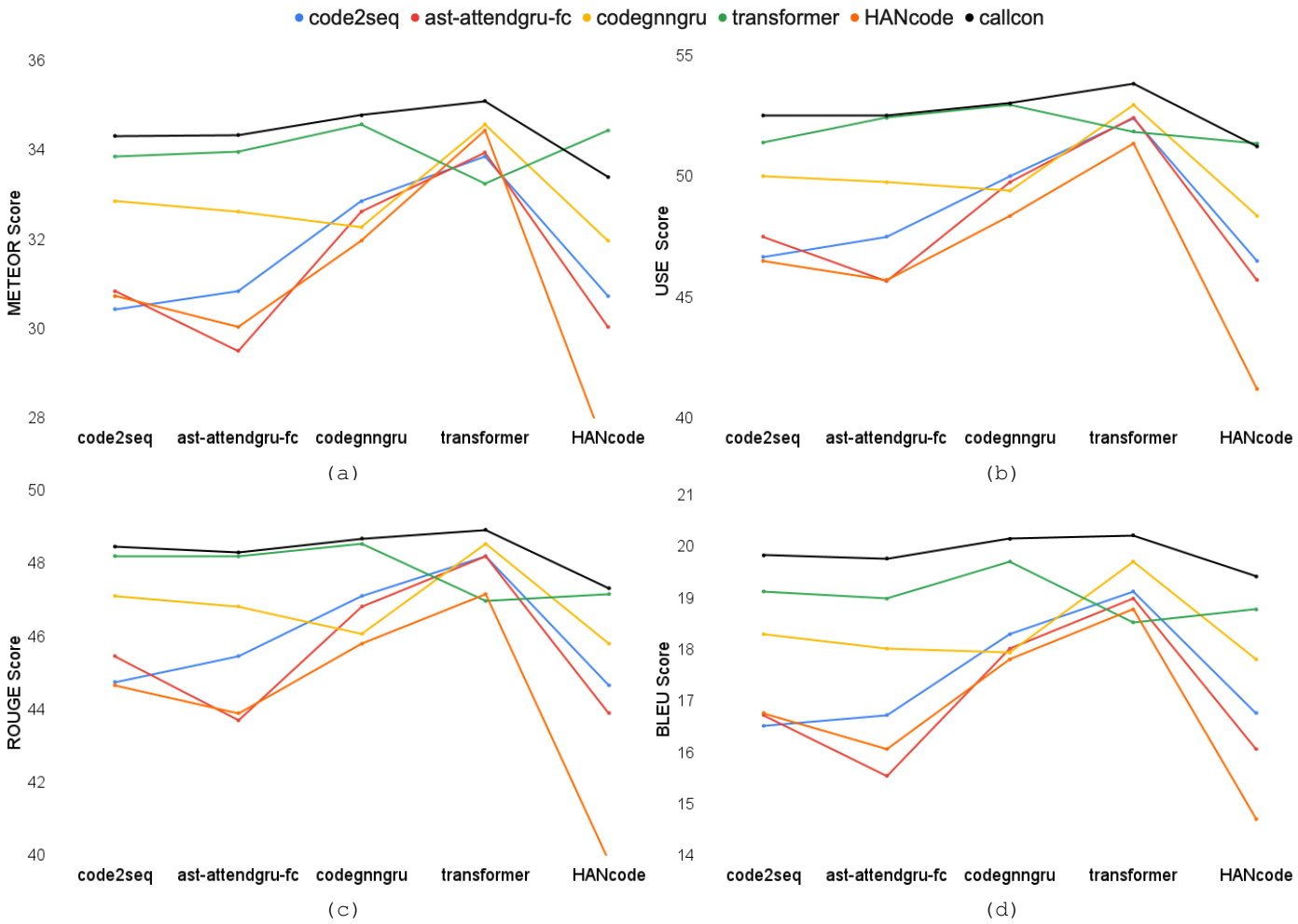

Fig. 4. Graphs comparing metric scores for every baseline ensembled with every other baseline as measured by (a) METEOR scores, (b) USE scores, (c) ROUGE-LCS F1 scores, and (d) BLEU scores. The x-axis names the five baselines and the y axis plots the scores ensembles. We use the baseline scores for model ensembled with itself to maintain the structure of the graph.

Specifically, `callcon` achieves a 33.80 METEOR score which we found to be a statistically significant improvement over all baselines using the T-test. Our approach achieves a 51.59 USE similarity score which is higher than all other baselines, with a caveat that the statistical significance cannot be confirmed over the `transformer` baseline. `callcon` achieves a 19.54 BLUE-A score outperforming the `transformer` baseline by 5.5%. Although it achieved a slightly lower ROUGE-L Precision score, it achieves better Recall and more importantly, better F1 scores. Therefore, we posit that call graph can be used as external context to improve performance of source code summarization models.

6.2 RQ_2: Ensembles

We perform comparison using ensembles and find that our approach improves all other approaches significantly and orthogonally. We present data related to our findings in Table 3 and Figure 4. Table 3 shows the METEOR, USE, ROUGE-LCS and BLEU-A scores for ensembles in the bottom sub-table. Compared to the top subtable, we observe that ensemble with `callcon` significantly improves metric scores for every baseline. For METEOR and USE, the paired T-test compares the ensemble over the baseline model. We observe high t-test values with P-values below 0.01 for every ensemble when compared to the respective baseline. We observe that overall `transformer+callcon` is the best performing ensemble with 35.09 METEOR, 53.84 USE, 48.92 ROUGE-LCS F1, and 20.22 BLEU scores. These represent a 4-7% grain over `transformer` baseline. Therefore, call context can achieve significant orthogonal improvements when ensembled with any of the baselines.

There is a chance that the simple process of creating ensembles can improve performance. Therefore we compare every possible combination of ensembles and find our approach consistently achieves the highest metric scores. In Figure 4 we observe that every baseline when ensembled with `callcon` achieves the highest METEOR, ROUGE, USE, and BLEU scores. The black line in the graph plots scores for baseline + `callcon`. We can see that this line is distinctly above other configurations in terms of (a) METEOR, (c) ROUGE, and (d) BLEU scores. For (b) USE score, we observe that scores for ensembles with `callcon` and `transformer` as the second model overlap for three out of 5 baselines. However, we can see for the `transformer` baseline, our approach is the best performing ensemble compared to any other approach. We posit that `transformer` may be a close competitor to our approach independently. They each perform best for different subsets of the test set. Therefore, we reassert that the increase in metrics achieved by call context is orthogonal to other baselines. We posit that future approaches may benefit from ensembles with call context.

TABLE 4
Scores for our approach and baselines when evaluated for different subsets of the test set. Here wo indicates the number of words in summary that exist in the source code of call graph nodes but not in the the target function.

	METEOR				USE			
model	wo=0	wo>=1	wo>=2	wo>=3	wo=0	wo>=1	wo>=2	wo>=3
code2seq	30.68	22.62	14.5	11.68	46.87	40.46	31.94	32.00
ast-attendgru-fc	29.75	21.58	17.10	09.21	45.88	39.29	34.28	28.97
codegnngru	32.56	22.67	16.62	14.75	49.64	42.51	33.52	35.08
transformer	33.50	24.65	15.83	14.27	52.07	45.38	32.95	33.43
HANcode	27.43	19.38	15.22	12.48	41.4	35.22	29.22	31.47
callcon	**34.07**	**24.91**	**19.12**	**15.5**	**52.14**	**45.72**	**36.35**	**37.42**

6.3 RQ$_3$: Word Overlap

In Table 4 we report the METEOR and USE scores for different values of word overlap. We observe that for all levels of overlap, our approach outperforms all of the baselines. More interestingly, for $wo >= 2$, we find that `callcon` achieves 20.7%, and 10.3% higher METEOR and USE scores respectively over the `transformer` baseline. For that small subset (<1% of the test set) we find that when there are two or more words in the summary that exist in the call graph, but not in the target function, our approach is able to use those words correctly to improve the summary. We also observe that some of these words might also be present in the "file context", which is why `ast-attendgru-fc` seems to improve on this subset as well. Overall, we did not find any level of word overlap where our approach performs worse than a baseline, even when there is no word overlap. This may indicate that our approach doesn't just rely on new words from the call graph functions. We posit this is just one of the ways our approach helps improve summaries.

6.4 RQ$_4$: Configurations

In Table 5 we report metric scores for `callcon`, with values of h ranging from 1 to 5. Recall from Section 3.3 that h is the number of hops in the call context GNN (Figure 3, area 3). A value of $h = 1$ achieved the highest METEOR, USE, and BLEU score. ROUGE-LCS F1 scores increase consistently with the increase of hop size. We observe increase is due to the increase in precision which comes at the cost of decrease in recall scores, while F1 score strikes a balance between precision and recall. Moreover, for all values of h, `callcon` achieves scores higher overall scores than the `transformer` baseline. This my indicate that while h has a slight effect on performance, models with call context achieve better performance than models without call context. The reason that there is low variance in scores for different hops may partially be explained by Section 6.3, as the improvements over the overlap subset may not be affected by hops, the model is able to see those words for any value of h.

In Table 6 we report how metric scores change based on the breadth b during the creation of the call graph. As one would expect, we observe a consistent increase in METEOR, USE, and BLEU scores as we increase the size of the call graph. Recall that we chose $b = 5$ because that is the maximum size of graph we could fit on our GPU. An interesting observation is that even for $b = 1$, we see that our approach achieves a higher BLEU score than all baselines including `transformer`. This is not corroborated by other metrics. Therefore we recommend future work evaluate their approach over several metrics. We make another interesting observation, in that score difference between $b = 4$ (maximum nodes = 41) and $b = 5$ (maximum number of nodes = 61) is very small. Therefore, we recommend $b = 4$ if resources are constrained. Due to limitation of our resources we are unable to test for values of b greater than 5.

TABLE 5
Performance summary for different hops h of the GNN.

	METEOR	USE	ROUGE			BLEU
h	Score	Score	P	R	F1	Score
1	**33.80**	**51.95**	51.41	**46.12**	47.16	**19.54**
2	33.68	51.65	51.97	45.89	47.21	19.41
3	33.74	51.75	51.92	45.98	47.25	19.38
4	33.75	51.76	51.92	45.99	47.26	19.46
5	33.68	51.76	**52.54**	45.73	**47.33**	19.29

TABLE 6
Performance summary for breadth values b for the call graph.

	METEOR	USE	ROUGE			BLEU
b	Score	Score	P	R	F1	Score
1	32.90	50.69	50.75	45.22	46.76	18.70
2	32.94	50.78	50.80	45.25	46.79	18.73
3	33.23	51.43	51.09	45.35	46.92	19.01
4	33.79	51.89	**51.63**	45.77	47.09	19.47
1	**33.80**	**51.95**	51.41	**46.12**	**47.16**	**19.54**

7 Qualitative Experiment

We conduct a qualitative experiment as a supplement to the quantitative experiment in the previous two sections. This qualitative experiment with human experts compares our approach to the reference, human-written summaries.

The scope of this study encompasses only a comparison of `callcon` to the reference summaries in the dataset. The quantitative experiment provides a "breath" evaluation of two large datasets with thousands of samples in the test set of each dataset. However, that experiment uses automated metrics to compare word overlap of predictions to a reference, which leaves a gap related to the overall quality of the predictions along criteria other than word overlap. In other words, even if the predictions perfectly matched the reference every time, the automated metrics do not provide an in depth picture of how programmers perceive these summaries. This qualitative experiment provides this "depth," though on fewer summaries than automated metrics as human studies are expensive in both cost and time.

Note that this experiment measures only *perceptions* of quality, and therefore is only intended to compare perceptions of two sets of summaries. Programmers' perceptions may be affected by a lack of knowledge about the entire software project [75], [76], so a low or high score should not be interpreted globally, i.e., a low accuracy score does not necessarily mean the summary is inaccurate – it is only relative i.e., "approach A is perceived by this human expert as less accurate than approach B."

7.1 Research Questions

The research objective of this experiment is to determine the difference in quality of the summaries we generate to the reference summaries in the dataset as perceived by human experts. We ask the following RQs:

RQ$_5$ What is the level of *overall accuracy* of generated and reference summaries?

RQ$_6$ What is the level of *readability* of generated and reference summaries?

RQ$_7$ What is the level of *completeness* of generated and reference summaries?

RQ$_8$ What is the level of *conciseness* of the generated and reference summaries?

We use the three criteria *accuracy*, *conciseness*, and *completeness* proposed for evaluating source code summaries by Sridhara *et al.* [77] and further recommended by McBurney *et al.* [78]. Accuracy is defined as the perceived level of correctness of the information in the summaries. Completeness is defined as the perception of whether the summaries are missing information that should be in the summary. Conciseness is defined as the perceived level of extraneous material in the summaries. We added the question about readability as a sanity check, to make sure the summaries are sensible and readable english sentences.

7.2 Methodology

Our methodology adheres closely to procedure recommended in earlier studies [77], [78]. For each participant, we randomly select 40 Java methods from a subset of 200 methods we randomly picked from the test set (based on the recommendation of around 90 seconds per method evaluation, with a total workload of around 1 hour per participant). Next, we created a survey which displays a Java method and a summary of that method. The summary is either from our approach or from the reference summary. To avoid biases [79], the survey did not reveal whether the summary was from our approach or the reference. The survey also displayed four statements:

1) Independent of other factors, I feel that the summary is accurate.
2) The summary is missing important information, which limits my understanding.
3) The summary contains a lot of unnecessary information.
4) The summary is written in easily readable english.

Next to each statement was set of radio buttons with a 1-4 scale, ranging from "Strongly Disagree" (1) to "Disagree" (2) to "Neutral(3)" to "Agree" (4) to "Strongly Agree" (5). We recruited **twenty** programmers with at least one year of professional Java development experience. The programmers rate summaries for 40 methods, randomly selected and different for each participant.

The survey output consisted of two sets of 1-5 scores for each of the four statements: one set was for our predicted summaries and one set was for the reference summaries. We report these sets of scores in aggregate (e.g., via boxplots). We also report results of Mann-Whitney U tests between sets of ratings for each quality criterion for predicted and reference summaries. For example, we compare overall accuracy between predicted and reference summaries. The Mann-Whitney U test is appropriate because it is a non-parametric and non-paired. A non-parametric test is suitable because our sample size is not large enough to reasonably assume a normal distribution. A non-paired test is suitable because we have only roughly equal numbers of ratings for the same method, because our survey chose randomly whether to display a generated or reference summary.

7.3 Threats to Validity

The main threats to validity of this study include the participants and the selection of methods from the test set. We recruited 20 participants from a large pool of programmers, but the risk remains that different programmers may give different answers. To mitigate this risk we present box-plots instead of average numbers so we may draw conclusions from the overall distribution. Also, we randomly selected 200 Java methods for the study, of which each participant evaluated 40, which is as large a pool of summaries we could reasonably ask participants to evaluate (give time conflicts with regular career duties). However, the risk remains that our conclusions could vary with methods. Another threat is that our qualitative study does not compare our approach against any of the baselines. We do this following the recommendations from Roy *et al.* [71] that found that score differences of less than 2 points may not be detectable by small human studies such as the one we conduct. Although these improvements are very important, it would require a large human study to detect a small improvement over a baseline. Therefore, our qualitative evaluation compares our approach against the ground truth. The goal of this study is to test if the predicted summaries are reasonably accurate and human readable, when compared with the ground truth.

8 Qualitative Study Results

We answer RQ$_5$ - RQ$_8$ in this section, using the experimental data we collected in our survey in the previous section.

8.1 RQ$_5$: Accuracy

In Figure 5 we show the accuracy ratings for both generated and reference summaries. The mean score for the reference summaries was about 3.4, which is between "neutral" and "agree" for the question about accuracy. In comparison, the mean score for `callcon` was about 3.3, which is comparatively close. We expect a lower accuracy from generated summaries (recall the METEOR scores are in the 30s). The range of scores in the boxplot from first to third quartile, shown by the grey box, is almost identical for both generated and reference summaries. A more interesting observation is that the median score for both is 4, which correlates to "Agree". This means that roughly half of the participants marked "agree" or "strongly agree" for the accuracy of generated and reference summaries. The other half were split between "neutral", "disagree", and "strongly disagree". Although this may indicate that the reference summaries of lower accuracy of quality, we caution against drawing such conclusions due to two reasons. First, we truncate all summaries to 13 tokens due to the design of our neural network and recommendations from related work. Programmers may simply expect a longer summary, and could mark the summary inaccurate. Second, we only show the participants raw code of the target function not the

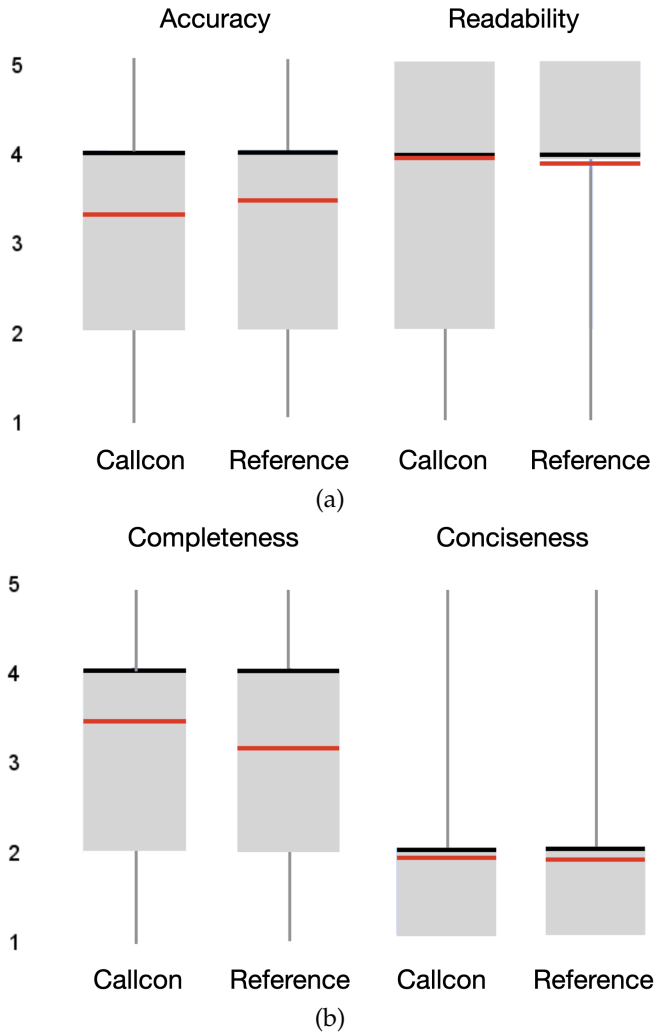

Fig. 5. Human evaluation ratings for (a) Accuracy and Readability, and (b) Completeness and Conciseness. Higher is better for accuracy and readability, while lower is better for completeness and conciseness as they were posed negatively. The black line in the boxplot indicates the median. Red line indicates the mean.

entire project. Programmers may also *perceive* information as inaccurate because they do not understand the entire project, even if the author of the summary actually included relevant insights [75], [76]. If the summaries have words that are not in the target function but in the project, such as found in Section 6.3, participants may mark those as inaccurate.

8.2 RQ_6: Readability

We found that most programmers found both generated and reference summaries readable. The mean score for `callcon` was 3.98, which is a hairline away from "agree". Surprisingly, the mean for reference summaries was slightly lower at 3.88, although the variance between quartiles is smaller for reference summaries. Mean values can be skewed by the opinion of couple of programmers that may simply prefer another sentence structure than the one used by reference summary. Table 8.3 suggests this difference is not statistically significant. In both cases, the median score was 4. This median was the lower end of the distribution for human-written reference summaries. Given both the mean and median, we posit that both generated and reference summaries follow a good sentence structure that programmers find acceptable, with a very small difference in human evaluation between the two.

TABLE 7
Mann-Whitney U test results comparing ratings for `callcon` and reference summaries.

	Accuracy	Readability	Completeness	Conciseness
U	82012	93323	100878	95320
Ue	100453	89142	81588	87146
p-value	0.008	0.531	0.006	0.228

8.3 RQ_7: Completeness

We found that most programmers found both generated and reference summaries to be incomplete – with a mean score above 3 for both, which is closer to "agree" than "disagree". This rating is slightly closer to "neutral" for reference summaries and this difference is statistically significant (See Figure 8.3). Median rating for both is 4, i.e., "agree". Recall that we pose this question negatively, therefore "agree" means that programmers think there is missing information that limits their understanding. One explanation for both generated and reference summaries, in line with related literature, is that the information in the reference summaries is out of date [10]. Another explanation is that we truncate summaries to 13 words as recommended by the original paper that released the dataset [61]. Additionally, for generated summaries, one explanation is that they have "<UNK>" tokens due to the limited vocabulary, while reference summaries do not have any such limitations. The occurrence of an "<UNK>" token would make the summary incomplete, especially if that word is important such as identifier names that usually fall out of vocabulary for our approach. Overall, we find that programmers want more information from source code summaries, and recommend future work to consider training on, and generating longer summaries.

8.4 RQ_8: Conciseness

We found no significant difference in the ratings for conciseness in Table 8.3. In general, programmers tended to view the summaries from both approaches as concise, with a mean and median for both sets of summaries around 2 ("disagree"). Recall that we framed this question in a negative tone, therefore "disagree" and "strongly disagree" indicate that the summary is concise and does not contain useless information. This result is in line with our observation in RQ_5-RQ_7 that a majority programmers felt the summary was missing information but accurate and readable.

9 Conclusion

This paper advances the state-of-the-art with an approach to source code summarization that includes function call context. Call context has long been a resource in software engineering research to improve techniques for a variety of problems, though current literature does not explain how to exploit it for neural models source code summarization. Prior to neural models "taking over" source code summarization research, using call context was mainstream. We show one way to make use of it in recent neural models.

We evaluated different configurations of our approach against several baselines in a quantitative experiment, followed by a comparison to the reference summaries in a qualitative experiment. In the quantitative experiment, we showed that our approach improves over the baselines in a large dataset. We also showed that our approach improves automated summaries over other models for a niche set.

In the qualitative experiment, we show that participants found our summaries reasonably accurate, readable, and concise. However, a majority of them found both generated and reference summaries incomplete.

Broader impacts of this paper include suggestions for future work implied by our experimental results. First, we advise that future work explore different subsets of their dataset where one approach excels over another. We recommend ensembles as a way to combine improvements for future work. Second, we found that metric scores do not correlate with each other in terms of improvements. We recommend future work to use multiple metrics, especially metrics such as METEOR in favor of the standard BLEU metric. Third, the qualitative experiment suggests studying the accuracy of the underlying reference examples. While caution is advised against concluding that the reference summaries are inaccurate (since the ratings are based on perceptions of reading only the method's code and summary), the results do indicate that more study is needed into the reasons for the lower perception of accuracy and completeness of these summaries.

10 Reproducibility

We strongly endorse and encourage reproducibility and future research. We provide our complete datasets, scripts for generating these datasets, our approach implementation, and various other information via our online appendix:

https://github.com/aakashba/callcon-public

Acknowledgments

The authors would like to sincerely thank participants of our qualitative study, NSF for its CCF-2211428 and CCF-2100035 grants, and members of the software engineering community that helped refine several drafts.

References

[1] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*, ser. SIGDOC '99. New York, NY, USA: ACM, 1999, pp. 147–153. [Online]. Available: http://doi.acm.org/10.1145/318372.318577

[2] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[4] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[6] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *J. Syst. Softw.*, vol. 7, no. 4, pp. 341–355, Dec. 1987. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(87)90033-1

[7] L. L. Levesque, J. M. Wilson, and D. R. Wholey, "Cognitive divergence and shared mental models in software development project teams," *Journal of Organizational Behavior*, vol. 22, no. 2, pp. 135–144, 2001. [Online]. Available: http://dx.doi.org/10.1002/job.87

[8] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[9] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.

[11] S. Jiang, C. McMillan, and R. Santelices, "Do programmers do change impact analysis in debugging?" *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, 2017.

[12] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *29th ACM/IEEE International Conference on Program Comprehension (ICPC'21)*, 2021.

[13] P. W. McBurney, C. Liu, and C. McMillan, "Automated feature discovery via sentence selection and source code summarization," *Journal of Software: Evolution and Process*, vol. 28, no. 2, pp. 120–145, 2016.

[14] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 625–636.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[16] P. Rodeghero, S. Jiang, A. Armaly, and C. McMillan, "Detecting user story information in developer-client conversations to generate extractive summaries," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 49–59.

[17] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "Autofolding for source code summarization," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1095–1109, 2017.

[18] S. Badihi and A. Heydarnoori, "Crowdsummarizer: Automated generation of code summaries for java programs through crowdsourcing," *IEEE Software*, vol. 34, no. 2, pp. 71–80, 2017.

[19] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2017, pp. 287–292.

[20] Y. Lu, Z. Zhao, G. Li, and Z. Jin, "Learning to generate comments for api-based code snippets," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2017, pp. 3–14.

[21] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[22] 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*. AAAI Press, 2018, pp. 2269–2275.

[23] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[24] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *International Conference on Learning Representations*, 2018.

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

[26] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *International Conference on Learning Representations*, 2019.

[28] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[29] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description,"

in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 414–421.

[30] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 795–806.

[31] P. Nie, R. Rai, J. J. Li, S. Khurshid, R. J. Mooney, and M. Gligoric, "A framework for writing trigger-action todo comments in executable format," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 385–396.

[32] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," *arXiv preprint arXiv:2005.06980*, 2020.

[33] 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.

[34] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," *International Conference on Mining Software Repositories*, 2020.

[35] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=Xh5eMZVONGF

[36] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid {gnn}," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[37] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[38] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[39] J. Krinke, "Effects of context on program slicing," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1249–1260, 2006.

[40] 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*, 2020, pp. 349–360.

[41] 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)*. IEEE, 2021, pp. 155–166.

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[43] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[44] M. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga, "A comprehensive survey of deep learning for image captioning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 118, 2019.

[45] K. Chen, J. Wang, L.-C. Chen, H. Gao, W. Xu, and R. Nevatia, "Abc-cnn: An attention based convolutional neural network for visual question answering," *arXiv preprint arXiv:1511.05960*, 2015.

[46] R. Dabre, C. Chu, and A. Kunchukuttan, "A survey of multilingual neural machine translation," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–38, 2020.

[47] J. R. Chaudhary and A. C. Patel, "Machine translation using deep learning: a survey," *Int. J. Sci. Res. Sci. Eng. Technol*, 2018.

[48] H. Sharma, M. Agrahari, S. K. Singh, M. Firoj, and R. K. Mishra, "Image captioning: A comprehensive survey," in *2020 International Conference on Power Electronics & IoT Applications in Renewable Energy and its Control (PARC)*. IEEE, 2020, pp. 325–328.

[49] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 104–119.

[50] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 482–498.

[51] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 111–120.

[52] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[53] A. Armaly, P. Rodeghero, and C. McMillan, "A comparison of program comprehension strategies by blind and sighted programmers," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 712–724, 2017.

[54] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 14–23.

[55] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *2008 16th IEEE International Conference on Program Comprehension*. Ieee, 2008, pp. 53–62.

[56] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2010.

[57] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 185–194.

[58] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *Conference on Empirical Methods in Natural Language Processing*, 2018.

[59] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *28th ACM/IEEE International Conference on Program Comprehension (ICPC'20)*, 2020.

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

[61] 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)*, 2019, pp. 3931–3937.

[62] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[63] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *International Conference on Learning Representations*, 2019.

[64] S. Haque, A. Bansal, L. Wu, and C. McMillan, "Action word prediction for neural source code summarization," *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021.

[65] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcml toolkit," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. IEEE, 2011, pp. 173–184.

[66] A. LeClair, A. Bansal, and C. McMillan, "Ensemble models for neural source code summarization of subroutines," in *2021 IEEE International Conference on Software Evolution and Maintenance (ICSME)*, 2021.

[67] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[68] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," *arXiv preprint arXiv:2204.01632*, 2022.

[69] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[70] 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 on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[71] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[72] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.
[73] Y. Zhou, J. Shen, X. Zhang, W. Yang, T. Han, and T. Chen, "Automatic source code summarization with graph attention networks," *Journal of Systems and Software*, vol. 188, p. 111257, 2022.
[74] 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.
[75] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 590–601.
[76] L. C. Briand, "Software documentation: how much is enough?" in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* IEEE, 2003, pp. 13–15.
[77] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.
[78] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.
[79] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, "Yours is better!: participant response bias in hci," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 1321–1330.