Automated Software Engineering (2024) 31:62
https://doi.org/10.1007/s10515-024-00460-x

Revisiting file context for source code summarization

Chia-Yi Su[1] **· Aakash Bansal**[2] **· Collin McMillan**[1]

Received: 17 August 2023 / Accepted: 7 July 2024

Abstract
Source code summarization is the task of writing natural language descriptions of source code. A typical use case is generating short summaries of subroutines for use in API documentation. The heart of almost all current research into code summarization is the encoder–decoder neural architecture, and the encoder input is almost always a single subroutine or other short code snippet. The problem with this setup is that the information needed to describe the code is often not present in the code itself—that information often resides in other nearby code. In this paper, we revisit the idea of "file context" for code summarization. File context is the idea of encoding select information from other subroutines in the same file. We propose a novel modification of the Transformer architecture that is purpose-built to encode file context and demonstrate its improvement over several baselines. We find that file context helps on a subset of challenging examples where traditional approaches struggle.

Keywords Source code summarization · Program comprehension · Software and its documentation · Information systems · Natural language processing · Machine translation

✉ Chia-Yi Su
csu3@nd.edu

Aakash Bansal
abansal1@nd.edu

Collin McMillan
cmc@nd.edu

[1] Department of Computer Science, University of Notre Dame, Holy Cross Dr, Notre Dame, IN 46556, USA

[2] Division of Computer Science, Louisiana State University, Baton Rouge, USA

Published online: 27 July 2024

1 Introduction

A source code "summary" is a short description of that code in natural language. Code summaries have long been at the center of documentation for programmers such as JavaDocs and PyDocs (Kramer 1999), though recent interest is ballooning for interactive programmer tools and educational systems. Tools with code summarization features such as Github Copilot and ChatGPT have captured the public imagination with their ability to read and describe code. Even a short summary such as "books a seat on an airplane flight" can help a programmer quickly understand what a snippet of source code does without having to read the code itself.

The beating heart of almost all source code summarization research is the neural encoder–decoder architecture (Sutskever et al. 2014). The setup is simple. The input to the encoder is the source code to be summarized, while the decoder generates a summary one word at a time. Usually in laboratory settings, the code to be summarized is a single subroutine. The problem with this setup is that not all of the information needed to write a summary of a subroutine is included in that subroutine. Software engineering literature has documented for decades how programmers need summaries to include high-level rationale about why the code exists rather than just restating words from the code itself (Holmes and Murphy 2005; Hill et al. 2009). Current approaches struggle to find the right words to provide this rationale because it often does not exist in single subroutine.

An alternative model was proposed by Haque et al. (2020) using "file context." File context means the other subroutines in the same file as a subroutine under investigation. Haque et al. built an encoder based on recurrent neural networks (RNN) to augment the attention component of the RNN-based encoder–decoder architectures that were prevalent at the time. Experimental results in the original paper and replicated by Bansal et al. (2021) showed how the file context encoder could be added to and improves several RNN-based baselines.

Since then, transformer-based architectures superseded RNN-based ones in nearly all respects. Transformer models tended to be able to outperform RNNs even with the file context encoder, when performance is measured by automated metrics over a whole dataset. E.g., BLEU scores were higher over popular benchmark datasets. Yet these results are slightly misleading. In fact, there is a subset of code for which file context helps, and a subset where it does not. Newer models achieve higher BLEU score by boosting performance on part of the dataset, but not in the way in which file context can help. Unfortunately, there is not a clear means to augment transformer-based models with the encoder proposed by Haque et al. (2020) due to the differences in how transformer and RNN-based architectures handle attention.

In this paper, we introduce a novel modification to the transformer architecture for code summarization designed to encode file context. In a nutshell, our approach has two transformer-based encoders: one for file context and one for the code being described. On the decoder side we use a stack two transformer-based decoders. The "lower" decoder receives the output of the file context

encoder. The "upper" decoder receives the output of the first decoder and the code encoder. This dual encoder design is an alternative to standard transformers that use a giant context window because we cannot simply use this off-the-shelf method. This architecture can further serve as a foundation of the large language models because this transformer architecture allows longer context windows and has better efficiency with better hardware support for scaling up to large language models compared with the RNN-based models.

We show experimentally that our approach outperforms several baselines over three datasets, two in Java and one in Python. We also show how the design decision of a separate encoder for file context helps performance. We show that our approach outperforms other transformer architecture that use a large context window or "prompt". But more importantly, we show that the performance increase is due to the file context instead of other factors such as scale. We also conduct a human study and report programmer opinions on the quality of summaries generated by our approach, when compared against the best performing baseline.

2 File context

"File context" is a term in Software Engineering research literature that means the other information in the same file as a section of code under investigation (Holmes and Murphy 2005; Hill et al. 2009; Guerrouj et al. 2014; Ding et al. 2022). In this paper, as in the earlier work by Haque et al. (2020), the sections of code under investigation are subroutines, and the file context includes a few of the other subroutines in the same file. File context has been cited for decades as a key source of information for understanding source code, since code lives in an ecosystem of interdependent software components.

```
public void setIntermediate(String intermediate)
{  this.intermediate = intermediate;  }
```

reference	sets the intermediate value for this flight
LeClair et al (2019)	sets the intermediate value for this <UNK>
Haque et al (2020)	sets the intermediate value for this flight

1. public void set airline name string airline
2. public void set destination string destination
3. public long get flight id return flight id
4. public void set flight id long flight id this flight id
5. public void set flight number string flight number

Example 1: (upper) Source code for method ID 26052502 in the `java-long` dataset. (mid) Reference summary and summaries generated by an RNN baseline and the same baseline enhanced with file context. (lower) The file context.

Consider Example 1 from Haque et al. (2020). The Java method `setIntermediate()` is a simple setter type function. Most baselines are capable of writing

a summary to this effect, such as the example from LeClair et al. (2019) shown. But this summary is not that useful because even a novice programmer is likely to expend almost no effort reading the code. What is useful is to know is *why* the value is set (Roehm et al. 2012).

In Example 1, this why-information is only evident when we consider the file context. Note the file context consists of terms such as "airline", "flight", and "destination" in the signature of other methods. The model using file context was able to find and learn to use the word "flight" correctly, thus predicting a more useful summary for the method.

Cases like these are very common in code summarization samples and have a strong impact on model performance. Consider the distribution of our dataset presented in Table 1. The term *wo* refers to the number of words that are in both the reference summary and file context, but are not present in the method itself:

$$wo = |(FCW - MW) \cap SW| \tag{1}$$

Here, FCW is the set of words in the file context, MW is the set of words in the method, and SW is the set of words in the summary. Note that only around 35% of methods have $wo = 0$, a majority contain at least one word in the file context that is not present in the method itself.

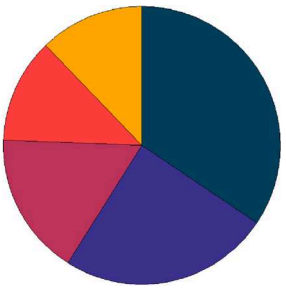
Meanwhile, the column METEOR refers to the METEOR score of a Transformer approach proposed by Ahmad et al. (2020). This baseline approach does not see or use file context. The METEOR score is 41.44 for when $wo = 0$, but drops over 25% to 30.78 for when $wo = 1$. We observe that the METEOR score continues to decline for subsets with a higher *wo*. This means summaries that use more words from the file context are often more difficult to write for standard approaches.

3 Related work

In Fig. 1 we provide an overview of selected related work from the last 5 years. The related work can be broadly classified into four families.

Table 1 Model performance declines as word overlap between the code summary and the file context increases

Word overlap	METEOR	Portion of dataset
wo = 0	41.44	
wo = 1	30.78	
wo = 2	29.26	
wo = 3	28.65	
wo >= 4	21.88	



The first column shows *wo* which is the number of words in the summary that are in the file context but not in the subroutine. The second column shows METEOR scores for a typical Transformer baseline

	R	A	G	T	C
Hu et al (2018b)	x				
Hu et al (2018a)	x	x			
Wan et al (2018)	x				
Liang and Zhu (2018)	x				
Alon et al (2019)	x	x			
LeClair et al (2019)	x	x			
Nie et al (2019)	x				
Haldar et al (2020)	x	x			
Ahmad et al (2020)				x	
Haque et al (2020)	x				x
LeClair et al (2020)	x	x	x		
Feng et al (2020)				x	
Wei et al (2020)	x	x			x
Bansal et al (2021)	x				x
Zügner et al (2021)	x	x			
Ahmad et al (2021)				x	
Liu et al (2021)		x	x		x
Li et al (2022)		x		x	
Kuang et al (2022)		x	x	x	
Tang et al (2022)		x		x	
Ahmed and Devanbu (2022)				x	
this paper				x	x

Fig. 1 Overview of select related work. Column R denotes use of RNNs such as GRU/LSTM. A denotes using Abstract Syntax Tree. G denotes graph neural networks based encoder. T denotes Transformer based encoder. C denotes the use of context

AST-Flat is a family of approaches marked by Column A in Fig. 1 that uses the Abstract Syntax Tree (AST) as a sequence of nodes. In 2018, Hu et al. (2018a) introduced DeepCom, a model that encodes code tokens and AST nodes together using LSTMs. In 2019, LeClair et al. (2019) introduced a similar model ast-attendgru, except that they use GRUs instead of LSTM and they decoupled the AST and source code sequence to separate encoders. They found that learning separate representations of code and structure helps generate better natural language summaries.

AST-GNN is a family of approaches marked by Column G in Fig. 1 that use Graph Neural Networks (GNN). In 2020, LeClair et al. (2020) introduced a hybrid GNN-RNN model that encodes AST as a graph using a GNN layer and combines it with a GRU based source code sequence encoder. They found that, compared to a flat representation of AST, a GNN can learn to better place AST nodes in the embedding space by using the edges of the AST. Since that landmark paper, a few approaches have been introduced that use GNNs to encode code structure for code summarization (Kuang et al. 2022; Liu et al. 2021).

Transformers is a family of approaches marked by Column T in Fig. 1. In 2020, Ahmad et al. (2020) introduced a Transformer based approach for code summarization. They found that the self-attention mechanism helps transformers better map words in the encoder to words in the decoder. Since then, several approaches have been introduced that use a network of transformers (Ahmad et al. 2021; Tang et al. 2022; Kuang et al. 2022). In 2022, Li et al. (2022) introduced SeTransformer that encodes decoupled AST and source code, using a network of transformers and CNNs.

Context is a family of approaches marked by Column C in Fig. 1 that mainly relies on contextual information. In 2020, Haque et al. (2020) introduced an encoder "FC", that encodes file context, i.e., the summaries around the target function in the same file as a separate encoder. They add this encoder to several RNN based baselines. They found that sometimes the important information needed to generate accurate summaries is not present in the target function, but can be found in the functions around it in the same file. Our paper is an extension of that work.

The ideas of integrating the context into the model have been used in different program comprehension tasks. For example, Li et al. (2021); Wang et al. (2021) integrated the file context to improve the results of method name suggestions. However, the workhorse of those papers is RNN-based models. We designed a novel approach to integrate file context into transformer-based models because of the difference in attention mechanism.

In 2021, Bansal et al. (2021) introduced another context-based encoder "PC" that encodes several files from the project as project context. They disclosed that the computational cost of their encoder is exponentially higher than "FC". Therefore, we leave "PC" for transformer-based models as our future works.

Since 2020, Large Language Models (LLMs) have become increasingly popular in several domains of applied NLP research. In 2020, Feng et al. (2020) introduced CodeBERT that uses a stack of Transformers which are bidirectionally trained. The network models source code syntax by learning to predict randomly masked tokens proposed. In 2021, Ahmad et al. (2021) proposed PLBART, using graph neural networks to learn programming language generation. Both report modest improvements in performance for code summarization, while reporting high training costs, which are critical for academic research.

In 2020, Wei et al. (2020) introduced an approach to use AST and exemplar representations of source code as context. They used these representations to fetch similar methods from a database to help refine the search for a more accurate summary. In 2021, Liu et al. (2021) introduced an approach to retrieve similar code properties from a data as context, then model that context as a property graph using GNNs. These are examples of retrieval-based techniques that use summaries from similar methods as an input to the model. In contrast, we make every effort to remove comments and doc-strings from our context input. Retrieval-based techniques are also more susceptible to data leaks between the training and test set that could skew the results over unseen samples. Therefore, we do not replicate their work as a baseline.

Overall, we observe a clear trend from GRU/LSTM based models that mainly relied on source code and AST—to transformer based techniques, with a few approaches increasingly incorporating some contextual information in their design.

This paper is a natural evolution of that trend, in that we present an in-depth analysis of how transformers and file context can be used to improve the state-of-the-art in source code summarization.

4 Model design

Our model is essentially the basic Transformer architecture, but with two key changes. First, we add a File Context Encoder to learn a representation of the other subroutines in the same file as the target subroutine. Second, we add another multi-head attention and fully-connected layer unit (which we call an *XFormBlock*) to the decoder, and use the output of the File Context Encoder as an input to this unit. The "target subroutine" is the subroutine for which we are writing a summary. The *XFormBlock* is an abstraction of common attention, FCN, and normalization/regularization operations used in Transformer-like models (see the next section for more details).

The intuition behind our changes is two fold: (1) to allow the model to "see" the file context prior to the target subroutine's source code when learning to predict words in the summary, and (2) to avoid using giant context windows such as large prompts. The expectation for giant context windows is that a large model will eventually learn context from all the information. But, that approach is computationally expensive and requires much a large amount of data. We show Sect. 6.3 that giant context window approach does not provide expected improvements, at least not out-of-the-box.

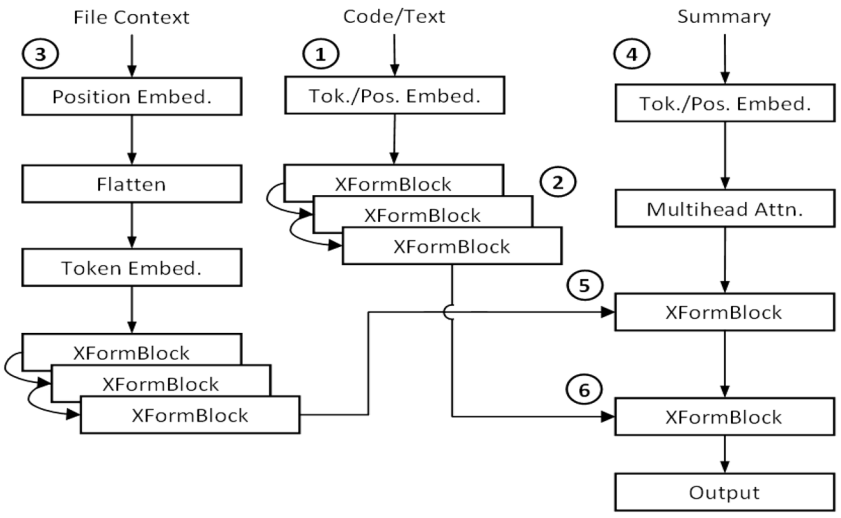


Fig. 2 Overview of our model design. Area 1 and 2 denote the target subroutine encoder. Area 3 highlights the novel addition of the file context encoder. Area 4 denotes the summary decoder. Area 5 and 6 denote out novel modifications to use dual encoders to improve the output prediction

Consider the overview in Fig. 2. There are two encoders and one decoder. One encoder is the target subroutine encoder, shown in Fig. 2, area 1 and 2. The input to this encoder is the first t tokens of the target subroutine's source code (area 2). These are encoded using L *XFormBlock* units, as recommended by Ahmad et al. (2020) for code summarization (area 2).

Another encoder is the file context encoder shown in area 3. The input is a matrix containing the first m tokens for n subroutines from the same file. The vocabulary for both of these encoders is the same, though unlike Haque et al. (2020), we use separate embedding spaces for each encoder. We used the same preprocessing technique as proposed by Bansal et al. (2021), which is very light with only the following operations: (1) extract all tokens based on the language specification (e.g., a "token" in Java is what the Java lexical analyzer sees), (2) split all tokens by underscore and camel case, and (3) change all tokens to lower case.

The input to the decoder a sequence of words in the summary for the target subroutine (area 4). The output is the next word in the summary. The architecture of the decoder is the same as a typical Transformer model, except that for each *XFormBlock* in the original model, we have two. One takes as input the File Context Encoder output (area 5) while another takes the Target Subroutine Encoder's output (area 6). As in other encoder–decoder architectures, we build the output summary one word at a time by feeding the summary predicted up to that point, back into the decoder. We use the teacher forcing strategy during training.

In the following subsections, we provide a formal definition of our model in Named Tensor Notation, formalized and proposed by Chiang et al. (2021). In our experiments, we call our approach `transformer-fc` .

4.1 File context encoder

The file context encoder forms a representation of the other subroutines in the same file as the target subroutine. The input to the file context encoder is an n x m matrix where n is the maximum number of other subroutines the encoder can intake and m is the maximum number of tokens per subroutine. We define the space in which the file context encoder operates as:

$$\mathbf{X}^{fc} \in \mathbb{N}^{\texttt{fcn}[n]\times\texttt{token}[m]}$$
$$\mathbf{P}^{fc} \in \mathbb{R}^{\texttt{fcn}[n]\times\texttt{token}[m]\times\texttt{dim}[e]}$$
$$\mathbf{T}^{fc} \in \mathbb{R}^{\texttt{fcn}[n]\times\texttt{token}[m]\times\texttt{dim}[e]}$$
$$\mathbf{E}^{fc} \in \mathbb{R}^{\texttt{fcntoken}[nm]\times\texttt{dim}[e]}$$
$$\mathbf{A} \in \mathbb{R}^{\texttt{fcntoken}[nm]\times\texttt{dim}[e]}$$

where $\mathbf{X}^{fc}$ is the input with each element as an entry in the source code vocabulary, $\mathbf{P}^{fc}$ and $\mathbf{T}^{fc}$ are position and token embeddings, and $\mathbf{E}^{fc}$ is the combined embedding space. $\mathbf{A}$ is the file encoder output space. We use a learnable position embedding proposed as "Method 3" by Huang et al. (2020), in which we assign each position in each subroutine (e.g., [0..m) for each subroutine, rather than [0..nm) across the entire

file). The token embedding is a typical learned space, and $\mathbf{E}^{fc}$ is the elementwise sum of $\mathbf{P}^{fc}$ and $\mathbf{T}^{fc}$, reshaped to a 2d $nm \times e$ matrix from 3d $n \times m \times e$:

$$
\begin{aligned}
\mathbf{I}^1 &= [0..n), \mathbf{J}^1 = [0..m) \\
\forall i, \forall j, i \in \mathbf{I}^1, j \in \mathbf{J}^1 \\
\mathbf{P}^{fc}_{\texttt{fcn}(i),\texttt{token}(j)} &= W^1 \underset{\texttt{pos}}{\odot} \mathbf{J}^1 + b^1 \\
\mathbf{T}^{fc}_{\texttt{fcn}(i),\texttt{token}(j)} &= W^2 \underset{\texttt{token}}{\odot} \mathbf{X}^{fc}_{\texttt{fcn}(i),\texttt{token}(j)} + b^2 \\
\mathbf{E}^{fc} &= (\mathbf{P}^{fc} + \mathbf{T}^{fc})_{(fcn,token)\rightarrow fcntoken} \\
W^1 &\in \mathbb{R}^{\texttt{hidden}[e]\times\texttt{pos}[\texttt{m}]} \qquad b^1 \in \mathbb{R}^{\texttt{hidden}[e]} \\
W^2 &\in \mathbb{R}^{\texttt{hidden}[e]\times\texttt{token}[\texttt{m}]} \qquad b^2 \in \mathbb{R}^{\texttt{hidden}[e]}
\end{aligned}
\tag{2}
$$

We apply Transformer-like operations via L layers of self-attention and fully-connected networks, with dropout and layer normalization. For brevity, we abstract these operations as the function $XFormBlock(Q, K, V)$, as it has been described as an "Encoder Block" with parameters Query, Key, and Value by numerous authors starting with Vaswani et al. (2017).

$$
\mathbf{A}^0 = XFormBlock^0(\mathbf{E}^{fc}, \mathbf{E}^{fc}, \mathbf{E}^{fc}) \tag{3}
$$

$$
\begin{aligned}
&\vdots \\
\mathbf{A}^L &= XFormBlock^{L-1}(\mathbf{A}^{L-1}, \mathbf{A}^{L-1}, \mathbf{A}^{L-1}) \\
\mathbf{A} &= \mathbf{A}^L
\end{aligned}
\tag{4}
$$

4.2 Target subroutine encoder

The target subroutine encoder forms a representation of the target subroutine itself. This encoder is a essentially a Transformer-like encoder described by Vaswani et al. (2017), except with the learned position embedding we also use in the file encoder. The operation space is:

$$
\begin{aligned}
\mathbf{X}^s &\in \mathbb{N}^{\texttt{token}[m]} \\
\mathbf{P}^s &\in \mathbb{R}^{\texttt{token}[m]\times\texttt{dim}[e]} \\
\mathbf{T}^s &\in \mathbb{R}^{\texttt{token}[m]\times\texttt{dim}[e]} \\
\mathbf{E}^s &\in \mathbb{R}^{\texttt{token}[nm]\times\texttt{dim}[e]} \\
\mathbf{B} &\in \mathbb{R}^{\texttt{token}[nm]\times\texttt{dim}[e]}
\end{aligned}
$$

where $\mathbf{X}^s$ is the input where each element is an entry in the vocabulary, $\mathbf{P}^s$ and $\mathbf{T}^s$ are position and token embeddings, and $\mathbf{E}^s$ is the combined embedding space:

$$\begin{aligned} \mathbf{J}^2 &= [0..t) \\ \forall j, j \in \mathbf{J}^2 & \\ \mathbf{P}^s_{\texttt{token}(j)} &= W^3 \underset{\texttt{pos}}{\odot} \mathbf{J}^2 + b^3 \end{aligned} \tag{5}$$

$$\mathbf{T}^s_{\texttt{token}(j)} = W^4 \underset{\texttt{token}}{\odot} \mathbf{X}^s_{\texttt{token}(j)} + b^4 \tag{6}$$

$$\begin{aligned} \mathbf{E}^s &= \mathbf{P}^s + \mathbf{T}^s \\ W^3 &\in \mathbb{R}^{\texttt{hidden[}e\texttt{]}\times\texttt{pos[m]}} \qquad b^3 \in \mathbb{R}^{\texttt{hidden[}e\texttt{]}} \\ W^4 &\in \mathbb{R}^{\texttt{hidden[}e\texttt{]}\times\texttt{token[m]}} \quad b^4 \in \mathbb{R}^{\texttt{hidden[}e\texttt{]}} \end{aligned} \tag{7}$$

Followed by *L XFormBlock* layers:

$$\begin{aligned} \mathbf{B}^0 &= \mathit{XFormBlock}^0(\mathbf{E}^s, \mathbf{E}^s, \mathbf{E}^s) \\ &\vdots \\ \mathbf{B}^L &= \mathit{XFormBlock}^{L-1}(\mathbf{B}^{L-1}, \mathbf{B}^{L-1}, \mathbf{B}^{L-1}) \\ \mathbf{B} &= \mathbf{B}^L \end{aligned} \tag{8}$$

4.3 Decoder

Our decoder is a Transformer-like decoder, except with **two** *XFormBlock* layers for each one in the typical design. In most Transformer-like decoders, there is a self-attention layer followed by *XFormBlock* layers. However in our decoder, we have an *XFormBlock* that receives **A** and another that receives **B**:

$$\begin{aligned} \mathbf{Y}^d &\in \mathbb{N}^{\texttt{word[}w\texttt{]}} \\ \mathbf{P}^d &\in \mathbb{R}^{\texttt{word[}m\texttt{]}\times\texttt{dim[}e\texttt{]}} \\ \mathbf{T}^d &\in \mathbb{R}^{\texttt{word[}m\texttt{]}\times\texttt{dim[}e\texttt{]}} \\ \mathbf{E}^d &\in \mathbb{R}^{\texttt{word[}nm\texttt{]}\times\texttt{dim[}e\texttt{]}} \\ \mathbf{C} &\in \mathbb{R}^{\texttt{word[}nm\texttt{]}\times\texttt{dim[}e\texttt{]}} \end{aligned} \tag{9}$$

The decoder word embedding space:

$$\begin{aligned} \mathbf{J}^3 &= [0..w) \\ \forall j, j \in \mathbf{J}^3 & \\ \mathbf{P}^d_{\texttt{token}(j)} &= W^5 \underset{\texttt{pos}}{\odot} \mathbf{J}^3 + b^5 \end{aligned} \tag{10}$$

$$\mathbf{T}^d_{\texttt{token}(j)} = W^6 \underset{\texttt{word}}{\odot} \mathbf{X}^s_{\texttt{word}(j)} + b^6 \tag{11}$$

$$\begin{aligned} \mathbf{E}^d &= \mathbf{P}^d + \mathbf{T}^d \\ W^5 &\in \mathbb{R}^{\texttt{hidden}[e]\times\texttt{pos}[\texttt{w}]} \quad b^5 \in \mathbb{R}^{\texttt{hidden}[e]} \\ W^6 &\in \mathbb{R}^{\texttt{hidden}[e]\times\texttt{word}[\texttt{w}]} \quad b^6 \in \mathbb{R}^{\texttt{hidden}[e]} \end{aligned} \tag{12}$$

And the *XFormBlocks*:

$$\mathbf{C}^0 = XFormBlock^0(\mathbf{E}^t, \mathbf{E}^t, \mathbf{E}^t) \tag{13}$$

$$\mathbf{C}^1 = XFormBlock^1(\mathbf{C}^0, \mathbf{A}, \mathbf{A}) \tag{14}$$

$$\begin{aligned} \mathbf{C}^2 &= XFormBlock^2(\mathbf{C}^1, \mathbf{B}, \mathbf{B}) \\ \mathbf{C} &= \mathbf{C}^2 \end{aligned} \tag{15}$$

Followed by an output layer to predict the next word in the summary sequence, where $\mathbf{Y}^p$ is the output prediction space:

$$\begin{aligned} \mathbf{Y}^p &\in \mathbb{N}^{\texttt{vocab}[w]} \\ \mathbf{Y}^p &= \sigma(W^7 \underset{\texttt{vocab}}{\odot} \mathbf{C}^s_{(word,dim)\rightarrow vocab} + b^7) \\ W^7 &\in \mathbb{R}^{\texttt{hidden}[e]\times\texttt{vocab}[\texttt{m}]} \qquad b^7 \in \mathbb{R}^{\texttt{hidden}[e]} \end{aligned}$$

In practice, the output is a vector with the length z of the summary vocabulary. We apply a softmax activation to this vector. During prediction, an argmax operation on this vector indicates the index of the predicted word in the vocabulary.

4.4 Hyperparameters

For reproducibility and transparency, we list key model hyperparameters in Table 2.

Table 2 Hyperparameters for our model

Parameter	Description	Java	Python
n	Subroutines in file context	20	20
m	Tokens per subroutine	25	25
t	Tokens in target subroutine	50	50
w	Words in summary	13	13
v	Source code vocabulary size	75 k	100 k
z	Summary vocabulary size	10908	11000
e	Embedding dimensions	128	128
L	Stacked XFormBlock layers	3	1
h	Attention heads	3	3
b	Batch size	8	50
r	Learning rate	3e−4	3e−4

We chose n, m, t, and w as recommendations from Haque et al. (2020) and Bansal et al. (2021). We chose v and z as recommended by LeClair and McMillan (2019). We chose e, L, h, b, and r from our own pilot studies and hardware limitations. Note that hyperparameters differ between Java and Python datasets because of semantic differences between the two languages, which require different parameters for comparable performance. Java and Python model performances must also be considered independent of each other.

5 Experiment design

This section describes our experiment to evaluate our model, including our research questions, methods, datasets, baselines, metrics, hardware and software versions, and threats to validity.

5.1 Research questions

Our research objective is to determine the degree of difference in performance between our model and baseline models. Thus, We ask the following Research Questions (RQs):

RQ1 What is the difference between our dual encoder design models and baseline models over all summaries in a dataset including the summaries contain words in file context and do not contain words in file context, in terms of automated metrics?

RQ2 What is the difference between our dual encoder design model and baseline models when we consider the summary that includes words from the file context only?

RQ3 What is the impact of our dual encoder design on model performance, when compared against combining inputs into one giant file as the current prompt methods?

The rationale behind RQ1 is to be consistent with years of related work. Almost all code summarization papers over the past decade have used automated metrics to measure improvement over baselines. This methodology is popular because any meaningful improvement should be observable over a whole dataset.

The rationale behind RQ2 is that model changes to add file context may provide more improvement in the subset of the dataset in which words in the summary are only present in the file context. We seek to evaluate our approach, when compared with the baselines, over these subsets of our datasets.

The rationale behind RQ3 is to evaluate the impact of our dual encoder design against a single input approach such as those popularized by Large Language Models (LLMs). State-of-Practice in industry and now research venues, is to use these LLMs, which are designed for instructional and conversational use. These models accept a single input sequence which contains all the information as a user query.

5.2 Research methods

Our research method is typical of many papers on code summarization and encoder–decoder models such as LeClair et al. (2019):

To answer RQ1, we first train our approach and each of the baselines models independently on different datasets for a maximum of ten epochs using the hyperparameters in Sect. 4. Then, we choose the epoch with the highest validation accuracy. We load the model at that epoch to predict the summaries for the unseen test set, which is 4–5% of the whole dataset. Then, we compute automated metric scores between predicted summaries and the reference summaries (described in Sect. 5.6).

To answer RQ2, we start with the same predictions we computed for RQ1. While computing metrics, we partition the test set based on the value of *wo*. We then report average metric scores for each subset.

To answer RQ3, we implemented two alternative approaches for modeling file context as described in Sect. 5.5. We follow the same procedures for evaluation of these models as RQ1, except that we only train `llama-lora` for one epoch, which takes roughly 60 h. We report metric scores for comparison against our approach and a previous approach that uses file context.

5.3 Datasets

We use three datasets in this paper. One is called `funcom-java` and contains around 2 m Java methods. The dataset originates from LeClair and McMillan (2019) and uses a split-by-project configuration to reduce the risk of data duplication. The version we use is from Bansal et al. (2021) who apply additional filters to reduce code clones as recommended by Allamanis (2019).

The second dataset is called `funcom-java-long` which was created by Bansal et al. (2023b) and consists of the top 10% longest methods from `funcom-java` in terms of number of tokens and implement fixes recommended by Shi et al. (2022). We focus on these longer functions due to an observation by Haque et al. (2021) that many Java methods in the `funcom-java` dataset are short getters/setters. This observation was corroborated by Bansal et al. (2023b). The found that longer methods are more challenging while shorter methods are easy to summarize using even the most basic code summarization models. This dataset consists of roughly 190k java methods.

Finally, we create a new dataset we call `funcom-python` that we extracted from a data dump of 40k Python projects downloaded from Github. We use the same cleaning and splitting procedures as LeClair and McMillan (2019) and Bansal et al. (2021). We also favor Python functions that are top 10% longest methods as recommended by Haque et al. (2021) and Bansal et al. (2023b). The final dataset consists of 270k python functions and the corresponding AST sequence, AST graphs, file context, and header summaries.

5.4 Baseline models

We compare our model to the following baselines. We reimplemented all baselines in our own framework to control experimental variables such as software version. We chose these baselines as they are representative of different families of approaches (see Sect. 3) which have different advantages and disadvantages.

transformer-alt This is a baseline of our own design using the same architecture of our proposed model in Sect. 4. The only difference is that we substitute the file context input with the subroutine code itself (repeated n times so model size is identical). More formally, prior to Eq. (2), we set $\mathbf{X}^{fc} = [\mathbf{X}^{s}{}_{\times n}]$. The reason for this baseline is that our model is larger than the normal `transformer`, so it is possible that improvements come from scale and not file context.

ast-attendgru An approach by LeClair et al. (2019) that mainly benefits from a flat representation of the code's Abstract Syntax Tree (AST) generated by the Structure-Based-Traversal (SBT) method. The model architecture consists of two GRU-based encoders, one for the source code tokens and the other for the AST.

ast-attendgru-fc The original approach by Haque et al. (2020) that this paper extends. The model architecture consists of three GRU based encoders—two for the source code tokens and AST tokens respectively, and a third to encode the file context using a number of GRUs consistent with the number of methods from the file being represented as context.

codegnngru A graph neural network-based approach by LeClair et al. (2020). This model architecture consists of two encoders, a GRU to encode the flat sequence of the AST graph, and GNN-GRU hybrid encoder to encode a graphical representation of the AST using the edges between AST tokens and source code.

transformer An approach by Ahmad et al. (2020). Essentially this approach uses a vanilla Transformer encoder–decoder design. The model architecture consists of a single encoder that uses a positional encoding and two attention heads.

setransformer A recent approach by Li et al. (2022) that uses a Transformer-CNN hybrid model to learn representation of the AST. The model architecture consists of two Transformer-based encoders, one for the source code and the other for the AST. Each of these encoders also consists of a CNN layer for feature reduction to reduce computational load.

5.5 Alternate approaches

We also test two alternate approaches for encoding file context as stand-ins for transformers and large language models with a giant context window. We combine the source code of the target subroutine and file context into one input. We use these two baselines exclusively to answer RQ3:

transformer-comb This is a Transformer-based encoder–decoder baseline, where we combine the source code and the file context into a single input. The model architecture is similar to `transformer`, scaled up for a much longer input, such that the file context is concatenated to the source code after tokenization.

llama-lora A recent approach by Hu et al. (2021) that proposes a framework for fine-tuning LLaMA, an instructional Transformer-Based Large Language Model (LLM) by Touvron et al. (2023). This is a decoder-only instructional and conversational LLM that accepts a single prompt and predicts the most likely words to complete the prompt. We fine tune the 7 billion parameter LLaMA model for one epoch. We create a fine-tuning prompt following the typical strategy by Hu et al. (2021):

1. Instruction: the text "describe the following function"
2. Input: < source code for the target method>
3. FC: < source code of the functions in the file context>
4. Output: < the reference summary sequence>

Note, the purpose of these approaches is to compare our model against an off-the-shelf approach by Hu et al. (2021) which consists of taking a large model and feeding it all the data in a single input. The `llama-lora` model is considerably larger than any of our other baselines. Therefore, we only run this baseline over the smaller datasets, namely `funcom-java-long` and `funcom-python`. Each of these datasets took roughly 110 h of training and inference. Estimated duration for training and inference over `funcom-java` would be 980 h (~ 6 weeks).

5.6 Metrics

We use three metrics to compare predicted summaries to their reference in the datasets:

METEOR is a well-known metric that calculates the harmonic mean of unigram precision and recall, introduced by Banerjee and Lavie (2005). We use this metric because Roy et al. (2021) conducted a study with human experts and found that it better correlates to human judgement compared to BLEU.

USE is an encoder-based semantic similarity metric proposed by Haque et al. (2022). They conducted a human study and found encoder-based metrics have a relatively high correlation with the judgement of human experts compared to BLEU.

BLEU is an n-gram based popular metric used to evaluate code summarization by almost all related work over the last decade. We report BLEU to be consistent with related work, though with the caveat that METEOR and USE are now preferred.

5.7 Hardware and software

The hardware we used for training and inference for our approach and baselines: AMD 5900x CPU, 2xTITAN RTX with 24GB VRAM each, and 128GB system memory.

Software that we used includes CUDA 11.2, Tensorflow 2.9.2, Python 3.10, Pandas 1.4, NLTK 3.6, Ubuntu LTS 22.04.

5.8 Threats to validity

Like all experiments, this paper carries threats to validity that could change our conclusions under different experimental conditions. There are three main threats we try to mitigate in the design of our experiment:

The first threat lies in the datasets. We attempt to mitigate this risk by using large datasets in two different programming languages extracted from a diverse set of repositories. We also clean and process the dataset using techniques recommended in related work (LeClair and McMillan 2019; Bansal et al. 2021), in an attempt to mitigate the risk of data leaks and skewed results.

The second threat lies in the automated metrics we use to evaluate the performance of our approach against the baselines. We attempted to mitigate this risk by reporting three metrics, following recommendations by latest related work.

The third threat lies in the hyperparameters of our model. We chose these hyperparameters based on limited pilot studies and related work, as we do not have resources for a large grid search. In theory, different hyperparameters could alter our

Table 3 Metric scores for the three datasets. Our model is `transformer-fc`, while `transformer-alt` is our model without file context input

Model	`funcom-java`			`funcom-java-long`			`funcom-python`		
	M	U	B	M	U	B	M	U	B
ast-attendgru	35.30	52.89	18.33	33.21	50.12	18.94	26.80	43.75	16.92
ast-attendgru-fc	35.71	52.94	18.94	33.52	50.48	18.91	27.72	44.93	16.82
codegnngru	35.82	53.26	18.77	32.98	49.85	18.75	26.11	42.36	17.33
transformer	35.68	54.03	18.29	33.18	51.27	18.52	26.74	43.86	15.68
setransformer	36.01	53.43	18.71	32.47	49.60	18.51	27.35	43.70	17.60
transformer-alt	35.84	53.98	18.54	33.98	52.62	19.67	28.47	**45.64**	17.58
transformer-fc	**37.12**	**54.61**	**20.18**	**34.67**	**52.77**	**19.90**	**28.58**	45.45	**18.21**

The best results across all models are given in bold

results and conclusions. To promote transparency and reproducibility, we report and discuss these hyperparameters in Sect. 4.4.

In additional to these threats, minor variations in performance can be seen due to different hardware and software versions. Therefore, we report the hardware and software versions used for this paper in Sect. 5.7.

6 Experiment results

In this section we discuss our experimental results for the four research questions.

6.1 RQ1: overall performance

Table 3 shows the overall performance for each model and each dataset. We found that performance for `transformer-fc` was around 3%, 1%, and 6.5% higher than the nearest baseline for METEOR, USE, and BLEU, respectively, over the `funcom-java` dataset. The differences were narrower for the `funcom-java-long` dataset: 2%, 0.3%, 1%. Results are mixed in Python, as `transformer-fc` had the highest scores for METEOR and BLEU, but not USE.

We make a few observations in these results. First, BLEU scores for our approach tend to show more improvement than other scores. One possible explanation for this difference is BLEU's dependence on exact word matches, while METEOR and USE have mechanisms for reducing this dependence. It is likely that our model is able to find more exact matches due to the additional information in the file context. Second, the difference between our model and baselines is greatest in `funcom-java`. There are two likely explanations: (1) `funcom-java` has around ten times more examples and therefore may be providing more opportunity for `transformer-fc` to learn from a more diverse dataset, and (2) that dataset has more short samples, which have less internal context and therefore may benefit more from file context.

Finally, we observe that the overall scores are not as high as other papers report (Li et al. 2022; Wei et al. 2019). We attribute this observation to our use of the split-by-project dataset design and duplicate removal techniques, which are recommended procedures from related work (Allamanis 2019; LeClair and McMillan 2019), that are unfortunately not used in many papers. The results we report are internally comparable but not comparable against those in other papers.

6.2 RQ2: effects of file context

We report metric scores at different levels of *wo* in Tables 4 and 5. We make a few key observations. First, for the Java datasets, we observe a decline in performance as *wo* increases across all baselines and all metrics. We attribute this decline to the difficulty of generating summaries which include ever more information from outside the method being described (see Sect. 2). This decline is prominent in `funcom-java-long`, where METEOR scores when *wo* >= 4 tend to be around half

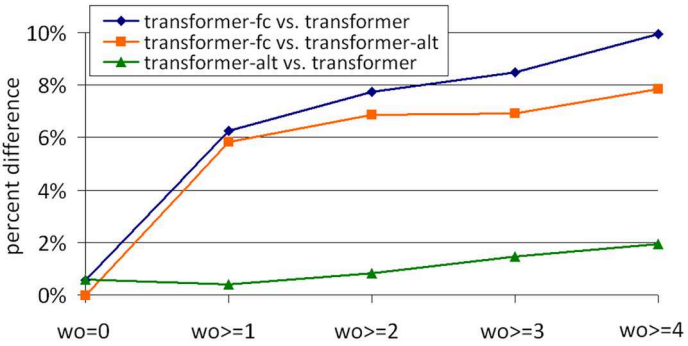


Fig. 3 Visual depiction of data for METEOR from Table 4(a). The file context model performance delta increases above 5% as *wo* increases, but the delta of the non-file context model does not

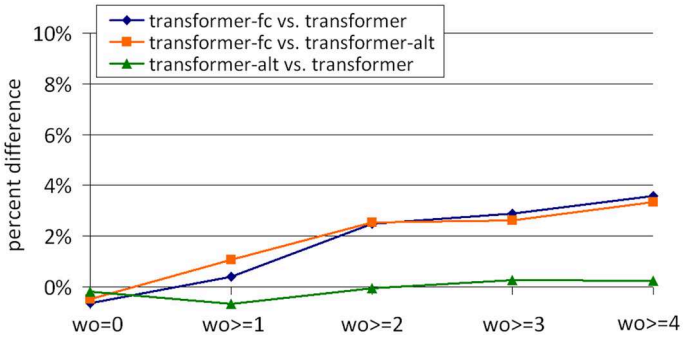


Fig. 4 Visual depiction of data for USE from Table 4(a). The file context model performance delta decreases when *wo*=0, but increases when $wo \geq 1$. File context is a key factor in overall model improvement

compared to $wo = 0$. However, the decline is not consistent in the Python dataset. METEOR and USE scores decline for `ast-attendgru`, `codegnngru`, and `transformer-alt`, but increase for `ast-attendgru-fc` and `transformer-fc`. These results may be expected because the two file context models may improve using file context.

However, `transformer` and `setransformer` also rise from $wo = 0$ to $wo >= 1$. One likely explanation is that Python contains fewer functions with reference explanations where file context is present: while in Java, around 65% of the methods have $wo >= 1$, in Python only around 20% do. The number of functions where $wo >= 2$ is only 4% of the dataset, compared to around 12% in Java. The number of functions in Python where $wo >= 3$ is less than 0.5% – only 53 functions in the test set, so low that we do not report metric scores due to possible unreliability. It is likely that factors other than file context are more important to model performance in Python, perhaps causing underperformance when $wo = 0$. In Java, we

Table 4 Raw METEOR and USE scores for different values of *wo*

Model	METEOR					USE				
	wo=0	wo≥1	wo≥2	wo≥3	wo≥4	wo=0	wo≥1	wo≥2	wo≥3	wo≥4
(a) `funcom-java` dataset										
ast-attendgru	39.25	33.24	30.67	30.14	25.06	56.30	52.01	49.27	48.69	44.34
ast-attendgru-fc	38.90	34.05	31.78	31.32	26.25	55.74	52.45	49.83	49.18	44.95
codegnngru	40.18	35.55	31.02	30.54	25.50	56.99	52.36	49.82	49.18	44.97
transformer	40.15	33.35	30.67	29.96	24.73	57.45	53.22	50.36	49.61	45.38
setransformer	39.94	33.97	31.42	30.77	25.83	56.47	52.65	50.17	49.56	45.52
transformer-alt	40.38	33.48	30.92	30.40	25.21	57.35	52.87	50.33	49.74	45.49
transformer-fc	40.37	35.43	33.04	32.50	27.19	57.88	53.44	51.61	51.04	47.01
(b) `funcom-java-long` dataset										
ast-attendgru	40.02	29.63	28.21	27.49	20.50	55.42	47.33	45.73	44.76	39.25
ast-attendgru-fc	40.33	29.94	28.46	28.11	21.45	55.79	47.70	46.15	45.39	39.68
codegnngru	39.85	29.38	27.88	27.14	20.47	55.13	47.08	45.46	44.48	38.78
transformer	39.98	29.62	28.24	27.70	20.95	56.62	48.46	46.89	45.94	40.48
setransformer	39.39	28.84	27.63	26.90	20.18	54.90	46.81	45.26	44.42	39.07
transformer-alt	41.02	30.29	28.96	28.33	21.21	57.73	49.93	48.28	47.28	41.83
transformer-fc	41.71	30.98	29.70	29.08	21.88	57.88	50.09	48.84	47.85	42.38
(c) `funcom-python` dataset										
ast-attendgru	26.74	27.06	25.33	–	–	43.58	44.44	42.08	–	–
ast-attendgru-fc	27.61	28.15	27.98	–	–	44.79	45.52	45.39	–	–
codegnngru	26.20	25.78	24.57	–	–	42.33	42.49	41.85	–	–
transformer	26.61	27.25	27.36	–	–	43.62	44.82	44.14	–	–
setransformer	27.17	28.06	27.80	–	–	43.36	45.08	44.01	–	–
transformer-alt	28.50	28.34	28.02	–	–	45.58	45.90	45.16	–	–
transformer-fc	28.49	28.92	28.81	–	–	45.22	46.37	44.66	–	–

We omit BLEU for brevity because it is less favored than other metrics (see Sect. 5.6). Note overall diminished performance for higher thresholds of *wo*.The term *wo* means the number of words that are in both the reference summary of a method and its file context, but not in the method itself (see Sect. 2)

note relatively high scores when $wo = 0$, likely due to many "easy" summaries such as "records a music file" for a subroutine `recordMusicFile()`.

A second observation is that the models which use file context tend to outperform models without it, and the delta between these models tends to increase as the threshold for *wo* increases. For example, we are able to replicate the result of Haque et al. Haque et al. (2020) in showing that `ast-attendgru-fc` improves over `ast-attendgru`. But we especially note that our model `transformer-fc` improves over `transformer` and `transformer-alt`. We present the data from Table 4 in a graphical format in Figs. 3 and 4. These show that the differences in METEOR and USE scores between `transformer-fc` and `transformer` rise above 5% when $wo >= 1$ in the `funcom-java` dataset. So even though the overall performance improvement from our model is around 3%, we note that the

Table 5 Difference between a given model and a comparison model for each dataset

	Model	Comparison	All (%)	wo=0 (%)	wo≥1 (%)	wo≥2 (%)	wo≥3 (%)	wo≥4 (%)
(a) `funcom-java` dataset								
M	transformer-fc	transformer	4.04	0.55	6.24	7.73	8.48	9.95
	transformer-fc	transformer-alt	3.57	−0.02	5.82	6.86	6.91	7.85
	transformer-alt	transformer	0.45	0.57	0.39	0.82	1.47	1.94
U	transformer-fc	transformer	1.07	−0.64	0.41	2.48	2.88	3.59
	transformer-fc	transformer-alt	1.17	−0.47	1.08	2.54	2.61	3.34
	transformer-alt	transformer	−0.09	−0.17	−0.66	−0.06	0.26	0.24
B	transformer-fc	transformer	10.33	2.53	15.14	16.57	16.86	27.90
	transformer-fc	transformer-alt	8.85	1.48	13.31	13.73	13.28	18.85
	transformer-alt	transformer	1.37	1.03	1.61	2.50	3.17	7.62
(b) `funcom-java-long` dataset								
M	transformer-fc	transformer	4.49	4.33	4.59	5.17	4.98	4.44
	transformer-fc	transformer-alt	2.03	1.68	2.28	2.56	2.65	3.16
	transformer-alt	transformer	2.41	2.60	2.26	2.55	2.27	1.24
U	transformer-fc	transformer	4.12	4.01	4.16	4.75	4.50	4.02
	transformer-fc	transformer-alt	0.29	0.26	0.32	1.16	1.21	1.31
	transformer-alt	transformer	2.21	2.22	2.21	2.56	2.30	1.33
B	transformer-fc	transformer	7.45	2.95	9.80	9.22	9.24	13.91
	transformer-fc	transformer-alt	1.17	−2.69	3.23	3.40	3.70	9.98
	transformer-alt	transformer	6.21	5.80	6.36	5.64	5.34	3.57
(c) `funcom-python` dataset								
M	transformer-fc	transformer	3.03	8.57	6.13	5.30	–	–
	transformer-fc	transformer-alt	0.39	1.37	2.05	2.82	–	–
	transformer-alt	transformer	2.63	7.10	4.00	2.41	–	–
U	transformer-fc	transformer	3.63	3.67	3.46	1.18	–	–
	transformer-fc	transformer-alt	−0.42	−0.79	1.02	−1.11	–	–
	transformer-alt	transformer	4.06	4.49	2.41	2.31	–	–
B	transformer-fc	transformer	16.14	17.22	12.88	3.97	–	–
	transformer-fc	transformer-alt	3.58	2.15	8.94	13.31	–	–
	transformer-alt	transformer	12.12	14.75	3.61	−8.24	–	–

Rows indicated for M=METEOR, U=USE, B=BLEU. For example, the BLEU score for `transformer-fc` is 27.90% higher than `transformer` when *wo* >= 4 for the `funcom-java` dataset. We do not report *wo* >= 3 for Python because the number of test samples is very small at those levels (<60)

Table 6 Training time of `transformer-fc`, `transformer-alt`, and `transformer` in different datasets in minutes

model	`funcom-java`	`funcom-java-long`	`funcom-python`
transformer	90	10	15
transformer-alt	135	15	20
transformer-fc	635	60	90

Table 7 Memory usage of `transformer-fc`, `transformer-alt`, and `transformer` during training in different datasets in GB

model	funcom-java	funcom-java-long	funcom-python
transformer	3	3	3
transformer-alt	3	3	3
transformer-fc	9	9	9

Table 8 Metric scores over the three datasets comparing different model designs for incorporating file context

Model	METEOR	USE	BLEU
(a) `funcom-java` dataset.			
ast-attendgru-fc	35.71	52.94	18.94
transformer-comb	26.07	40.68	12.67
transformer-fc (ours)	**37.12**	**54.61**	**20.18**
(b) `funcom-java-long` dataset.			
ast-attendgru-fc	33.52	50.48	18.91
transformer-comb	26.24	41.18	14.09
llama-lora	20.37	38.63	6.99
transformer-fc (ours)	**34.67**	**52.77**	**19.90**
(c) `funcom-python` dataset.			
ast-attendgru-fc	27.72	44.93	16.82
transformer-comb	19.86	34.93	11.15
transformer-fc (ours)	**28.58**	**45.45**	**18.21**

The best results across all models in that bin are given in bold

improvement is concentrated among a small set of especially challenging summaries that primarily benefit from file context. Although *wo* >= 4 only accounts for tiny part of the dataset, this part of the dataset is particularly difficult for the problem. The improvement over this part of the dataset futher shows the effectiveness of our method.

An alternative interpretation is that the delta only seems larger because the baseline scores are lower as the threshold of *wo* increases—a 1 METEOR point improvement is 3.3% of 30 but 5% of 20. However, consider the `transformer-alt` scores compared to `transformer` (the green lines in Figs. 3 and 4). METEOR scores do improve between zero and two percent for METEOR, but are essentially flat for USE. The `transformer-alt` model does not include file context but does have architectural differences over `transformer`. The improvements from the scale of `transformer-alt` are spread across all levels of *wo*. Therefore, the evidence suggests that `transformer-fc` improves due to file context when *wo* >= 1, and not due to architectural differences or mathematical illusions.

In Table 6 and 7, we reported the training time and the memory usage of `transformer`, `transformer-fc`, and `transformer-alt`. Although `transformer-fc` shows the file contexts help to improve the performance, the computational time and the memory usage increase. This is because of the extra

computation to process the file context data in our models. Overall, we observed that the file contexts improve performance, but requiring more training time and computational resources.

6.3 RQ3: alternate approaches to model file context

In Table 8, we report the metric scores for `transformer-comb` and `llama-lora`, compared against our approach and a previous GRU based file context baseline `ast-attendgru-fc`. We observe that `transformer-comb` achieves scores 23-30% lower than our approach as well as `ast-attendgru-fc` for all three datasets. We posit that combining both method and file context into a single input may not be providing the model with enough information to learn how to place the method in the file context. Therefore, these scores suggest that our model design is better suited for the task of using file context to improve code summarization, when compared to a single input Transformer that is provided with the file context simply appended to the source code.

Recall, we only test `llama-lora` on `funcom-java-long` and `funcom-python` due to high estimated training and inference time over the larger dataset. We observe that for `funcom-java-long`, `llama-lora` achieves scores 40–65% lower than our approach as well as `ast-attendgru-fc`. We do not report the scores for `funcom-python` because the scores were less than 1 point for each metric, which means the model is obviously not learning anything. Upon manual inspection we found that the model was prone to predicting code from the target function as the "response". It appears the model learned to simply fetch code from the function and file context to reduce training loss. We think that the poor performance of `llama-lora` is because it is originally pre-trained primarily on conversational English data (Touvron et al. 2023). Now it is true that recent work such as the short study by Ahmed and Devanbu (2022) shows promise using LLMs and few-shot learning for code summarization. However, it may simply be that the data we used to fine-tune the model is not enough for the model to re-adjust the learned conversational word embeddings in favor of programming-language specific word associations.

Given ever-increasing model and prompt length size, it may seem like the "obvious" solution is to simply include the entire file context with the target function. However, we find that that solution is not effective off-the-shelf. We posit that careful model design and improvements are required, when using decoder-only LLMs to learn from file context for source code summarization. In our view, a likely solution is a novel neural architecture, like `transformer-fc` that we propose. Additionally, our model can be scaled up to an arbitrary number of layers and attention heads by adjusting our model parameters L and h (see Sect. 4), just like the original Transformer architecture.

7 Human study

In this section we describe parameters and results of our human study. This study adds a qualitative evaluation to complement our quantitative evaluation in the last section.

7.1 Research questions

To design our human study and evaluation, we asked two additional research questions:

RQ4 When comparing summaries generated by `transformer-fc` and `transformer-alt`, which ones do programmers give higher rates in terms of accuracy, conciseness, completeness, and similarity to reference?

RQ5 When comparing summaries generated by `transformer-fc` and `transformer-alt`, which ones do programmers prefer in terms of overall preference?

The rationale behind RQ4 is to compare our approach against the best performing baseline, in terms of the most important qualities of a summary from related work Sridhara et al. (2010); Bansal et al. (2023a). These qualities are accuracy, conciseness, completeness, and similarity to reference. Although automated metrics are the standard for evaluation in related work, programmer opinion is important as an

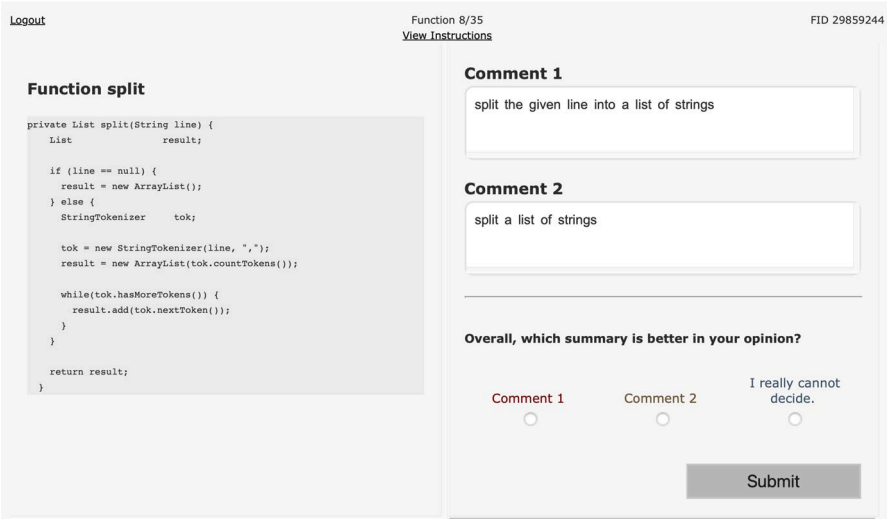


Fig. 5 A screenshot of our human study interface

indicator of qualities of a summary which automated metrics may not necessarily represent (Haque et al. 2022).

The rationale behind RQ5 is that programmers may prefer one summary over the other for reasons not formalized by RQ4. Although the qualities we ask subjects to rate in RQ4 are extensively used in related work, there may be other qualities that programmers prefer in summaries.

7.2 Interface

For our human study, we designed a web interface, a screenshot of which is in Fig. 5. The interface showed Java source code on the left and two summaries on the top right, comment 1 on top and comment 2 under it. To prevent demand characteristic bias (Dell et al. 2012), we do not reveal the source of the comments evaluated. Source of comments 1 and 2 were randomly selected and anonymized for each method and participant. For example, for some methods comment 1 is from our approach `transformer-fc` and comment 2 from the baseline transformer-alt, while it is the opposite for other methods seen by the same participant. For each method, participants were asked 5 questions on the bottom right:

Q1 Independent of other factors, which summary is more accurate?

Q2 Which summary is missing more information that is important for understanding the method?

Q3 Which summary contains more unnecessary information?

Q4 Overall, which summary is better in your opinion?

Q5 Which summary is more similar to this third summary on the left?

For Q5, the participants are shown the reference summary on the bottom left below the code. For each question the participants are presented with three choices: (1) "comment 1", (2) "comment 2", and (3) "I really cannot decide".

7.3 Dataset

The dataset we use for our human study consists of summaries generated for 35 Java methods from the test set of `funcom-java`. We used `funcom-java` as the dataset for human study because this dataset has the largest improvement in terms of automatic metrics. Roy et al. (2021) found that human programmers may not be able to differentiate two summaries when the improvement is less than two points and the results are mixed i.e. mixing the results of Java with Python. We select a small subset for human evaluation, because while the automated metrics in Sect. 6 are computed over large test sets, human studies are time-restrictive. Extended studies

can lead to fatigue bias, a decrease in quality, and reliability of the data (Jeong et al. 2023).

To select these summaries, we filtered the test set for methods where the predicted summaries from `transformer-fc` and `transformer-alt` differ by at least 2 words. Then, we picked 35 random methods. We restrict the dataset to 35 methods to keep the study duration to around 1 h, to prevent fatigue bias. The average evaluation time reported by similar studies is 1.5 min/method (Bansal et al. 2023a).

7.4 Participants

We recruited 15 Java programmers using Prolific, a web service that facilitates screening and recruitment of research study participants from the UK and USA. We compensated each participant at a flat rate of $20 for roughly a one hour session.

7.5 Threats to validity

Like any human study, the biggest threats to validity are from participant exhaustion or bias and data selection. To mitigate the threat of participant exhaustion we restrict the time of our study to around 1 h as recommended in related work (Sievertsen et al. 2016). To mitigate the threat of participant bias, we designed our interface as a blind test, without revealing the source of comments. We randomly generate the order of samples shown, which is different for each of the 15 participants. We also analyze their answers in a post-processing step to look for suspicious patterns such as same option for successive choices or identical choices between participants. We did not find any samples that exhibit these patterns. To mitigate the threat of data

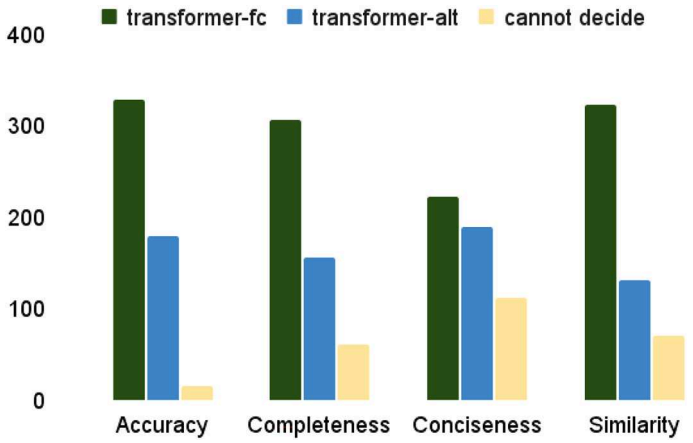


Fig. 6 Qualitative comparison of `transformer-fc` and `transformer-alt`. The participants were also given a third option–cannot decide

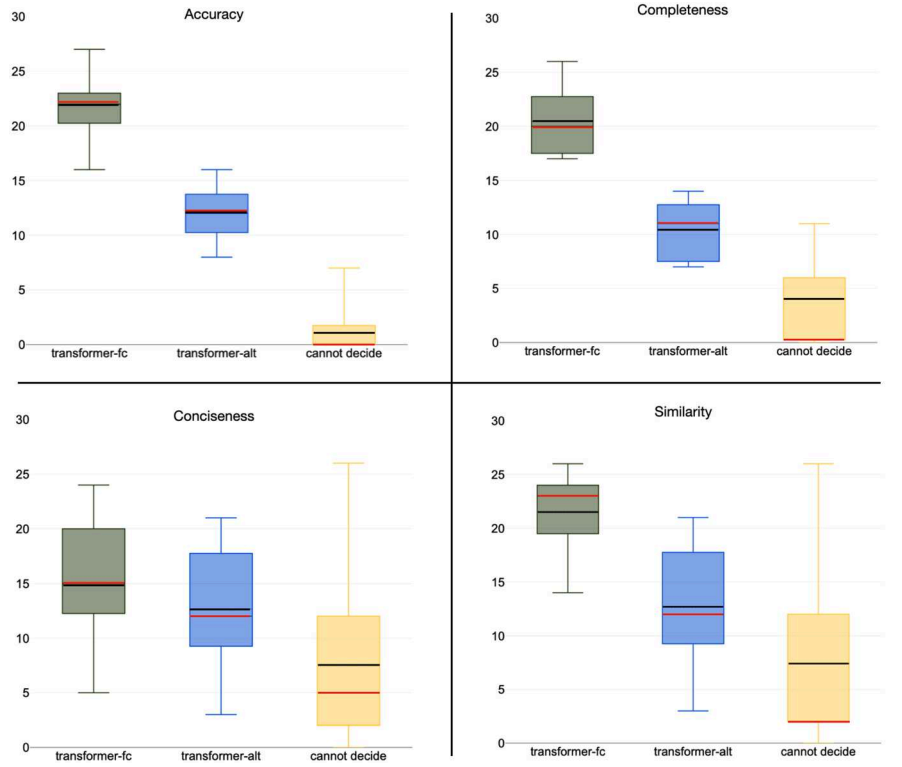


Fig. 7 This Box-plot shows distribution for accuracy, completeness, conciseness, and similarity (to reference). The whiskers indicate maximum and minimum values across all 15 participants. The red line in each box indicates median and the black line indicates mean

selection, we select the data with the highest improvement in automatic metrics, so the participants can see the difference as suggested by Roy et al. (2021).

8 Human study results

In this section we report and discuss the results of the two additional RQs we asked for the human study.

8.1 RQ4: qualitative comparison

In Fig. 6, we report the distribution of all human ratings. The total number of human ratings is 525, i.e., all 35 methods rated by each of the 15 participants. Note that we phrased completeness and conciseness questions negatively in the interface. During post-processing, we flip those ratings (except where participants could not decide) to

obtain positive scores to compare with other qualities. Additionally we present boxplots showing the distribution of these ratings in Fig. 7.

In terms of accuracy, we found that 62% of individual ratings picked summaries generated by our approach `transformer-fc` as more accurate than the baseline `transformer-alt`. In comparison, only 32% of the individual ratings picked the baseline as more accurate. This re-affirms our hypothesis that file context helps most for a subset of cases, while overall metric scores might be affected by some cases where it may not improve the summary. In Fig. 7, we observe a small standard deviation indicating general consensus between participants, with high median value of 22 samples out of 35 for `transformer-fc`. Overall, we observe that `transformer-fc` generates more accurate summaries than `transformer-alt` for majority of samples.

In terms of completeness, we see similar trends as accuracy, where 58% of all ratings indicated that our approach generated more complete summaries. In Fig. 7 we observe that each participant indicated that `transformer-fc` generated summaries were more complete for at least 17 of the 35 samples, with a median of 20 samples. These values are also seen to be higher than the maximum values for `transformer-alt`, where each participant found summaries generated by the baseline to be better in 14 or less samples, with a median of 11. Overall, we find that for a majority of samples, participants favored summaries generated by our approach with file context.

In terms of conciseness, we observe closer aggregate scores of 43% in favor of `transformer-fc`, 37% in favor of `transformer-alt`, and 20% could not decide. A possible reason for this is that our summaries are limited to 13 words. We posed this question negatively in the study, asking which summary had more unnecessary information. Due to the short length of our summaries, it may have been harder for participants to decide which information was unnecessary. In Fig. 7, we

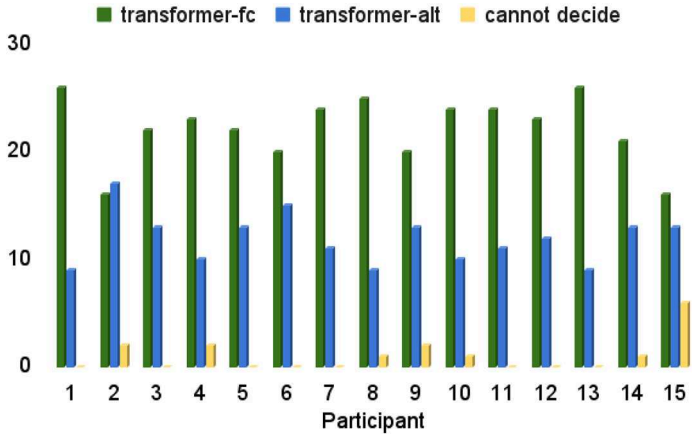


Fig. 8 Overall preference of each participant when presented with summaries generated by `transformer-fc` and `transformer-alt`. The participants were unaware of the source of the summaries. The y-axis denotes number of samples marked each answer and x-axis denotes the participants

observe a lot of overlap in the distributions. Overall, `transformer-fc` achieves a higher median and mean when compared to `transformer-alt`, albeit with a smaller margin than other qualities.

In terms of similarity to reference, we observed that the participants found it difficult to decide similarity to reference for 13% of all samples. One possibility is that even though one of the generated summaries was more accurate, the reference summary may be completely different, such as generic summaries from Javadocs. In Fig. 7 we observe that the minimum number of methods in favor of our approach is higher than the mean and median values for the distribution of the baseline. Also, the median value for our approach is higher than the maximum value for the baseline. Overall, a majority of participants found `transformer-fc` generated summaries to be more similar to reference by a considerable margin.

In short, we observe that subjects of our human study found summaries generated by `transformer-fc` to be more accurate, complete, and similar to reference when compared to transformer-alt. For conciseness, the results are not as clear, which maybe attributed to the fact that we limit our summaries to a maximum of 13 words.

8.2 RQ5: overall preference

In Fig. 8 we report the distribution of overall preference for each of the 15 participants. We observe that 13 out of 15 participants found summaries generated by `transformer-fc` to be better overall for a majority of the samples (50% or more). For participant numbers 2 and 15, our approach did not reach the majority threshold of 18 samples, but neither did the baseline. A few outliers are expected in human studies such as participant 2, but a vast majority of participants favored summaries generated by `transformer-fc` when compared with summaries generated by `transformer-alt`.

We also performed a Mann–Whitney U test to measure the statistical significance of this difference in our distribution of participants. We computed values of $U1 = 223$ and $p - val = 4.6e^{-6}$. As $p <<< 0.05$, we reject the null hypothesis and find that the difference is statistically significant. Overall, participants preferred summaries generated by our approach `transformer-fc`, with a statistically significant margin, when compared to summaries generated by the best performing baseline of the same model size without file context, `transformer-alt`.

9 Conclusion

This paper advances the state of the art in four ways: First, we present a neural model for source code summarization that augments a standard Transformer encoder/decoder architecture to accept file context. We propose a novel architecture as an alternative to the popular practice of using large context window that rely on model size alone. We evaluate our model against several baselines over three datasets in

two programming languages. We show that our model outperforms these baselines under our experimental conditions according to three metrics from related work.

Second, we demonstrate that file context is a key factor in source code summarization and the improvements gained by our model. We report model performance at different levels of the overlap between file context and the summaries, for words not appearing in the code being summarized (we denote this value *wo* according to the formula in Sect. 2). We find that in the Java datasets, a marked decrease in performance occurs as thresholds of *wo* increases. The relationship is less clear in Python, though we still note generally increasing improvement between our approach and the baselines for METEOR and BLEU.

Third, we we demonstrate that our model design is well-suited to use the file context. We directly compare our model design against the aforementioned large context window approaches. We evaluate two such alternate approaches. One is a single encoder and decoder Transformer-based network. The other is a decoder-only LLM fine-tuned over one of our datasets. We find that these off-the-shelf approaches that simply combines target source code and file context into a giant context window perform considerably worse than our design.

Fourth, we conduct a human study to add a qualitative aspect to our evaluation. We find that when presented by two different summaries for the same method, a majority of participants favored summaries generated by our approach compared to the best performing baseline. The participants found summaries generated with file context to be more accurate, complete, similar to reference, and better overall in their opinion. We note that we did not see clear consensus on whether our approach generates more concise summaries than the baseline.

Reproducibility To ensure maximum reproducibility of the results, we release all datasets in Data Availability Section. Also, we provide an online reproducibility guide with step-by-step instructions showing how we produced the our results and source code in Code Availability Section.

Acknowledgements This work is supported in part by the NSF CCF-2100035 and CCF-2211428. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Author contributions C.S., A.B., and C.M. all worked on the paper together. All authors reviewed the manuscript.

Data availability We released the datasets that we created to APCL Hugginface repository, https://huggingface.co/datasets/apcl/funcom-python

Code availability We release our code for experiments in our APCL Github repository, https://github.com/apcl-research/TransformerFC

Declarations

Competing interests The authors declare no competing interests

References

Ahmad, W., Chakraborty, S., Ray, B., et al.: A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 4998–5007 (2020)

Ahmad, W.U., Chakraborty, S., Ray, B., et al.: Unified Pre-training for Program Understanding and Generation (2021). arXiv preprint arXiv:2103.06333

Ahmed, T., Devanbu, P.: Few-shot training LLMs for project-specific code-summarization (2022). arXiv preprint arXiv:2207.04237

Allamanis, M.: 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)

Alon, U., Zilberstein, M., Levy, O., et al.: code2vec: learning distributed representations of code. In: Proceedings of the ACM on Programming Languages 3(POPL):1–29 (2019)

Banerjee, S., Lavie, A.: 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, pp. 65–72 (2005)

Bansal, A., Haque, S., McMillan, C.: 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)

Bansal, A., Eberhart, Z., Karas, Z., et al.: Function call graph context encoding for neural source code summarization. IEEE Trans. Softw. Eng. (2023a). https://doi.org/10.1109/TSE.2023.3279774

Bansal, A., Sharif, B., McMillan, C.: Towards modeling human attention from eye movements for neutral source code summarization. iN: Proceedings of ACM Human–Computer Interaction, vol. 7 (2023b)

Chiang, D., Rush, A.M., Barak, B.: Named Tensor Notation (2021). arXiv preprint arXiv:2102.13196

Dell, N., Vaidyanathan, V., Medhi, I., et al.: "yours is Better!" Participant Response Bias in HCI. In: Proceedings of the Sigchi Conference on Human Factors in Computing Systems, pp. 1321–1330 (2012)

Ding, Y., Wang, Z., Ahmad, W.U., et al.: CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context (2022). arXiv preprint arXiv:2212.10007

Feng, Z., Guo, D., Tang, D., et al.: CodeBERT: A Pre-trained Model for Programming and Natural Languages (2020). arXiv preprint arXiv:2002.08155

Guerrouj, L., Di Penta, M., Guéhéneuc, Y.G., et al.: An experimental investigation on the effects of context on source code identifiers splitting and expansion. Empir. Softw. Eng. **19**, 1706–1753 (2014)

Haldar, R., Wu, L., Xiong, J., et al.: A Multi-perspective Architecture for Semantic Code Search (2020). arXiv preprint arXiv:2005.06980

Haque, S., LeClair, A., Wu, L., et al.: Improved automatic summarization of subroutines via attention to file context. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 300–310 (2020)

Haque, S., Bansal, A., Wu, L., et al.: Action word prediction for neural source code summarization. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 330–341. IEEE (2021)

Haque, S., Eberhart, Z., Bansal, A., et al.: Semantic similarity metrics for evaluating source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 36–47 (2022)

Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically capturing source code context of NL-queries for software maintenance and reuse. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 232–242. IEEE (2009)

Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Proceedings of the 27th International Conference on Software Engineering, pp. 117–125 (2005)

Hu, E.J., Shen, Y., Wallis, P., et al.: Lora: Low-Rank Adaptation of Large Language Models (2021). arXiv preprint arXiv:2106.09685

Hu, X., Li, G., Xia, X., et al.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210. ACM (2018a)

Hu, X., Li, G., Xia, X., et al.: Summarizing source code with transferred API knowledge. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 2269–2275. AAAI Press (2018b)

Huang, Z., Liang, D., Xu, P., et al.: Improve transformer models with better relative position embeddings. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 3327–3335 (2020)

Jeong, D., Aggarwal, S., Robinson, J., et al.: Exhaustive or exhausting? Evidence on respondent fatigue in long surveys. J. Dev. Econ. **161**, 102992 (2023)

Kramer, D.: API documentation from source code comments: a case study of Javadoc. In: Proceedings of the 17th Annual International Conference on Computer Documentation, pp. 147–153 (1999)

Kuang, L., Zhou, C., Yang, X.: Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. Autom. Softw. Eng. **29**(2), 43 (2022)

LeClair, A., McMillan, C.: Recommendations for datasets for source code summarization. In: Proceedings of NAACL-HLT, pp. 3931–3937 (2019)

LeClair, A., Jiang, S., McMillan, C.: 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)

LeClair, A., Haque, S., Wu, L., et al.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)

Li, Y., Wang, S., Nguyen, T.N.: A context-based automated approach for method name consistency checking and suggestion. In: Proceedings of the 43rd International Conference on Software Engineering. IEEE Press, ICSE '21, pp. 574–586 (2021). https://doi.org/10.1109/ICSE43902.2021.00060

Li, Z., Wu, Y., Peng, B., et al.: SeTransformer: A transformer-based code semantic parser for code comment generation. IEEE Trans. Reliab. **72**, 258–273 (2022)

Liang, Y., Zhu, K.Q.: Automatic generation of text descriptive comments for code blocks. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)

Liu, S., Chen, Y., Xie, X., et al.: Retrieval-augmented generation for code summarization via hybrid GNN. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=zv-typ1gPxA

Nie, P., Rai, R., Li, J.J., et al.: 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, pp. 385–396. ACM (2019)

Roehm, T., Tiarks, R., Koschke, R., et al.: How do professional developers comprehend software? In: 2012 34th International Conference on Software Engineering (ICSE), pp. 255–265. IEEE (2012)

Roy, D., Fakhoury, S., Arnaoudova, V.: 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)

Shi, L., Mu, F., Chen, X., et al.: 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)

Sievertsen, H.H., Gino, F., Piovesan, M.: Cognitive fatigue influences students' performance on standardized tests. Proc. Natl. Acad. Sci. **113**(10), 2621–2624 (2016). https://doi.org/10.1073/pnas.1516947113

Sridhara, G., Hill, E., Muppaneni, D., et al.: Towards automatically generating summary comments for java methods. In: Proceedings of the 25th IEEE/ACM international conference on Automated software engineering, pp. 43–52 (2010)

Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. Adv. Neural Inf. Process. Syst. **27**, 3104–3112 (2014)

Tang, Z., Shen, X., Li, C., et al.: AST-trans: Code summarization with efficient tree-structured attention. In: Proceedings of the 44th International Conference on Software Engineering, pp. 150–162 (2022)

Touvron, H., Lavril, T., Izacard, G., et al.: LLaMA: Open and Efficient Foundation Language Models (2023). arXiv preprint arXiv:2302.13971

Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30**, 6000–6010 (2017)

Wan, Y., Zhao, Z., Yang, M., et al.: Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 397–407. ACM (2018)

Wang, S., Wen, M., Lin, B., et al.: Lightweight global and local contexts guided method name recommendation with prior knowledge. In: Proceedings of the 29th ACM Joint Meeting on European

Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2021, pp. 741–753 (2021). https://doi.org/10.1145/3468264.3468567

Wei, B., Li, G., Xia, X., et al.: Code generation as a dual task of code summarization. Adv. Neural Inf. Process. Syst. **32**, 6563–6573 (2019)

Wei, B., Li, Y., Li, G., et al.: 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)

Zügner, D., Kirschstein, T., Catasta, M., et al.: Language-agnostic representation learning of source code from structure and context. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=Xh5eMZVONGF