Analysis of automated code comment generation with zero-shot learning, in-context learning and prompt tuning on LLMs

1st Tanmay Vakare
Master's in Computer Science
University of Texas at Dallas
thv200000@utdallas.edu

2nd Harichandana Neralla
Master's in Computer Science
University of Texas at Dallas
hxn210036@utdallas.edu

***Abstract*—A code comment is a programmer-readable explanation or annotation in the source code of a computer program. They are added to make the source code easier for humans to understand and are generally ignored by compilers and interpreters. This project aims to investigate and enhance the automated code comment generation process using Large Language Models (LLMs). Leveraging techniques such as zero-shot learning, in-context learning, and prompt tuning, the project seeks to improve the quality and relevance of generated code comments. Through comprehensive experimentation and evaluation, the research aims to provide valuable insights into the capabilities of LLMs in the context of software development, ultimately contributing to more efficient and effective code formatting practices. Source Code [1]**

***Index Terms*—prompt tuning, zero-shot, code comment generation, in-context**

I. Introduction

It's common for software developers to try to understand the purpose and usage of a particular code unit. They can accomplish this by reading the source code documentation. Well-written documentation is essential for efficient software development, especially in situations where the code base's subject matter expert is unavailable or nonexistent [1], [3].

Nevertheless, the creation and maintenance of such documentation is expensive and time-consuming. Since they believe it to be less rewarding and productive, most developers frequently express their reluctance to write documentation. Manual documentation consequently becomes problematic or useless. Additionally, as code is updated or modified continuously, documentation eventually becomes outdated. Missing, inadequate, or mismatched comments can result from a lack of professional knowledge or ignorance of the significance of comments, which severely impacts software understanding, reuse, and maintenance [1], [3], [26].

The automatic code comment generation task involves generating a brief natural language description for a given piece of code [1]. Therefore, the availability of robust automatic comment-generation tools is essential to elevate the overall quality of software products[1]. However, several challenges and issues persist in this domain, necessitating research and innovation:

- **Quality and Consistency**: Existing automated code comment generation tools often struggle to consistently produce high-quality comments. Comments may vary in clarity, relevance, and correctness, leading to inconsistencies in-code documentation.
- **Code Complexity**: Complex code structures, nested functions, and intricate algorithms pose a significant challenge for automated comment generation. Current systems may fail to capture the intricacies of such code, resulting in inadequate or misleading comments.
- **Contextual Understanding**: Automated tools cannot frequently comprehend the context in which the code operates fully. This limitation hampers their capacity to generate comments that accurately reflect the code's purpose and functionality.
- **Codebase Variability**: Software projects frequently involve multiple programming languages, frameworks, and libraries. Automated comment generation systems must adapt to diverse codebases, making it challenging to ensure universal applicability.
- **Informativeness vs. Conciseness**: Striking the right balance between providing informative comments and maintaining code readability is a persistent challenge. Automated systems often produce comments that are either overly verbose and complex or overly simplistic and uninformative.

Addressing these challenges is crucial to improving the effectiveness and adoption of automated code comment generation systems. Solutions that enhance code documentation while accommodating the complexities of real-world software development scenarios are essential for advancing this field.

A. Motivation

The rapid advancement of Large Language Models (LLMs) has transformed various natural language processing tasks. They are characterized by their massive scale and size, often containing hundreds of millions to billions of parameters (learnable weights). The scale of these models allows them to capture complex patterns and nuances in language. LLMs are typically pre-trained on large and diverse text corpora, such

[1]https://github.com/Tanmay06/automated_code_comment

as the entire internet. While conventionally all supervised deep learning(DL) based systems relied on large amounts of data to learn patterns, currently significant research is being done to efficiently leverage LLMs capabilities for unseen downstream tasks.

¡Add why llms are good for a generation.¿ The application of LLMs to the comment generation task is motivated by two factors. Firstly, LLMs designed for the code domain are typically pre-trained using code and its associated pairwise comments to establish semantic connections between programming language and natural language [**?**]. For example, the commonly used pre-training task, masked language modeling [14], is specifically intended to align programming language and natural language representations. Secondly, existing research has shown that code comments from real-world projects, which form the training corpus for LLMs, often contain multiple syntactic and semantic meanings and intents [29]. As a result, during pre-training, LLMs are trained to understand code from various perspectives, potentially allowing them to capture different code semantics. Thus, by fully exploiting the capabilities of pre-trained LLMs, we can achieve good performances on the code comment generation task.

In software development, generating accurate and informative code comments is crucial for enhancing code comprehensibility and maintainability. We propose a comprehensive analysis of automated code comment generation techniques leveraging the capabilities of LLMs, with zero-shot learning, in-context learning, and prompt tuning methodologies. The capabilities of LLMs will enable it to capture complex patterns in the code and generate comprehensive comments. The methodologies mentioned before will help to efficiently leverage the LLMs for conditional generation, without requiring large amounts of resources or data.

B. Objective

The objective of this research is to assess the capabilities of modern LLMs in the generation of code comments with limited data and without explicit fine-tuning of parameters. For this study, we employ advanced prompting techniques to guide the LLMs to the desired results. We study the generation results of the following prompting techniques:

- **Zero-Shot Learning**: Without any explicit information the model is asked to generate code comment for the given input code snippet. We assess the model's capability of understanding the input code and generating comprehensive comments given the model is not trained/finetuned on the comment generation task.
- **In-Context Learning**: Along with the task and input code snippet, the model is shown a few well-designed examples of the input with their corresponding outputs. The model looks at the examples at inference, learns the structure, and focuses on the required information. We assess the model's capability to generate the required output with given samples.
- **Prompt Tuning**: It involves learning prompts to elicit desired behaviors from LLMs. We will explore prompt-tuning strategies to fine-tune LLMs' behavior for code comment generation tasks, emphasizing the optimization of prompt design to achieve better results.

We discuss these approaches in detail in the later sections of the article.

II. Related Work

Automated code comment generation has evolved significantly, driven by advancements in machine learning and natural language processing:

- **Template-based**: Template-based uses an automatic tool to insert information according to pre-defined rules and layout. Sridhara et al. utilized natural language templates to capture the key statements from a Java method and build a method-level summary [9]. Developers can create templates that include placeholders for information such as function names, parameters, descriptions, and other relevant details. One main drawback of this approach is the inability to capture context-based information. Rastkar et al.and Moreno et al. used heuristics to extract and summarize information from source code [18], [19]. Mcburney et al. merged contextual information with method statements [22]. Abid et al. produced a summary of C++ methods by stereotyping them with their source code analysis framework (srcML) [1]. Rai et al. summarized Java code that uses code-level nano-patterns [21].
- **RNN-based/LSTM-based**: The current probabilistic model-based techniques generate comments directly from code. One such technique is by Iyer et al. [10]who proposed a completely data-driven attention-based RNN and LSTM model called Code-NN which recommends code comments for the given source code units. Allamanis et al. used an attention neural network that employs convolution to learn local, time-invariant features [23]. Barone et al. built a dataset of Python functions and their docstrings using GitHub data and employed Neural Machine Translation (NMT) to generate docstrings from given functions [24]. Wan et al. proposed a reinforcement learning framework that incorporates the abstract syntax tree (AST) along with the sequential code content [25]. Hu et al. developed DeepCom and Hybrid-Deepcom to generate code comments by learning from a large corpus and by combining lexical and structural information of code [26], [27]. Chen and Zhou designed a neural framework called BVAE to improve code retrieval and summarization tasks [28]. Several studies employed transformer-based models, e.g., Ahmad et al. used a self-attention-based transformer model [30] while Wang et al. developed a BERT-based model called Fret [13]. Feng et al. presented CodeBERT, pre-trained on 2.1M bimodal data points (codes and descriptions) and 6.4M unimodal code across six languages (Python, Java, JavaScript, PHP, Ruby, and Go) [**?**]. They evaluated CodeBERT on two downstream NL-PL tasks (code search and documentation) by fine-tuning model parameters. Gao et al. pro-

posed the concept of a Code structure-guided transformer for source code summarization that incorporates code structural properties into the transformer to improve performance [7]. Phan et al. presented CoTexT which learns the representative context between natural programming languages and performs several downstream tasks including code summarization [31]. Ahmad et al. developed PLBART, a sequence-to-sequence model, pre-trained on a large set of Java and Python functions and their textual descriptions collected from GitHub and Stack Overflow [30]. Later, Parvez et al. showed that the addition of relevant codes/summaries retrieved from a database (such as GitHub and Stack Overflow) can improve the quality of documentation produced by a generator model [2]. Though transformer-based models showed promise in documentation generation, the GPT-3 model had not been systematically evaluated for this task yet despite its success and popularity. Our study employed the GPT-3-based Codex model for documentation generation and compared its performance with existing approaches.

- **Classification-based**: Classification-based approaches have predefined comments that use a classification model to label the given input source code. In [32] proposal, they used domain-specific Supervised learning models and classifiers to classify the comment text as Useful and not useful. In [33] author works on toxic comment classification. They compare the public multi-label dataset of more than 200,000 user comments. They apply the supervised machine learning classifier to train the data and validate the results
- **Code-T5**: CodeT5, a unified pre-trained encoder-decoder Transformer model that better leverages the code semantics conveyed from the developer-assigned identifiers [11]. This model was proposed to continuously employ both code understanding and generation tasks and allows multi-task learning. As it considers token-type information in the code, it cannot precisely get the context of the code to produce accurate comments for any given code unit. As illustrated and proposed by [11] they have extended the noising Seq2Seq objective in T5 by proposing two identifier tagging and prediction tasks to enable the model to better leverage the token type information from PL, which are the identifiers assigned by developers. To improve the NL-PL alignment, they further propose a bimodal dual learning objective for a bidirectional conversion between NL and PL.

Existing work underscores the potential for innovation in leveraging LLMs and related techniques to improve code comment quality, relevance, and context awareness [15], [17].

III. Technical Approach

In our exploration of the automated code comment generation capabilities inherent in state-of-the-art Large Language Models (LLMs), we employ sophisticated prompting techniques to delve into the intricacies of these powerful LLMs. The research methodology involves extensive experimentation using diverse samples from coding languages like Python code units and SQl commands. In this section, we discuss the dataset, and LLMs used in this study, followed by technical details of the implementation.

A. Dataset

We built a subset of the Code-Search-Net¡add ref¿ dataset on *Python* for our experiments. We chose *Python* because compared to other programming languages, the *Python* code is closer to natural language making it easier for language models to comprehend. Furthermore, because of its popularity, it is easier to find diverse examples of *Python* code. Code-Search-Net dataset has *Python* function examples with its corresponding docstring, all extracted from popular GitHub repositories. However, due to its popularity and size, it is also used for pretraining the LLMs used in this study. We inspect possible biases in the model we extract a small sample set of SQL query comments from the internet. Table **??** shows the sample size and data distribution of the dataset used.

Dataset	Language	Train	Test	Validation
Code-Search-Net	Python	1000	150	150
Web Scrapped	SQL	-	-	25

TABLE I
DATASET SOURCE AND SPLITS.

B. Large Language Models

These LLMs, boasting billions of parameters, undergo a pretraining phase on vast datasets with the ability to generate comments relatable to the context of the given code. In this study, we focus on the results generated by LLaMa-2 and PaLM because of their open-source availability and performance. While we also experimented with models like FLAN-T5, BART, and OpenLLaMa-7b, we decided to skip their discussion because their results were not comparable.

- **LLaMA-2** [**?**] is a Meta fine-tuned base large language model trained on 500 billion tokens of code data, which can understand the natural language instructions. It used a specialized task called "Code Infilling" which is to fill the current cursor position, considering the contents that appear before and after it, which helps in generating the code including the comments that best match an existing prefix and suffix. We use Huggingface's implementation *meta-llama/Llama-2-7b-chat-hf*[2] with 7B parameters fine-tuned for chat like instruction based tasks. We use a local machine with 4 NVIDIA Tesla V100S GPUs each with 32GB of VRAM for running our experiments.
- **PaLM 2** [**?**] is a powerful LLM with impressive multilingual, reasoning, and coding capabilities with around 540B parameters. It can understand and generate text in over 60 languages, excels at reasoning tasks like answering open-ended questions, and even generates code in multiple programming languages. PaLM 2 is used by Bard, Gmail, and offers an API for developers to integrate

[2]https://huggingface.co/meta-llama/Llama-2-7b-chat-hf

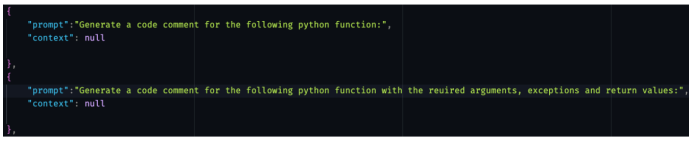

Fig. 1. Sample prompt used for Zero-shot prompting

its power into their own applications. We use PaLM for our experiments using the PaLM API[3].

C. Experimental Setup

a) ***Zero-Shot Learning(Hard Prompt)****:* In zero-shot learning the model is given a task followed by relevant information and is expected to generate desired results. We leverage the model's inherent understanding capabilities and expect that it will be able to comprehend the task. In our experiment, we designed multiple task prompts to check which gives better results, Figure 1 shows the final task prompt. The task prompt is augmented with every code snippet and used as input to the model for inference.

b) ***In-Context Learning(Few-Shot)****:* In this technique we provide additional information, as examples, to a large language model (LLM) during inference. This helps the LLM understand the structure of the task, focus on the relevant information, and generate more accurate and relevant outputs, even when it hasn't been specifically fine-tuned for the specific task at hand. By providing examples and context, the prompt guides the LLM to learn and adapt, leading to better performance on diverse tasks. Figure 2 shows a template with examples used for in-context learning, where the task is followed by examples, separated by #### and every example the input and required generation is separated by =>.

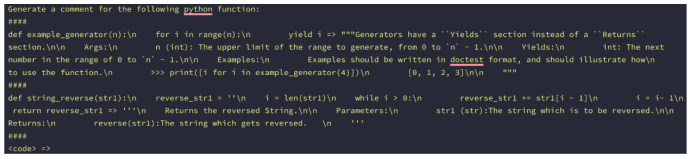

Fig. 2. Sample prompt template used for in-context learning.

c) ***Prompt Tuning(Soft Prompt)****:* Prompt tuning is an additive method for only training and updating the newly added prompt tokens to a pretrained model. This way, we can use one pretrained model whose weights are frozen, and train and update a smaller set of prompt parameters for each downstream task instead of fully finetuning a separate model. We use prompt tuning with *Parameter Efficient Fine Tuning(PEFT)*[4] on LLaMa-2. For PaLM we use the fine tune API endpoint in the PaLM API. With PEFT on Llama-2-7b-chat-hf trainable params: 32,768 $\|allparams$: $6,738,448,384\|trainable\%$: 0.0004862840543203603.

[3]https://developers.generativeai.google/products/palm
[4]https://huggingface.co/docs/peft/index

IV. Experimental Results

In this section, we comprehensively evaluate the results from experiments using three key metrics: *BertScore*, *Rouge2-F1*, and *RougeL-F1*. These metrics offer insights into the quality and similarity of the generated text compared to the reference text.

- **BertScore** is a metric that quantifies the semantic similarity between model-generated text and reference text by leveraging contextual embeddings from BERT (Bidirectional Encoder Representations from Transformers). A higher BertScore indicates a more significant similarity between the generated text and the reference, suggesting that the model produces outputs closely aligned with the expected content.
- **Rouge2-F1** is a metric that measures the overlap of bigrams (sequences of two words) between the generated and reference text. A higher Rouge2-F1 score signifies a better match in terms of bigram overlap, emphasizing the quality of generated phrases.
- **RougeL-F1** is a metric that considers the longest common subsequence (LCS) between the generated and reference text. A higher RougeL-F1 score suggests a stronger alignment in terms of the length of the shared subsequence.

Table II shows the results of the generation experiments. As expected empirically in automatic evaluations, PaLM outperforms LLaMa by a huge difference. This is due to its sheer size and training dataset. However, looking at the generated examples, LLaMa is not that far of from PaLM. We also observed that if given appropriate prompts LLaMa can generate comparable comments.

The reduction in ROUGE-2 between in-context and zero-shot is because the generations for in-context are following the structure of well-crafted examples provided in the input while most of the ground truth examples lack this structure. As ROUGE-2 captures bigram overlap between the hypothesis and reference, the generation are not able to completely match the ground truth. This evident from the examples in Table III and IV.

V. Discussions and Findings

In our experimental analysis, *PaLM* and *LLaMa-2* demonstrate impressive performance in zero-shot learning for *Python*, producing code comments that exhibit high cohesion and relevance. This proficiency in generating meaningful comments, even without specific training data, suggests a bias as CodeSearch-Net data is also used for pretraining these models. However, similar quality generations for *SQL* queries confirm that not merely rely on biases present in their training data but are capable of adapting and generating relevant comments across diverse contexts.

In-context learning emerges as a valuable technique, aiding the models in structuring their output to meet specified formats and extracting pertinent information. This capability enhances the adaptability of the models, allowing them to generate comments that align with the requirements of different contexts

Model	Zero-Shot			In-Context			Prompt Tuning		
	BertScore	**Rouge2-F1**	**RougeL-F1**	**BertScore**	**Rouge2-F1**	**RougeL-F1**	**BertScore**	**Rouge2-F1**	**RougeL-F1**
Python									
PaLM(text-bison-001)	**0.794**	**0.095**	**0.21**	**0.82**	**0.089**	**0.2**	**0.85**	**0.12**	**0.56**
Meta-LLaMa-2 7b	0.754	0.025	0.075	0.799	0.015	0.082	0.809	0.053	0.091
SQL									
PaLM(text-bison-001)	**0.878**	**0.159**	**0.429**	**0.921**	**0.131**	**0.53**	-	-	-
Meta-LLaMa-2 7b	0.689	0.057	0.082	0.767	0.043	0.101	-	-	-

TABLE II
EVALUATION SCORES FOR MODELS

Model	Ground Truth	Zero-Shot	In-Context	Prompt Tuning

PaLM	`Extracts video ID from URL.`	`Get video id from url`	`Get the video id from a YouTube URL.`
			`Args:`
			`url (str): The YouTube URL.`
			`Returns:`
			`str: The video id.`
			`Raises:`
			`ValueError: If the URL is not a valid YouTube URL.`
				`Extracts video ID from URL.`

LLaMa-2		`This function takes a URL as input and returns the video ID associated with the URL. It uses a combination of regular expressions and query parameter parsing to identify the video ID.`	`This function takes a URL as input and returns the video ID of the video`
			`embedded in the URL. It uses a combination of regular expressions and`
			`parsing of query parameters to extract the video ID.`
			`Args:`
			`url (str): The URL to extract the video ID from.`
			`Returns:`
			`str: The video ID of the video embedded in the URL, or None if no`
			`video ID could be found.`
			`Examples: …`
				`This function takes a URL as input and returns the video ID associated with the URL.`

TABLE III
EXAMPLE OF GENERATED COMMENTS FOR PYTHON FUNCTION.

and input scenarios.

Furthermore, our findings highlight the effectiveness of prompt tuning in refining the model outputs. By fine-tuning the models with specific prompts, we observe improvements in the alignment of generated code comments with the ground truth. This adjustment in the training approach underscores the significance of tailored prompts in enhancing the precision and relevance of the generated comments.

VI. CONCLUSION AND FUTURE WORKS

In conclusion, in this project, we systematically investigate different LLMs of different sizes and show that automated code generation is possible with these models. We integrate zero-shot learning, in-context learning, and prompt engineering techniques to enhance the performance of LLMs in automated code comment generation. Our study showcases the potential of LLMs in code comment generation, emphasizing the importance of thoughtful and systematic investigation of prompting techniques for achieving optimal results across different model scales. The findings from this research have provided valuable insights and methodologies that can be applied in real-world software development scenarios, ultimately leading to more efficient and effective coding practices.

REFERENCES

[1] C. Wang, H. Su, H. Gao and L. Zhang, "A Method for Automatic Code Comment Generation Based on Different Keyword Sequences," in 2023 International Conference on Frontiers of Robotics and Software Engineering (FRSE), Changsha, China, 2023 pp. 57-64. doi: 10.1109/FRSE58934.2023.00016.

[2] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. arXiv preprint arXiv:2108.11601 (2021).

[3] Khan, J. Y., and Uddin, G. (2022). Automatic Code Documentation Generation Using GPT-3. ArXiv. /abs/2209.02235

[4] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension (ICPC '18). Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[5] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. ACM Comput. Surv. 55, 9, Article 195 (September 2023), 35 pages. https://doi.org/10.1145/3560815

[6] Pearce, Hammond, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. "Examining zero-shot vulnerability repair with large language models." In 2023 IEEE Symposium on Security and Privacy (SP). 2023.

[7] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lun Yiu Nie, and Xin Xia. 2021. Code structure guided transformer for source code summarization. arXiv preprint arXiv:2104.09340 (2021).

[8] Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020).

Model	Ground Truth	Zero-Shot	In-Context
PaLM	Creates a table named 'customers' with the specified columns and constraints	Create table customers	-- Comments: -- * Creates a table called customers. -- * The customer_id column is the primary key and is auto-incremented. -- * The first_name and last_name columns are required and cannot be null. -- * The email column is unique and cannot be null.
LLaMa-2		Creates a new table called "customers" with the following columns: * customer_id (INT, PRIMARY KEY, AUTO_INCREMENT): A unique identifier for each customer. * first_name (VARCHAR(50), NOT NULL): The customer\'s first name. * last_name (VARCHAR(50), NOT NULL): The customer\'s last name.\n * email (VARCHAR(100), UNIQUE, NOT NULL): The customer\'s email address. This table will contain all the customers in our database.	-- Comments: -- * Creates a table called customers. -- * The customer_id column is the primary key and is auto-incremented. -- * The first_name and last_name columns are required and cannot be null. -- * The email column is unique and cannot be null.

TABLE IV
EXAMPLE OF GENERATED COMMENTS FOR SQL QUERY.

[9] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K VijayShanker. 2010. Towards automatically generating summary comments for java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering. 43–52.

[10] Iyer, Rajasekar, et al. "Summarizing Source Code using a Neural Attention Model." *Proceedings of the 34th International Conference on Machine Learning.* 2017.

[11] CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation EMNLP 2021 · Yue Wang, Weishi Wang, Shafiq Joty, Steven C. H. Hoi ·

[12] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665.

[13] Ruyun Wang, Hanwen Zhang, Guoliang Lu, Lei Lyu, and Chen Lyu. 2020. Fret: Functional reinforced transformer with bert for code summarization. IEEE Access 8 (2020), 135591–135604.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 4171–4186.

[15] Svajlenko, Jeffrey, and Timothy Menzies. "When Is a Liability a Benefit? An Empirical Study of Java Program Commenting." *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* 2015.

[16] Hu, Heng, et al. "Deep code comment generation." *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers).* 2018.

[17] Hindle, Abram, et al. "On the naturalness of software." *2012 34th International Conference on Software Engineering (ICSE).* 2012.

[18] Laura Moreno, "Automatic generation of natural language summaries for java classes". *In 2013 21st International Conference on Program Comprehension (ICPC). IEEE, 23–32.*

[19] Sarah Rastkar. "Generating natural language summaries for crosscutting source code concerns". *In 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, 103–112.*

[20] Nahla J Abid. "Using stereotypes in the automatic generation of natural language summaries for c++ methods". *In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 561–565.*

[21] Sawan Rai. " A Review on Source Code Documentation". *ACM Transactions on Intelligent Systems and Technology (TIST) (2022).*

[22] Paul W McBurney. "Automatic documentation generation via source code summarization of method context." *In Proceedings of the 22nd International Conference on Program Comprehension. 279–290.*

[23] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In International conference on machine learning. PMLR, 2091–2100.

[24] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:1707.02275 (2017).

[25] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 397–407.

[26] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, 200–20010.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering 25, 3 (2020), 2179–2217.

[28] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 826–831.

[29] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. DeveloperIntent Driven Code Comment Generation. arXiv preprint arXiv:2302.07055 (2023).

[30] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653 (2020).

[31] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. arXiv preprint arXiv:2105.08645 (2021).

[32] Comments in Source Code: A classification approach Amisha Shingala1 Namrata Shroff2 CHARUSAT University, CMPICA, Changa, Gujarat, India Gujarat Technological University, Ahmedabad, Gujarat, India

[33] Modha, Sandip Jayantilal. "Microblog processing: summarization and impoliteness detection." PhD diss., Dhirubhai Ambani Institute of Information and Communication Technology, 2019.