Trends and Challenges in Machine Learning for Code Summarization and Comprehension: A Systematic Literature Review

Panagiotis Mantos, Fotios Kokkoras and George Kakarontzas
Digital Systems Department, University of Thessaly, Gaiopolis Campus, Larissa, Greece

Keywords: Code Summarization, Code Comprehension, Systematic Literature Review, Machine Learning, Neural Networks.

Abstract: This systematic literature review explores current trends in automatic source code summarization and comprehension. Through extraction and analysis of information from six reputable digital libraries, we answered the following three questions: a) Which are the current machine learning models to generate summaries for source code? b) What factors should be considered when selecting an appropriate machine learning model for code summarization and comprehension? c) What are the possible future directions for research and development in machine learning for code summarization and comprehension, considering current limitations and emerging trends. The findings show significant progress with deep learning methods dominating this area.

1 INTRODUCTION

Code comprehension is a key challenge for programmers, requiring informative and concise documentation. Creating effective summaries is time-consuming and costly. Traditionally, developers write source code documentation by hand, leading to poor quality and inconsistencies due to unique coding style of each individual developer and frequent software updates (Zhang et al., 2022). These issues highlight the need for advanced automated methods to accurately and clearly analyse and summarize code, particularly in projects involving many developers.

In recent years, significant progress has been made in automated source code summarization using machine learning and deep learning technologies. Advances in Natural Language Processing (NLP), deep neural networks, and large code datasets have been crucial. The transformer-based architecture, introduced by (Vaswani et al., 2017), uses self-prediction to efficiently process sequences, improving on traditional models. This technology is now applied in various fields, including NLP and source code summarization.

Unlike traditional models, transformers process the entire text at once rather than in sequence. This enables them to grasp the interconnections of words without considering their distance or position, which is crucial for understanding the general structure of source code.

High-quality source code summaries are crucial for understanding existing systems, especially for tasks like system rearchitecting (e.g., for the cloud). This topic is gaining popularity in software engineering, creating specific needs, and this motivated us to explore it further. This literature review explores research trends in source code summarization, focusing on machine learning-based techniques and discussing their advantages and disadvantages.

The rest of this paper is organized as follows: Section 2 presents the plan or our review. Section 3 presents important aspects of the review, whereas Section 4 presents the findings of our review. Discussion and research questions are set in Section 5 and Section 6 concludes the paper.

2 REVIEW PLANNING

This literature review on code summarization and comprehension focuses on the use of machine learning techniques. Our work was based on (Kitchenham, B. and Charters, S., 2007) and the research protocol followed included the following steps:

1. Sources of Literature: major digital libraries, peer-reviewed journals and reputable scientific conferences, were used.

Mantos, P., Kokkoras, F. and Kakarontzas, G.
Trends and Challenges in Machine Learning for Code Summarization and Comprehension: A Systematic Literature Review.
DOI: 10.5220/0013275300003928

In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 468-475
ISBN: 978-989-758-742-9; ISSN: 2184-4895

2. Article Selection: We set inclusion and exclusion criteria to choose the right scientific articles for our research
3. Data Extraction & Synthesis: Data was extracted and synthesized from the selected articles.
4. Quality Assessment and Presentation of Findings: The quality of the included articles was assessed, and findings were recorded.

The search strategy followed contains several steps, including database selection and search, the application of inclusion and exclusion criteria, keyword search and the snowball search method. A detailed search log was recorded during the process, capturing the search terms, the databases used, and the number of articles retrieved.

2.1 Sources of Literature

First, literature selection was performed. The selected publications were extracted from six reputable digital libraries and databases, containing large collections of literature such as academic research, peer-reviewed articles, conference papers and journal publications related to computer science and software engineering, which are presented in Table 1.

Table 1: Sources of articles and articles used.

Source	URL	Articles Used
Google Scholar	scholar.google.com/	13
ACM DL	portal.acm.org/	16
IEEE Xplore	ieeexplore.ieee.org/	11
ScienceDirect	www.sciencedirect.com/	4
SpringerLink	link.springer.com/	1
ArXiv	arxiv.org/	13
Total:		58

The timeframe of the selected articles is between 2017 and 2024, to ensure the best possible relevance to current scientific data and findings.

2.2 Initial Article Selection

For the retrieval of publications from the sources of Table 1 we used the logic of the following query:

```
("Machine Learning" OR "Deep Learning"
 OR "Neural Networks" OR "Natural Lan-
guage Processing" OR "Artificial Intel-
ligence") AND ("Code Summarization" OR
"Code Documentation" OR "Comment Gener-
ation" OR "Program Comprehension" OR
"Code Comprehension").
```

In addition, we applied the snowballing method (Wohlin 2014), which uses relevant documents to identify other similar documents through their citations and references. This allows the discovery of more influential papers in the domain.

2.3 Inclusion and Exclusion Criteria

The articles retrieved were filtered based on the inclusion and exclusion criteria presented in Table 2 and Table 3, respectively. We kept articles that met all the inclusion and none of the exclusion criteria.

Table 2: Inclusion Criteria (IC).

IC-1	Machine Learning techniques applied for code comprehension and summarization.
IC-2	Timeframe (2017-2024)
IC-3	Quantitative and Qualitative Studies
IC-4	Peer-Reviewed Studies
IC-5	The study proposes a method for assisting with the topic of our literature review.

Table 3: Exclusion Criteria (EC).

EC-1	Off-Topic Studies
EC-2	Lack of Empirical Evaluation
EC-3	Non-Peer-Reviewed Publications
EC-4	Non-English Studies
EC-5	Short Studies (below 5 pages)
EC-6	Duplicate Removal

3 ASPECTS OF REVIEW

Articles were initially screened by their titles and abstracts, excluding those not clearly related to the topic. The selected studies were then fully analysed to ensure they specifically discuss the use of machine learning techniques for analysing, summarizing, or understanding source code.

From a total of 89 studies retrieved from the sources used, 58 were included (Table 1) and 31 were excluded, based on the IC/EC criteria. The included articles were categorized according to their methodology in the categories presented in Figure 1.

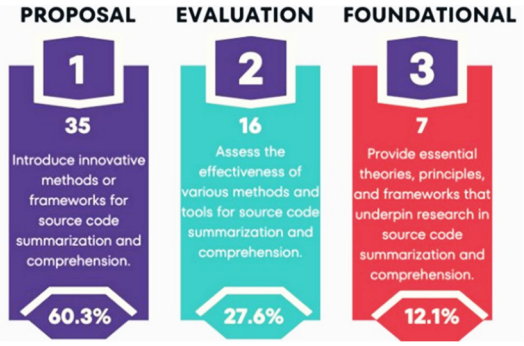

Figure 1: Classification of included studies.

3.1 Machine Learning Methodology

We reviewed the 58 articles selected, and we categorized them according to the machine learning approach they use for the comprehension and the summarization of source code. Table 4 displays the number of articles of each category examined.

Table 4: Classification of articles examined based on architecture and methodology.

Category	Number of Articles
Transformers with Self-Attention	19
LSTM, Attention Mechanism	9
Graph Neural Networks (GNN)	5
Deep Reinforcement Learning	2
Hybrid Models (Including ASTs)	23

3.2 Evaluation Metrics

BLEU and ROUGE (Yang et al., 2018) (Blagec et al., 2022) are metrics commonly used to evaluate the quality of source code summaries.

The BLEU (Bilingual Evaluation Understudy) metric measures how many n-grams, that is, continuous strings of *n* elements (words or tokens) of the generated text are present in the reference text. Its goal is to determine how accurately the generated text matches the reference text, by comparing the n-grams with each other.

ROUGE evaluates the quality of summaries by measuring the overlap of n-grams and word order between the reference text and the generated text.

3.3 Practical Applications

Methods and machine learning techniques used to automatically summarize and comprehend source code, help to its review and maintenance by generating new comments or documentation whenever changes are made. Thus, ideally, when programmers modify the source code, they do not need to provide further explanations, which is extremely useful.

3.4 Challenges and Possible Biases

As machine learning models advance, new techniques bring up more questions. A common challenge is adapting to the diverse and unique coding styles of different developers. Another issue is ensuring the accuracy and efficiency of machine-generated summaries, aiming to make them more human-like for better comprehension of source code functionality.

4 APPROACHES REVIEWED

4.1 Transformer Models and Self-Attention Mechanisms

In their paper “Attention is All You Need”, (Vaswani et al. 2017) introduced a transformer model that depends only on self-attention mechanisms and does not need to use traditional recurrent or convolutional neural networks. According to (Vaswani et al. 2017), recurrent neural networks (RNNs) process the data sequentially, making them less effective in capturing long-range dependencies. In contrast, the Transformer model applies the self-attention mechanism, so that it processes the input data of the entire sequence in parallel simultaneously, and this reduces the training time and provides better scalability. The model can focus on different parts of the input data, and so it can more efficiently capture the dependencies and the relationships that exist within the source code, even if they exist in different parts of the code.

The Python function below, that calculates the average of a list of numbers, shows how transformers analyze code snippets, understanding the relationships within the code, and then proceed to create summaries:

```
def calculate_average(numbers):
   total = sum(numbers)
   count = len(numbers)
   return total / count
```

When a transformer model is used, the first thing it does is to divide the function into tokens (words or symbols). Then, the self-attention mechanism calculates the focus scores for each token based on its relationship to every other token in the code. In this way, it learns which tokens are most related to each other, i.e. the most important dependencies that exist within the code. In the example, the token `sum(numbers)` is closely related to the token `total`. Tokens are seen and compared in large code expanse, and this results in the production of a better summary.

The paper "CodeBERT: A Pre-Trained Model for Programming and Natural Languages" by (Feng et al., 2020), examines and analyzes the advantages and the benefits of pre-training transformer models on both source code data and natural language data. This model uses the self-attention mechanism to capture the semantic and syntactic relationships between the code and the natural language. Through this, it shows high performance in retrieving the source code as well as in generating a summary of the source code with high accuracy.

Due to the self-attention mechanism, this model can produce summaries that look like they were created by a human hand, as this mechanism allows the model to consider only the most important parts of the code and as a result to produce higher quality summaries and more contextually related.

While CodeBERT emphasizes natural language and code, another transformer-based model, Codex, also shows remarkable performance in code summarization.

The next Python example, shows the ability of the model to understand semantic relationships between code elements, using the Self-Attention mechanism to identify critical parts of the function and generate an accurate and concise summary.

```
def vec_to_halfvec(vec):
  d = vec[1:] - vec[:-1]
  if ((d / d.mean()).std() > 1e-14)
     or (d.mean() < 0):
    raise ValueError('vec must be np
     .arange() in increasing order')
  dx = d.mean()
  lowest = np.abs(vec).min()
  highest = np.abs(vec).max()
  return np.arange(lowest, highest +
    0.1*dx,dx).astype(vec.dtype)
```

The mechanism identifies that the key parts of the function are the computations related to vec

e.g. `d = vec[1:] - vec[:-1]`

and the conditions that shows its structure

e.g. `if ((d/d.mean()).std()>1e-14)`
`or (d.mean() < 0)`

These parts have more weight because they define the main logic of the method that is used and ensure the correctness of it. Parts that are less important, such as the return statement, get less attention and therefore less weight, as they do not change the overall purpose of the function very much.

(Khan and Uddin, 2022) proposed and evaluated the potential and performance of the Codex model in the field of automatic source code summarization. This model is a version of GPT-3, and its results have shown that even when it is trained with basic, minimal training using one-short learning, it can outperform conventional code summarisation techniques.

4.2 LSTM and Attention Mechanisms

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Networks (RNNs) capable of maintaining information in long code sequences, which helps them to better understand the source code and to generate accurate summaries.

(Hu et al., 2018), showed that when LSTM networks are combined with attention mechanisms, the relevance and the accuracy of the generated summaries is improved, as they can focus on the most important parts of the code.

```
def add(a, b): return a + b
```

In the preceding Python example, a LSTM network processes each token (`def`, `add`, `a`, `b`, `return`, etc.) sequentially, keeps track of the dependencies between these tokens, and can understand that the function takes `a` and `b` as input and returns the operation `a+b`. This information is stored in memory cells throughout the sequence. At each step the LSTM decides which information to keep because it is important and which to forget.

(Wan et al., 2018) researched how reinforcement learning can be combined with the LSTM architecture to improve the quality of the automated generated source code summaries. They found that by adding reinforcement learning, the model each time improves its performance by giving focus to the feedback it receives. These models are classified as hybrid as they combine techniques and different methods, and as a matter of fact, the specific ones that combine LSTM networks and other machine learning techniques, show significant improvements in both the conciseness and accuracy of the generated source code summaries.

LSTMs are effective and with the addition of the transformers presented by (Ahmad et al., 2020), produce better source code summaries.

4.3 Graph Neural Networks

Graph Neural Networks (GNNs) were created to process and display data in a graph structure of any shape and size. They use nodes and edges to represent the relationships of the data. They find application in social networks as well as in source code analysis, an area we study in this literature review. Therefore, GNNs have the potential to effectively show the interactions between data and this leads to a better and deeper understanding of it (Allamanis et al., 2017), (Zhang et al., 2023).

By representing source code in the form of graphs, such as Abstract Syntax Trees (ASTs) or control flow graphs (CFGs), GNNs learn through the complicated dependencies that exist within the code. Because of this, they can then generate accurate summaries because the model has a better understanding of the syntactic and semantic structure of the code. (LeClair et al., 2020) showed major improvements in source code

summarization by focusing on the connections between the parts of the code as he researched how automatic source code summarization can be improved using GNNs. The authors suggest that future research can focus on embedding and enhancing dynamic code analysis, capturing runtime behaviours and interactions. They propose extending the model to support multiple programming languages and larger code bases, addressing diversity in coding styles.

For example, we present a source code Java example extracted by (LeClair et al., 2020), and explain it:

```
public int indexOf(Object o) {
  if (o == null) {
    for (int i = 0; i < size; i++) {
      if (gameObjects[i] == null) {
        return i;
      }}}
  else {
    for (int i = 0; i < size; i++) {
      if (o.equals(gameObjects[i])){
        return i;
      }}}
  return -1;}
```

By using a graph neural network (GNN), the abstract syntax tree (AST) of this method is processed as a graph where nodes represent code structures such as `if`, `for`, `return`, and `o.equals`. The edges capture the relationships between these structures. GNN analyzes this structure and focuses on the most important elements of the code. For example, it detects that the return statement is critical (`return i;`), because they define the output of the method. It understands that comparisons (`o==null` and `o.equals`) are essential and affect the flow of the execution. Using this analysis, GNN returns the index of the first appearance of the defined element. This example shows how GNNs use code structural information to produce accurate and informative summaries.

(Guo et al., 2021) further improved the model's ability to understand these relationships by enhancing GNNs with data flow information and becoming able to produce more accurate and relevant summaries.

4.4 Deep Reinforcement Learning

Reinforcement learning improves decisions through interactions with the environment and feedback. In source code comprehension and summarization, it enhances summary quality by generating summaries, receiving feedback, and using it to improve future summaries. With continuous feedback the model produced summaries become more accurate in capturing source code functionality. (Wan et al., 2018) noted the limitations of traditional neural models and used deep reinforcement learning to enhance summary quality through feedback. They achieved better results but suggested that future research should use more complex reward functions and prioritize human feedback.

(Wang et al., 2022) further improved it by proposing a combination of reinforcement learning with hierarchical attention mechanisms. This combination allowed the model to understand the overall context of the code, but also to focus on the details of it.

4.5 Hybrid Models

(Hu et al., 2020) proposed a hybrid model, which combines lexical and syntactical analysis for achieving better and more efficient source code summaries. This model uses a neural network that processes lexical information and a syntactic analyzer to improve the comprehension of the code structure. The results are comments and code summaries that are more accurate and of higher quality.

(Parvez et al., 2021), proposed a hybrid model that combines retrieval-based techniques with generative models. Through the retrieval part, the proper code snippets are identified and extracted from a large dataset. Then, these are used by a neural network which produces more accurate and reliable summaries.

(Lu et al., 2023), proposed a method that combines deep learning models with semantic analysis, which can capture the syntactic and the semantic structure of the source code. Thus, it is possible to produce summaries that not only focus on being concise, but also provide important contextual information, which improves the quality of the summary.

5 DISCUSSION

In this section, we commented on the findings of the review, identified some issues and proposed directions for further research. We organized this discussion using three main research questions.

5.1 RQ1

Which Are the Current Machine Learning Models to Generate Summaries for Source Sode?

It is observed that in recent years, various models have been proposed that use different machine learning techniques, but it is noteworthy that they all focus on a common factor, which is to achieve a better understanding of the structure of the source code, either at the semantic or syntactic level, in order to have a

better view of the functionality of the code that is being analysed.

Current transformer-based models, such as CodeBERT, and Codex (Khan and Uddin, 2022) have undoubtedly presented very encouraging results and remarkable accuracy in generating automated code summaries. For example, Codex achieves an average BLEU score of 20.63 across six programming languages, showing an 11.2% improvement compared to older state-of-the-art methods such as CoTexT (Phan et. Al., 2021). Similarly, CodeBERT have shown great progress on many datasets, especially for Java and Python, allowing the parallel processing of the data. (Feng et al., 2020)

However, despite this progress, there are still limitations on how they can understand the intention and the logic of the developer behind the code they are examining.

1. Metric Limitations: BLEU and ROUGE cannot be fully associated with human understanding. (Stapleton et al., 2020) found that programmers performed better with human generated summaries than with machine-generated summaries despite high BLEU scores.
2. Contextual and semantic gaps: Current models often have difficulty capturing deeper intentions or logical relationships within the code. For example, BLEU scores drop by 4-6 points for complex methods. (Stapleton et al., 2020)

When choosing a machine learning model, the authors recommend prioritizing how well programmers understand the generated summaries, rather than just the evaluation metric scores. This is because high scores don't always reflect the actual comprehension of the summaries by developers.

Future research should focus on developing more intelligent models that analyse both the syntactic and semantic structure of code. This includes considering the history of code changes and the intentions behind each approach. Additionally, it will be nice to enable models to perform real-time analysis of source code, speeding up development. This would eliminate the need to execute code from scratch each time, as the model would already have updated crucial details.

Finally, these models can improve collaboration between teams. Summarization tools provide crucial information about each team's code, helping them understand it quickly and efficiently.

5.2 RQ-2

What Factors Should Be Considered When Selecting a Machine Learning Model for Code Summarization and Comprehension?

Choosing the most suitable model for a source code summarization and comprehension task is critical. A key factor to consider is adaptability. Related to the adaptability part, models can adapt to different programming languages, and it is worth mentioning that they can perform efficiently even with limited training data. Two of these use cases are one-shot learning, where the model performs a task by learning from a single example, and few-shot learning which allows the model to understand and summarize source code using a small number of training examples. An evaluation performed by (Khan and Uddin, 2022), showed that Codex model outperformed other models in six programming languages when trained with few-shot examples. Therefore, adaptability makes these models very useful in the field of software engineering, especially in cases such as comprehending identifier names, APIs and different coding styles. Moreover, they are useful in tasks where the available data is limited, and large-scale model training is not possible. For example, the TL-CodeSum method, presented by (Hu et al., 2018), uses API sequences as mid-range representations to capture long-range dependencies between code points and APIs, to improve the understanding and summarization of source code.

To select the most suitable machine learning model for source code summarization, several factors that affect the performance and usability of the model are considered:

1. Handling long-range dependencies and scalability: The ability of the model to handle long-range dependencies is critical for understanding complex code structures, where functions and variables may be placed in different parts of the source code. (Ahmad et al., 2020) proposed a transformer architecture combined with self-attention mechanisms to capture long-range dependencies within the source code. This technique outperformed traditional RNN-based models, including LSTMs, in terms of producing more accurate and relevant summaries. Additionally, scalability ensures that the model can handle large datasets and more complex tasks, offering solutions to big data challenges.
2. Parallel information processing: Transformers can process source code using the Self-Attention Mechanism (Vaswani et al., 2017) to analyze multiple aspects of the code at the same time, increasing speed and efficiency.
3. Use of reliable evaluation metrics: The use of reliable evaluation metrics allows the performance of the model to be measured accurately.

Based on these findings, we propose some future research directions:

1. Expanding language support: Current models mainly focus on a few programming languages like Python, commonly used in machine learning. Future models need to be more adaptable to various programming environments and styles.
2. Improving data quality: Models often learn from datasets with errors or unhelpful comments, which affects summary accuracy. Fixing these dataset issues is essential to improve model reliability.
3. Resource efficiency: For large code bases, a model's runtime efficiency and computing power are crucial. Future models need to balance performance and resource consumption to ensure practical usability.

5.3 RQ-3

What Are Possible Future Directions for Research and Development for Code Summarization, Considering Current Limitations and Emerging Trends?

Apart from the research directions mentioned in the two previous sections, another important research focus should be scalability and performance. As codebases continuously grow, models need to process larger amounts of datasets, while maintaining accuracy and reducing computational power. The goal is to optimize current models and architectures to achieve greater efficiency. This could be done by developing neural networks, which would require less processing power and computational resources but still produce accurate and reliable summaries of the source code. In other words, these models should handle larger amounts of data without the need for more expensive hardware.

An additional question is how these models could be integrated in developers' IDEs to continuously produce summaries that are up to date as the code is written?

As we have already mentioned in the paper, a very common issue faced is the diversity in the coding style of developers. Future research should focus on how machine learning models can adapt more effectively to different coding styles, possibly through direct feedback, and thus offer greater personalization and efficiency.

Another relevant challenge is the issue of biases. How can we detect and moderate the biases observed in the data used to train machine learning models?

Machine learning models are trained on data that often contain biases. Future research should focus on creating algorithms that can detect and reduce biases and unwanted distortions in the summaries, ensuring that the generated summaries are accurate and truthful. At the same time, ethical issues are extremely important, including the need to ensure that automated summaries do not accidentally reveal sensitive or private information, such as credentials or copyright information. In addition, it is necessary to respect the licenses and rights of the source code, so that the summaries produced do not violate licensing terms or even reveal parts of the code that should not be made public.

A key question is how we can develop mechanisms which ensure that artificial intelligence is applied responsibly and ethically, while giving the necessary attention to user's privacy and the integrity of generated results.

To address these issues, we propose some future research directions:

1. For IDE integration, models must be dynamically updated with new information, without the need for training from scratch every time. This ensures that summaries remain accurate and updated.
2. Personalized feedback (e.g. reviews or quick edits from developers) can help customize summaries to the specific needs of a team, improving overall productivity.
3. An innovative idea for reducing biases in AI models is the integration of self-evaluating mechanisms. Through these, the model will continuously evaluate its responses to identify and correct potential biases. This idea is based on finding a way for models not to rely only on external factors, but to learn from their own responses, specifically in the ethics part to achieve a higher level of integrity.

6 CONCLUSIONS

This literature review examined source code summarization and comprehension using machine learning techniques. It highlights progress and challenges, with deep learning, especially transformers, leading the way.

While deep reinforcement learning shows promise, it evolves slowly. Significant progress requires both syntactic and semantic code understanding, as well as new performance metrics.

Finaly, three umbrella research questions were proposed to guide future research directions.

REFERENCES

Zhang, C., Wang, J., Zhou, Q., Xu, T., Tang, K., Gui, H., & Liu, F. (2022). A Survey of Automatic Source Code Summarization. *Symmetry*, *14*(3), 471. https://doi.org/10.3390/sym14030471

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Inf. Processing Systems*, 30. https://doi.org/10.5555/3295222.3295349

Kitchenham, B., & Charters, S. (2007). *Guidelines for performing systematic literature reviews in software engineering* (V.2.3). EBSE Technical Report EBSE-2007-01, Software Engineering Group, School of Computer Science and Mathematics, Keele University, and Department of Computer Science, University of Durham.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proc. of the 18th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE), ACM*. https://doi.org/10.1145/2601248.2601268

Yang, A., Liu, K., Liu, J., Lyu, Y., & Li, S. (2018). Adaptations of ROUGE and BLEU to better evaluate machine reading comprehension task. In *Proc. of the Workshop on Machine Reading for Question Answering* (pp. 98–104). Association for Computational Linguistics. https://doi.org/10.18653/v1/W18-2611

Blagec, K., Dorffner, G., Moradi, M., Ott, S., & Samwald, M. (2022). A global analysis of metrics used for measuring performance in natural language processing. *arXiv*. https://arxiv.org/abs/2204.11574

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. https://arxiv.org/abs/2002.08155

Khan, J. Y., & Uddin, G. (2022). Automatic code documentation generation using GPT-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 124–135). IEEE. https://doi.org/10.1145/3551349.3559548

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2018). Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC)* (pp. 200–210). Association for Computing Machinery. https://doi.org/10.1145/3196321.3196334

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., & Yu, P. S. (2018). Improving automatic source code summarization via deep reinforcement learning. In *Proc. of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE)* (pp. 397–407). ACM https://doi.org/10.1145/3238147.3238206

Ahmad, W., Chakraborty, S., Ray, B., & Chang, K.-W. (2020). A transformer-based approach for source code summarization. In *Proc. of the 58th Annual Meeting of the Association for Computational Linguistics* (pp. 4998–5007). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.449

Allamanis, M., Brockschmidt, M., & Khademi, M. (2018). Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1711.00740

Zhang, M., Zhou, G., Yu, W., Huang, N., & Liu, W. (2023). GA-SCS: Graph-augmented source code summarization. *ACM Transactions on Asian and Low-Resource Language Information Processing, 22*(2), Article 20. https://doi.org/10.1145/3554820

LeClair, A., Haque, S., Wu, L., & McMillan, C. (2020). Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)* (pp. 184–195). ACM https://doi.org/10.1145/3387904.3389268

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. In *Int. Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2009.08366

Wang, W., Zhang, Y., Sui, Y., Wan, Y., Zhao, Z., Wu, J., Yu, P. S., & Xu, G. (2022). Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Transactions on Software Engineering*, *48*(1), p.102–119. https://doi.org/10.1109/tse.2020.2979701

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2020). Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering, 25*(3), pp.2179–2217. https://doi.org/10.1007/s10664-019-09730-9

Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021, August 26). *Retrieval Augmented Code Generation and Summarization*. arXiv.org. https://arxiv.org/abs/2108.11601

Lu, X., & Niu, J. (2023). Enhancing source code summarization from structure and semantics. In *Proc. of the 2023 Int. Joint Conf. on Neural Networks*. IEEE. https://doi.org/10.1109/ijcnn54540.2023.10191872

Phan, L., Tran, H., Le, D., Nguyen, H., Anibal, J., Peltekian, A., & Ye, Y. (2021). CoTexT: Multi-task Learning with Code-Text Transformer. In *Proc. of the 1st Workshop on Natural Language Processing for Programming* (pp. 40–47). Assoc. for Computational Linguistics. https://doi.org/10.18653/v1/2021.nlp4prog-1.5

Stapleton, S., Gambhir, Y., LeClair, A., Eberhart, Z., Weimer, W., & Leach, K. (2020). A human study of comprehension and code summarization. In *Proceedings of the 28th Int. Conference on Program Comprehension (ICPC)* (pp. 2–13). Association for Computing Machinery. https://doi.org/10.1145/3387904.3389258

Due to space limitations, the list of the 58 articles used in this literature review is provided online at: https://gkakaron.users.uth.gr/files/ENASE_2025_SLR_ARTICLES.pdf