DOI: 10.15514/ISPRAS-2026-38(4)-6

A Survey of Deep Learning Methods for Source Code Summarization

A.I. Valeev, ORCID: 0000-0002-9511-1488 <ai.valeev@innopolis.ru>
A.O. Korshuk, ORCID: 0009-0008-6719-8281 <a.korshuk@innopolis.university>
V.V. Ivanov, ORCID: 0000-0003-3289-8188 <v.ivanov@innopolis.ru>

Innopolis University,
1, University st., Innopolis, 420500, Russia.

Abstract. Deep neural models have become the state of the art for code summarization. Before the era of Large Language Models, many specialized models applied various methods to improve baseline sequence-to-sequence models, leveraging code structure and similarities between code snippets and summaries, and employing multi-task learning. This survey elicits the design building blocks of these methods and builds a taxonomy of them. It also describes how different models apply the same method or how a single model applies different methods. Furthermore, we discuss the current flaws of these models, potential research directions, and the applicability of large language models in code summarization, comparing them with the surveyed methods.

Keywords: code summarization; transformer; RNN encoder-decoder; deep learning.

For citation: Valeev A.I., Korshuk A.O., Ivanov V.V. A Survey of Deep Learning Methods for Source Code Summarization. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 4, part 1, 2026, pp. 101-134. DOI: 10.15514/ISPRAS-2026-38(4)-6.

Acknowledgements. The authors gratefully acknowledge the institutional support of Innopolis University.

Обзор методов глубокого обучения для суммаризации кода

А.И. Валеев, ORCID: 0000-0002-9511-1488 <ai.valeev@innopolis.ru>
А.О. Коршук, ORCID: 0009-0008-6719-8281 <a.korshuk@innopolis.university>
В.В. Иванов, ORCID: 0000-0003-3289-8188 <v.ivanov@innopolis.ru>

Университет Иннополис,
Россия, 420500, г. Иннополис, ул. Университетская, д. 1.

Аннотация. Глубокие нейронные модели стали передовым подходом в задаче суммаризации кода. До эпохи больших языковых моделей многие специализированные модели применяли различные методы для улучшения базовых моделей преобразования последовательностей, используя структуру кода, за счёт сходства фрагментов кода и их описаний, а также применяя многозадачное обучение. В данном обзоре выделяются ключевые архитектурные компоненты этих методов и предлагается их таксономия. Также описывается, как разные модели используют один и тот же метод или как одна модель сочетает несколько методов. Кроме того, мы обсуждаем текущие недостатки этих моделей, потенциальные направления дальнейших исследований и применимость больших языковых моделей в задаче суммаризации кода, сравнивая их с рассмотренными подходами.

Ключевые слова: суммаризация кода; трансформер; рекуррентные нейронные сети; энкодер; декодер; глубокое обучение.

Для цитирования: Валеев А.И., Коршук А.О., Иванов В.В. Обзор методов глубокого обучения для суммаризации кода. Труды ИСП РАН, том 38, вып. 4, часть 1, 2026 г., стр. 101–134 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(4)–6.

Благодарности: Авторы выражают искреннюю благодарность Университету Иннополис за институциональную поддержку.

1. Introduction

In recent years, deep neural networks have significantly advanced natural language processing. Originally developed for machine translation, sequence-to-sequence models have achieved strong performance in various language tasks. The emergence of self-attention further accelerated progress by improving performance and scalability over recurrent models. The two main architectures – Recurrent Neural Network (RNN) Encoder–Decoder and Transformer – are widely applied to text-to-text tasks, including source code summarization, where they serve as strong baselines [1-2].

To improve RNN Encoder-Decoder and Transformer baselines, many studies leverage properties that distinguish code from natural language. Code follows strict syntax rules and can be represented as code graphs, e.g., abstract syntax tree or control flow graph. Moreover, different code snippets share similarities if they implement same functionality across a wide range of applications, e.g. interfaces and validation for databases, asynchronous or parallel processing, etc. Researchers apply various methods to leverage these features and improve code summarization.

Our survey elicits the design building blocks of RNN Encoder-Decoder and Transformer models specialized for code summarization. We build a taxonomy of these building blocks and describe how different models apply the same method, or how a single model applies different methods.

This survey addresses the following research questions:

- What constitutes a typical framework for code summarization, including the architecture of baseline models and common experimental setups?
- What methods have been proposed to improve deep learning–based code summarization, and which challenges do they address?
- What are the current open issues, challenges, and future research directions in code summarization?

We reviewed the studies that improve code summarization and built a taxonomy of methods. This taxonomy has three main categories: leveraging code graphs, retrieval augmentation, and multi-task learning. Each method mentions the related models, although some models may occur multiple times if they apply different methods.

We accompany the taxonomy with illustrations of methods and a tree diagram of models. We provide model results in each category to compare and contrast similar methods. We discuss the current issues of these models, potential research directions, and the applicability of large language models and their comparison with surveyed methods in code summarization.

The remainder of the paper is organized as follows: Section 2 overviews code summarization approaches, datasets, and metrics; Section 3 presents improvement methods by category; Section 4 discusses open issues and research directions; Section 5 reviews related surveys; and Section 6 concludes this survey.

2. Overview of Neural Code Summarization

Automatic source code summarization (ASCS) aims to generate natural language comments describing source code snippets. Most research focuses on method-level summarization, while fewer studies address commit message generation [3-5] class-level [6-7] and repository-level summarization [8-10].

2.1 Code Summarization

Formally, a code summarization model generates a natural language comment that describes what given code snippet does, as illustrated in Fig. 1. Code summarization is a part of broader code documentation, which includes API documentation, arguments description, method or class summaries, user stories, and schemes. However, most research focused on method-level summaries due to data availability on GitHub [11], whereas datasets for other documentation types are scarce. In this survey, we focus on deep learning models for method-level code summarization.

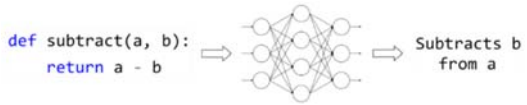

Fig. 1. Illustration of code summarization.

2.2 Machine Translation Solution to Code Summarization

Machine Translation (MT) approach prevails among the solutions to code summarization. The code summarization task can be seen as a translation from a programming language to a natural language. The general deep learning solution to the MT task is a sequence-to-sequence model trained on a collection of code snippets aligned with their comments.

Sequence-to-sequence models for code summarization are generally based on Recurrent Neural Networks (RNN) or Transformers [12]. The RNN solution consists of an encoder, a decoder and potentially an attention mechanism. For code summarization, the encoder takes code tokens as input and encodes them into latent representations, then the decoder generates the code comment one token at a time. If an attention mechanism is present, the decoder is able to focus on input tokens differently for each token being decoded. The encoder is a single- or multi-layer RNN cell, usually Long Short-Term Memory (LSTM) [13] unit or a Gated Recurrent Unit (GRU) [14], and might be bidirectional [15]. Similarly, the decoder is a LSTM or GRU but strictly unidirectional due to autoregressiveness. The attention mechanism for RNN is generally an additive Feed-Forward NN-based attention, which weighs the encoder outputs based on each encoder output and current decoder state. Transformer also consists of an encoder and a decoder but every layer of both has an attention mechanism. A single Transformer layer is a multi-head dot-product attention followed by a feed-

forward NN with normalization layers and residual connections. A high-level diagram of a baseline sequence-to-sequence model is illustrated in Fig. 2 (a) to compare and contrast with other methods.

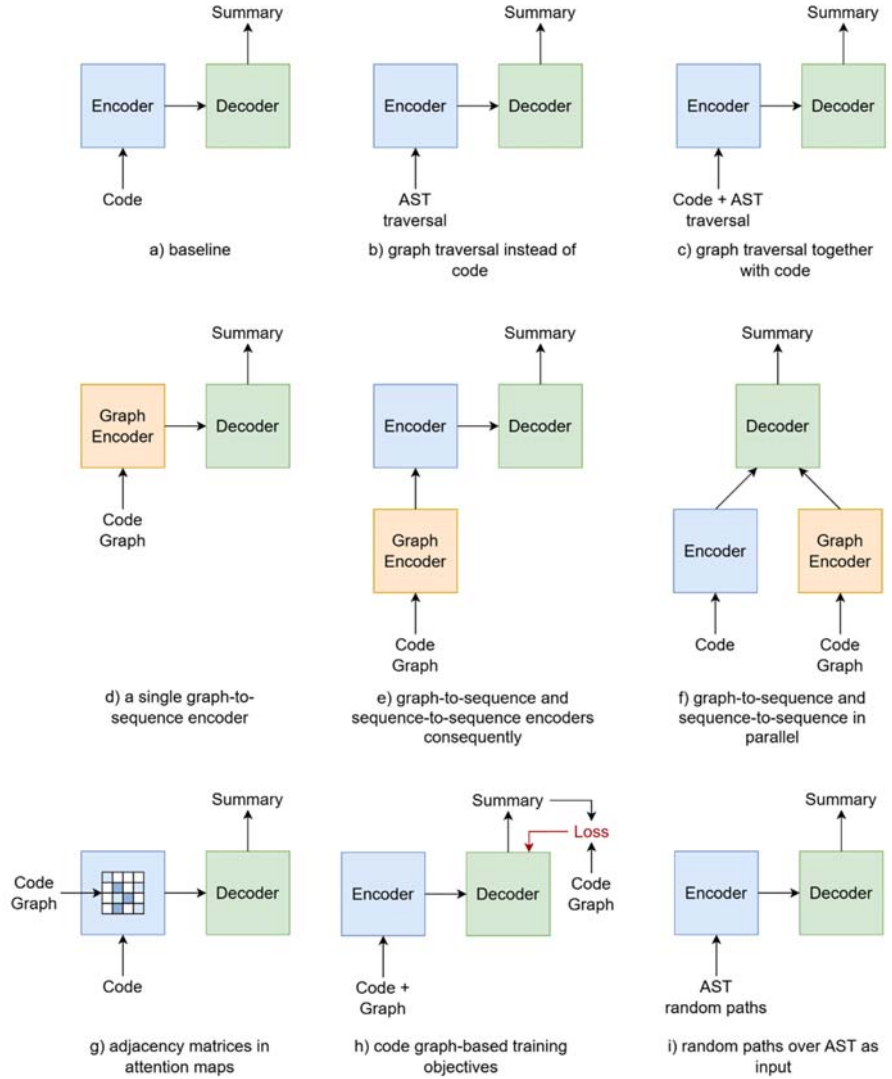

Fig. 2. High-level diagrams of code graph methods.

2.3 Baselines

This section describes the baseline models commonly used in code summarization research. Many advanced methods extend or enhance these baselines by incorporating additional architectural components and learning strategies.

- NNGen [5] is a non-neural nearest neighbor search for the closest comment, where code is represented as vectors in the form of bag-of-words.
- CODE-NN [2] is a 1-layer bidirectional LSTM Encoder-Decoder model with global attention [16] to summarize SQL and C# code, applying SGD with learning rate decay, 64 batch size and uniform parameter initialization.
- Seq2seq+Att [17] is a 2-layer GRU/LSTM Encoder-Decoder model with Bahdanau attention, Adam optimizer, 128 batch size and Xavier initialization.
- Transformer [12] is also a sequence-to-sequence model where RNN layers are replaced by multi-head dot-product attention followed by a Feed-Forward NN. Moreover, it features Byte-Pair Encoding to build a vocabulary and Absolute Positional Encoding.
- NCS [1] improves over Transformer by introducing Relative Positional Encoding to improve semantics capture and Copy Attention to favor generation of tokens from input.

2.4 Code Graphs

Source code can be represented in the form of various code graphs. Abstract Syntax Tree (AST) is a tree representation of the syntactical structure of the source code. Details appear only in the leaves, while the nodes represent hierarchical structure, e.g., a function body is split into statements, and statements are split into operands and operators. Data Flow Graph (DFG) represents relations between variables, shows which variables each variable depends on or computed from. The nodes are variables, and the directed edges denote where the value comes from. Control Flow Graph (CFG) represents possible execution paths of source code. The nodes are unbranched code blocks, and the oriented edges show which blocks might be executed next. Program Dependency Graph (PDG) combines data flow and control flow dependencies into a single graph. Fig. 3 and 4 show examples of AST, CFG and DFG for a Python function from Listing 1.

```
def fn(record):
    for db in all_databases:
        result = db.get(record)
        if result is not None:
            return result
        return None
```

Listing 1: Python code example.

2.5 Datasets

Model design contributes to the quality of code summarization but not as much as training data. Most studies evaluate their models on one or two of the six popular datasets for code summarization. Table 1 presents these popular datasets: code-docstring-corpus, TL-CodeSum, Funcom, CodeSearchNet, EMSE-Deepcom and C Code Summarization Dataset (CCSD). Former two datasets are also known as Python Code Summarization Dataset (PCSD) and Java Code Summarization Dataset (JCSD), respectively.

Shi et al. [18] analyzed three of these datasets and recommend using multiple datasets for a comprehensive model evaluation, since model rankings can differ across datasets.

The distribution of programming languages across datasets remains imbalanced, with a strong focus on Java and Python. This is largely due to historical dataset availability, as widely adopted benchmarks for other languages emerged later. At the same time, the surveyed methods are designed to be language-agnostic, suggesting that the observed findings may generalize beyond the most commonly evaluated languages.

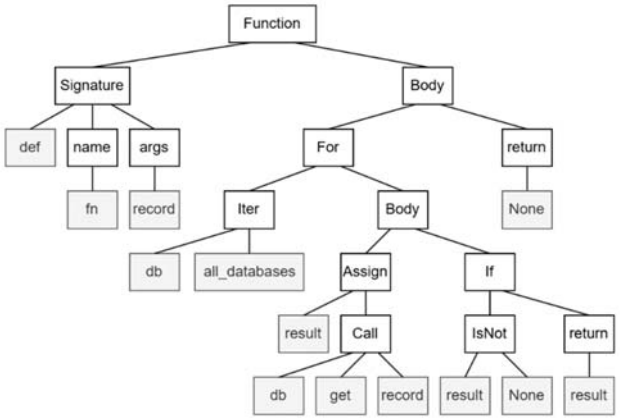

Fig. 3. Abstract Syntax Tree for Python code example.

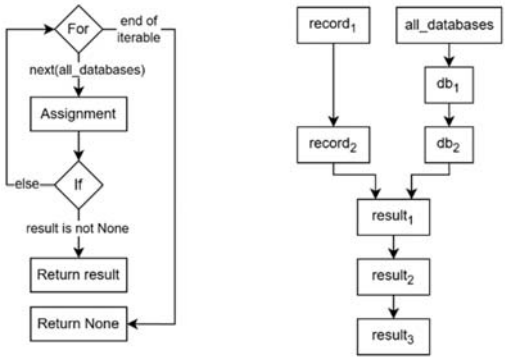

Fig. 4. Control Flow Graph (left) and Data Flow Graph (right) for Python code example.

Table 1. Datasets for code summarization.

Dataset	Lang.	Size	Source
PCSD (code-docstring-corpus) [19-20]	Python	113,108	GitHub [11]
JCSD (TL-CodeSum) [21]	Java	87,136	GitHub [11]
Funcom [22]	Java	2,149,121	Sourcerer [23]
CodeSearchNet [24]	Python, Java, Ruby, JavaScript, Go, PHP	2,326,976	GitHub [11]
EMSE-Deepcom [25]	Java	458,812	GitHub [11]
CCSD [26]	C	95,281	GitHub [11]

2.6 Text Preprocessing

Data has to conform model input format and should contain a minimum of noise. Preprocessing includes tokenization and any filtering or transformation applied to code or text. Tokenization splits the code or text into tokens – words or subwords. RNN solutions mostly split the code and text into

words and struggle from the out-of-vocabulary problem. On the contrary, Transformer solutions split code and text into tokens using Byte-Pair Encoding, which ensures that the selected subwords are the most frequent and can compose any word from the corpora.

Different studies preprocess code differently. Shi et al. [18] identified four main transformations: lower-casing, splitting by snake or camel case, replacing literals with special tokens, and removing punctuation. The ensemble of models with different preprocessing showed the highest overall score suggesting that different preprocessing techniques bring complementary information.

Inappropriate preprocessing of code comments can introduce significant noise into datasets. Shi et al. [27] identify seven major flaw categories: content tampering (up to 24%), where tags or URLs are mistakenly incorporated into summaries; over-splitting of variable identifiers (up to 24%), which may distort semantic meaning; verbose sentences (up to 22%) caused by extracting full sentences instead of concise summaries; partial sentences (up to 17%) resulting from truncated comments; non-literal processing (up to 7%) when English tokens are extracted from non-English comments; under-developed comments (up to 3.7%), such as auto-generated placeholders; and interrogative comments (up to 1%) that do not describe code functionality. After ensuring dataset quality, the subsequent step is to define appropriate evaluation protocols and metrics for model assessment.

2.7 Evaluation Metrics

After building and training a few models, we need to evaluate them properly for a fair comparison. To evaluate the performance of a code summarization system, the reviewed studies compare the generated summaries with the true comments. Different metrics exist for text-text comparison, the following are the most popular: BLEU, ROUGE-L, METEOR.

2.7.1 BLEU

BLEU (Bilingual Evaluation Understudy) [28] computes an n-gram precision of a candidate sentence with reference sentences. An n-gram is generally a sequence of few words, while BLEU uses sequences of one up to four words leading to BLEU-1, BLEU-2, BLEU-3 and BLEU-4, respectively. Brevity Penalty coefficient penalizes short sentences, since BLEU considers precision and not recall. To get a single BLEU score different smoothing methods are applied to BLEU 1-4. BLEU is widely used in Machine Translation.

$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{n\text{-gram} \in C} Count_{\text{clip}}\,(n\text{-gram})}{\sum_{C' \in \text{Candidates}} \sum_{n\text{-gram}' \in C'} Count(n\text{gram}')}$$

$$\text{BP} = \text{I}[\text{c} > \text{r}] + \text{I}[\text{c} \le \text{r}] * \text{e}^{(1-\text{r}/\text{c})}, \qquad \text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

where p_n is a precision score, $Count_{clip}$ clips the counts of matching words to the maximum count in references, BP is a brevity penalty, c is the length of a candidate and r is the corpus-level effective reference length, N is the maximum n-gram length, w_n is the n-gram weight, typically $w_n = \frac{1}{N}$.

Occasionally, the value of p_n is zero, resulting in zero BLEU-score, even if p_m are large for $m \in N, \text{m} \neq \text{n}$. To overcome this issue, many smoothing methods are proposed, e.g., Laplace-like smoothing or smoothing methods from NLTK (Natural Language Toolkit) [29].

2.7.2 ROUGE-L

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation - Longest Common Subsequence) [30] computes F-score based on Longest Common Subsequence.

$$P_{lcs} = \frac{LCS(X,Y)}{m}, \qquad R_{lcs} = \frac{LCS(X,Y)}{n}, \qquad ROUGE-L = \frac{(1+\beta^2)P_{lcs}R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}, \qquad \beta = \frac{P_{lcs}}{R_{lcs}}$$

where LCS is the length of the longest common subsequence, while X and Y are sentences with lengths of m and n, respectively.

2.7.3 METEOR

METEOR: (Metric for Evaluation of Translation with Explicit ORdering) [31] computes penalized F-score based on n-grams and their alignment. Alignment is a set of mappings between the same n-grams in the candidate and reference sentences. If we place one sentence on top of another and connect the same n-grams with lines, the preferred alignment would have the fewest intersections of these mapping lines. Chunk is the longest subsequence of n-grams that appears in both sentences.

$$P = \frac{k}{m}, R = \frac{k}{n}, p = 0.5\left(\frac{c}{k}\right)^3, \qquad METEOR = (1-p)\frac{10PR}{R+9P}$$

where k is the number of n-grams in the candidate sentence that are mapped, c is the number of chunks, m and n are lengths of candidate and reference sentences, respectively.

Several other methods exist for summarization evaluation (e.g., CIDER [32], Bert-Score [33]), however, the reviewed studies focus almost exclusively on BLEU, ROUGE-L and METEOR.

2.8 Taxonomy of Methods

To select studies for this survey, we searched the ACM and IEEE digital libraries using keywords such as source code, summarization, and deep learning, and extended the search via Google Scholar. We considered works published between 2016 and 2024. Papers were manually filtered based on the following criteria: (i) application of deep learning methods and (ii) focus on source code summarization. Works outside this scope (e.g., code generation or non-neural approaches) were excluded.

We note that relatively few works from 2024 satisfy these criteria, as recent research has largely shifted toward LLMs, which are considered separately in this survey. LLMs are not included as a separate category in the taxonomy, as they follow a different research paradigm focused on scaling and fine-tuning strategies rather than task-specific architectural design. Instead, they are discussed separately in Section 4.6. Overall, the final set includes 42 models and several baseline approaches. The reviewed studies apply different methods to improve Machine Translation baseline. We categorized them into three groups: leveraging code graphs, applying retrieval augmentation and multi-task learning. The taxonomy of methods is presented in Fig. 5. Note that studies may apply multiple methods from different categories on their models.

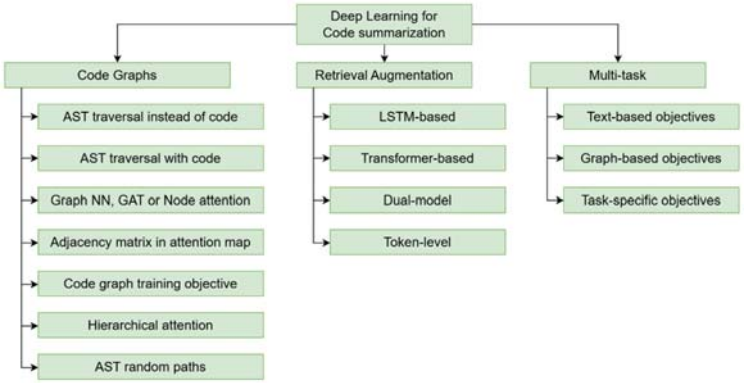

Fig. 5. Tree taxonomy of deep learning methods for code summarization.

The first group covers 25 studies that incorporate code structural information to facilitate code understanding. This structural input is generated from code graphs or code syntax features. Some

studies feed this structural input to the encoder input, some use this input as architectural constraints, while others define a training objective upon this input.

The second group unites eight studies that apply retrieval augmentation. A common approach retrieves the summary of the most similar code snippet from a codebase and fuses its token probabilities. Retrieval methodology differs from study to study and may be syntactic, relying on text search engines, or semantic, retrieving nearest neighbors in a latent space.

The third group contains thirteen studies that utilize additional training objectives beyond basic unidirectional language modeling. The list of these objectives may include text-code matching, contrastive learning, graph edge prediction, method name generation, etc. (see section 3.3).

Fig. 6 shows histograms of studies by year and categories. We can see a consistent raise of published papers from the first deep model applied to code summarization till the release of ChatGPT, and a rapid decrease afterwards. The raise corresponds to a vast diversity of modifications to model architectures with a minimum of data and evaluation change, which we discuss in more detail in Section 4.6. The left histogram shows how Transformers gradually replaced RNN models, while the right histogram presents the proportions of studies in every group.

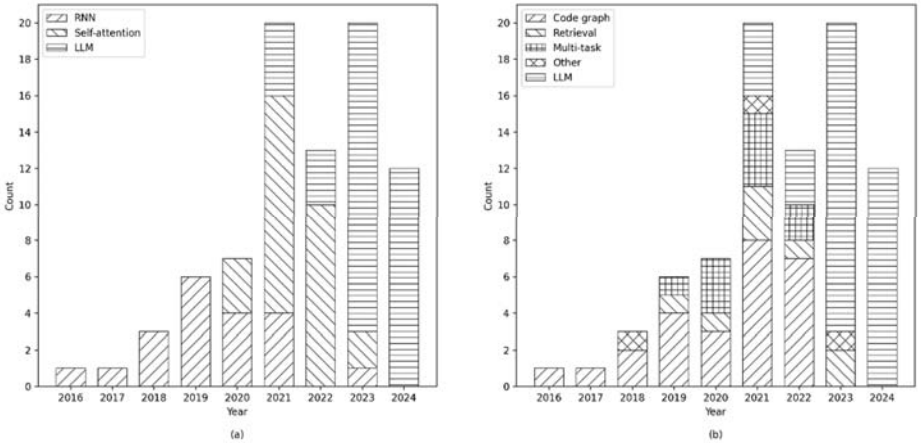

Fig. 6. Histograms of studies by base model architecture and category every year. (a) shows distribution of architectures for code summarization over the recent years, while (b) shows the distribution of models by main method category over the same period. We show 'LLM' for comparison, it includes large language models for code larger than 1B without code summarization being the only feature, while 'Self-attention' corresponds to code summarization models with self-attention.

Different categories of methods exhibit characteristic limitations. Sequence-based models often struggle with long-range dependencies and structural information. Graph-based approaches better capture code structure but may depend on accurate graph construction. Retrieval- or knowledge-augmented methods rely on external data quality. LLMs, while generally strong, may produce overly verbose or hallucinated summaries and are sensitive to prompting and decoding settings.

While conceptually different, these three groups aim a single purpose to improve the quality of code summarization. The following section describes these groups in detail.

3. Methods

In section 2.8 we categorized the approaches to improve code summarization into three categories: leveraging code graphs, retrieval augmentation and multi-task learning. In this section, we describe and discuss these approaches in detail.

To improve code summarization over machine translation baseline, a straightforward idea is to utilize a strict code syntax. All programming languages have syntax and formal grammar, and code must follow them to compile or run. "Code is often semantically brittle – small changes (e.g., swapping function arguments) can drastically change the meaning of code; whereas natural language is more robust in that readers can often understand text even if it contains mistakes" [34]. In natural language, different words contain independent meanings, and syntax connects these meanings. In code, operations and their order define what a function does, not the identifier names, although a good programming style involves proper naming.

Sequence-to-sequence models cannot process tree or graph structures directly as input, thus either code graphs should be linearized or the encoder should be adapted to graph input. Hu et al. [17] introduce Structure-based traversal (SBT) to linearize AST into a sequence. SBT adds parentheses to separate nodes, ensuring that the tree can be unambiguously reconstructed from the traversal.

SPT-Code [35], ast-attendgru [22] and ComFormer [36] also leverage SBT or its modified version, while mAST+GCN [37], AST-Trans [38], Rencos [39] and READSUM [40] apply a simpler Pre-order traversal. Pre-order traversal visits tree nodes by processing the current node, followed by its children from left to right. Pre-order traversal disallows reconstructing the initial tree.

Neural networks, including RNNs and Transformer, are statistical models and learn the structure from data rather than syntax or grammar rules. To facilitate code understanding, numerous studies introduce code structures into a model:

- pass linearized AST as input to the model [17, 37-38, 41];
- pass linearized AST to accompany the code tokens [22, 25, 36, 42-43];
- apply Graph Neural Networks (GNN), Graph Attention Networks (GAT), or Node-attention [44] to encode code graphs [20, 26, 37, 44-50];
- use code graph adjacency matrices to weigh or constrain the relations between tokens [38, [51-54];
- add code graph tasks as an additional training objective [55-56];
- introduce hierarchical attention [43, 51];
- encode AST random paths [56-57].

We describe these methods in more detail below.

3.1 Leveraging Code Graphs

3.1.1 Code graphs instead of code tokens

This method leaves the architecture unchanged and only replaces input code with AST traversal, as illustrated in Fig. 2 (b). The following models accept AST traversal instead of code tokens as input:

- DeepCom [17] is an attention-based LSTM Encoder-Decoder that accepts SBT of AST as input.
- AST-Trans [38] and Relative Structural Transformer (RST) [41] are Transformer models that consider code graphs in the attention map and accept pre-order traversal of AST.
- mAST+GCN [37] is a Transformer that accepts pre-order traversal of Graph Convolutional Network (GCN) – convolved AST with sibling edges added.

Table 2 shows BLEU scores for models that replace input code with AST traversal. We see that DeepCom surpasses Seq2seq+Attn model, while falling behind Transformer models. RST is a superior model on CSN, while AST-Trans outperforms Transformer on JCSD, PCSD.

Table 2. BLEU scores of models in the 'AST traversal instead of code' category.

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Java	CSN Python	CSN Go	CSN JavaScript	CSN PHP	CSN Ruby
Seq2seq+Attn [25]				35.51						
Transformer [41, 38]	44.58	25.77	21.1		18.2	18.19	18.08	14.61	23.12	13.9
mAST+GCN [37]	45.49	32.82								
DeepCom [58, 25]	39.75	20.78		38.22						
AST-Trans [38]	48.29	34.72								
RST [41]			21.7		18.63	17.93	18.62	14.96	23.77	14.81

3.1.2 Code graphs accompanying code tokens

This subgroup is similar to the previous subgroup with the only change that an AST traversal does not replace input code but extends it, as illustrated in Fig. 2 (c).

- Hybrid-DeepCom [25] is an extension of DeepCom model (see Section 3.1.1) that accepts both code and SBT of AST and splits camel case identifiers
- ast-attendgru [22] combines code with its AST traversal using separate GRU encoders, the GRU decoder then attends both encoder outputs via separate attention mechanisms.
- DRL-HAN [43] is an LSTM Encoder-Decoder with encoders for AST and CFG traversals, hybrid hierarchical attention and an actor-critic RL to overcome exposure bias.
- ComFormer [36] is a Transformer that fuses the code tokens with AST traversal. Yang et al. [36] showed that a model with a single encoder for code and AST outperforms both separate and shared encoder versions.
- Graph-to-Sequence Converter (G2SC) [42] is a rule-based traversal over PDG that can be integrated to any sequence processing model.

Table 3 presents the results of this category. DeepCom, ast-attendgru and DRL-HAN show controversial results on PCSD and Funcom. ComFormer is superior on EMSE-DeepCom.

3.1.3 Code graph encoding

Various methods exist to encode a code graph, some of them utilize GCN or Recurrent models, while others apply self-attention. Moreover, a graph encoder may replace, precede or accompany a baseline encoder, as illustrated in Fig. 2 (d-f).

Table 3. BLEU scores of models in the 'AST traversal with code' category.

Model	PCSD	Funcom	EMSE-Deepcom
DeepCom [43, 59, 25]	13.27	7.7	38.22
Transformer [36]			38.69
Hybrid-DeepCom [25]			39.51
ast-attendgru [59]	8.71	11.3	
DRL-HAN [43]	33.16		38.25
ComFormer [36]			48.44

Few studies encode code graph and then pass the resulting embeddings into a sequence-to-sequence model:

- mAST+GCN [37] (mentioned above, see Section 3.1.1) convolves AST with sibling relations using GCN and applies preorder traversal to prepare input to Transformer.

- BASTS [48] generates ASTs for CFG nodes, embeds them with TreeLSTM, and fuses the resulting embeddings into the encoder inputs of a Transformer.

In contrast, PDG Boosting Module [49] encodes PDG relations into the encoder outputs using Node attention to improve a Transformer.

Few other studies encode code graph using a separate encoder:

- code+gnn+GRU [47] combines code and AST using separate encoders and attention mechanisms, where AST encoder accepts nodes and edges instead of traversal which are encoded using GCN and following GRU layers.
- RL+Hybrid2Seq [20] leverages hybrid attention with TreeRNN over AST to integrate structural information together with the source code and reinforcement learning finetuning to overcome the teacher-forcing exposure.
- M2TS [45] is a Transformer with a GCN encoder for AST and an additional decoder layer for features fused from code and AST.
- CodeScribe [46] adds a GNN encoder to Transformer to encode AST triplet positions parallel to the vanilla encoder.
- The final vocabulary distribution is fused with the encoder outputs through Multi-source Pointer Generation Network to allow copying from code and AST.
- Re_Trans [50] is a retrieval augmented Transformer with a separate BiGRU+GCN encoder for semantic retrieval.

Other studies replace the initial encoder with an encoder adapted for structured input:

- Multi-way TreeLSTM [49] is an LSTM capable of encoding a tree of tokens to encode AST directly. While a typical LSTM has two inputs – hidden state and input token, Multi-way TreeLSTM has an additional third input – hidden state from another subtree.
- HGNN [26] is a retrieval augmented GNN that accepts Code Property Graph, combines it with retrieved similar graph and applies dynamic and static message passing – iteratively aggregates and transforms feature information from their neighbors, to capture both global and local relations before passing these embeddings to LSTM decoder.

Table 4 presents results for this category. Notably, NNGen is a strong baseline that CODE-NN, DeepCom and RL+Hybrid2Seq could not beat. CodeScribe is superior on JCSD and PCSD, with M2TS and mAST+GCN following close behind. Results on Funcom span two separate ranges of scores, suggesting different experimental set-ups. M2TS and code+gnn+GRU are leading models on Funcom. BASTS has a large margin over Transformer on EMSE-DeepCom and might compete with CodeScribe, though we cannot directly compare them due to absence of results on the same dataset.

3.1.4 Leveraging adjacency matrices

Many studies use code graphs indirectly in attention maps, as illustrated in Fig. 2 (g). Attention map is a matrix of attention weights, i.e., $softmax(\frac{QK^T}{\sqrt{d_k}})$. These models constrain the attention weights flow. Specifically, they scale an attention weight by the path length in a code graph or set to zero if not connected. Adjacency or path-length matrices become attention masks.

Structure-induced Transformer (SiT) [53] and SCRIPT [52] compute the adjacency matrices from Abstract Syntax Tree, Data Flow and Control Flow graphs, and set attention weights to zero if two tokens do not have a direct edge in these graphs.

AST-Trans [38] (mentioned above, see Section 3.1.1), CodeTransformer [54] and Relative Structural Transformer (RST) [41] extract path lengths, ancestor and sibling relations from AST to mask attention. Furthermore, RST considers a movement pattern (path direction, up or down the

tree), while CodeTransformer enriches the mask with PageRank algorithm. Note that AST-Trans and RST accept pre-order traversal of AST, while all other models in this section accept code tokens.

Structure Guided Transformer (SG-Trans) [51] leverages data flow, token and statement relations, but apply different adjacency matrices in attention heads instead of joining them into a single matrix.

Table 5 shows the performance of models with adjacency matrices in attention maps. We see that all models show similar results, though AST-Trans is moderately leading.

Table 4. BLEU scores of models in the 'Code graph encoding' category.

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CCSD	CSN Python
NNGen [26]		21.6			12.76	
CODE-NN [39, 49]	18.29	8.1				
DeepCom [58]	39.75	20.78				
Code2seq [47]			18.84			
Transformer [45, 48]	42.55	30.58	27.41	30.98		14.8
NCS [50]			13.77			
mAST+GCN [37]	45.49	32.82				
BASTS [48]				36.38		15.97
M2TS [45]	46.84	33.84	31.63			
CodeScribe [45]	49.19	35.11				
code+gnn+GRU [46]			19.7			
RL+Hybrid2Seq [39]	13.3	15.0				
Re_Trans [50]			16.83			
HGNN [26]		24.42			14.01	
Multi-way TreeLSTM [49]	20.15					

Table 5. BLEU scores of models in the 'Adjacency matrix in attention map' category.

Model	JCSD	PCSD
Transformer [51, 38, 53]	44.87	32.85
SG-Trans [51]	45.89	33.04
SCRIPT [52]	46.89	34.0
AST-Trans [38]	48.29	34.72
SiT [35]	45.3	33.8
CodeTransformer [38]	45.74	30.93

3.1.5 Code graph tasks as training objectives

Several studies introduce new training objectives derived from code graphs to assist the standard Next Token Prediction or Fill-In-the-Middle objectives, as illustrated in Fig. 2 (h).

GraphCodeBERT [55] takes code summary, code snippet and data flow variable sequence as input during pre-training. The latter allows to integrate two new training objectives: Data Flow Graph edge prediction and code token to data flow correspondence.

TreeBERT [56] is a pre-trained Transformer adapted to take random paths over AST as input with two training objectives: Tree Masked Language Modeling and Node Order Prediction. The former is an auto-regressive masked language modeling that has a strategy to mask nodes in the AST paths, while the latter distinguishes whether a path has a correct or shuffled order of nodes.

Table 6 shows results of models with a code graph training objective. We see that both GraphCodeBERT and TreeBERT outperform NCS, though TreeBERT has a moderate advantage.

Table 6. BLEU scores of models in the 'Code graph training objective' category.

Model	JCSD	PCSD
NCS [35]	44.5	32.3
GraphCodeBERT [35]	46.0	33.9
TreeBERT [35]	47.9	34.7

3.1.6 Hierarchical attention

Following studies explicitly integrate the hierarchical structure of code in terms of statements and syntax tokens into the model.

DRL-HAN [43] (mentioned above, see Section 3.1.2) is a LSTM Encoder-Decoder with additional LSTM encoders for AST and CFG traversals, hybrid hierarchical attention and an actor-critic reinforcement learning to overcome exposure bias. Hybrid hierarchical attention encodes tokens with an LSTM and applies token-level attention to form statement representations. These are processed by another LSTM with statement-level attention over code tokens, AST, and CFG traversals, and the resulting embeddings are fused for the decoder.

Structure Guided Transformer (SG-Trans) [51] (mentioned above, see Section 3.1.4) is a Transformer that leverages DFG adjacency matrix and hierarchical structure. Token relation appears between two tokens belonging to the same entity, e.g., identifier composed of few tokens. Similarly, a statement relation appears between two tokens belonging to the same code statement. These adjacency matrices appear in separate attention heads along with DFG matrix and standard attention.

Table 7 presents the results of hierarchical models. SG-Trans and DRL-HAN show similar results with a moderate improvement over Transformer. DeepCom has a low score on PCSD, suggesting a different experimental set-up, though it still falls behind DRL-HAN on EMSE-DeepCom.

Table 7. BLEU scores of models in the 'Hierarchical attention' category.

Model	JCSD	PCSD	EMSE-Deepcom
Transformer [51]	44.2	31.34	
DeepCom [43]		13.27	31.55
SG-Trans [51]	45.89	33.04	
DRL-HAN [43]		33.16	38.25

3.1.7 AST random paths

Distinct methods with the same functionality should have structural similarities reflected in their syntax trees. If one samples enough paths over both trees, a significant portion of paths will be shared. Code2seq [57] and TreeBERT [56] linearize AST by sampling random paths and feeding their embeddings to a sequence-to-sequence model, as illustrated in Fig. 2 (i).

Code2seq [57] embeds random paths over AST and passes these embeddings as input. First, a set of paths is sampled between random pairs of terminal nodes. Second, BiLSTM Encoder encodes each path into a single embedding, which is concatenated with the terminal node embeddings. Then, each resulting path representation is passed through a fully-connected layer. Finally, a LSTM decoder decodes the summary tokens while attending the path representations.

TreeBERT [56] (mentioned above, see Section 3.1.5) accepts random paths over AST as input and passes them through a linear layer before the Transformer encoder. First, a set of paths is sampled between a root and terminal nodes. Second, k nodes are masked in each path with higher probability

for nodes closer to the terminal. Third, node position embeddings are added to the node embeddings. Finally, nodes in each path are concatenated and fed into a linear layer to fit model dimensionality.

Table 8 shows results of models leveraging AST random paths. As we can see, Code2seq takes a place between CODE-NN and NCS baselines, while TreeBERT excels on JCSD and PCSD.

Leveraging code structural information unites these subgroups, though they follow different approaches, from replacing input with code graphs traversals or random paths to constraining attention flow by adjacency matrices and introducing new training objectives. In contrast to leveraging code graphs to transform or adjust input code with structural information, the following group retrieves similar examples from the codebase and utilizes them to improve the code summary.

Table 8. BLEU scores of models in the 'AST random paths' category.

Model	JCSD	PCSD	Funcom
CODE-NN [45]	26.07	16.58	16.25
NCS [35]	44.5	32.3	
Code2seq [45]	38.86	21.86	21.5
TreeBERT [35]	47.9	34.7	

3.2 Retrieval Augmented Models

Similar functions have similar summaries. The following studies leverage the code and summary of similar code snippets from the database, as illustrated in Fig. 7.

3.2.1 LSTM-based

HGNN [26] (mentioned above, see Section 3.1.3) is a retrieval augmented GNN with attention-based LSTM decoder. A similar code is retrieved by Lucene according to text edit distance. Code Property Graphs are constructed for both the input and retrieved code snippets, and then the retrieved code graph is fused into the input code graph. BiLSTM encodes the nodes and GNN applies dynamic and static message passing to capture both global and local relations. BiLSTM encodes the retrieved summary, the summary representations are scaled and concatenated to GNN output. LSTM decoder generates the final summary according to the resulting embeddings.

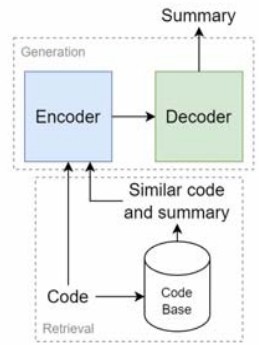

Fig. 7. High-level diagram of a simple retrieval augmented model.

Rencos [39] is a retrieval augmented attention-based LSTM Encoder-Decoder. During inference this model retrieves two samples from the database: semantically and syntactically similar to the requested code. Semantic similarity is defined as cosine similarity between code embeddings, which are pooled encoder representations. Syntactically similar code is retrieved by Lucene engine over the code AST traversals. These three code snippets are encoded and decoded in parallel, and their conditional probabilities are fused to predict the next token.

EditSum [60] is a retrieval augmented attention-based LSTM Encoder-Decoder. First, Lucene retrieves the similar code-summary pair from a database according to BM25 and a BiLSTM encodes the retrieved summary. The input and retrieved code snippets are compared, and words to insert are extracted from the input code and words to delete are extracted from the retrieved code. Embeddings of words to insert and words to delete are aggregated into context vectors by separate attention mechanisms, and concatenated into the "edit" vector. The LSTM decoder generates the summary based on this vector and attention to the retrieved summary encoder outputs.

3.2.2 Transformer-based

Re_Trans [50] (mentioned above, see Section 3.1.3) is a retrieval augmented Transformer with a separate BiGRU+GCN encoder for semantic retrieval. For retrieval, code is encoded with BiGRU and AST is encoded with GCN, the outputs are concatenated and the closest sample is retrieved from a database according to Euclidean distance. For generation, a Transformer with a GCN encoder for AST is used. During testing, a similar code and its summary are retrieved, then a trained discriminator decides whether input code is similar to the retrieved code. If the decision is positive, Re_Trans outputs the retrieved summary, otherwise the generative model generates a new summary.

READSUM [40] is a retrieval augmented Transformer with AST fusion and adaptive attention. The AST of the retrieved code is augmented into the AST of the input code through a multihead attention, and the pre-order traversal of the resulting tree is fused into the input code pre-order traversal. An encoder has an adaptive attention that considers AST shortest path distances in the attention map. A second encoder processes the retrieved summary, and finally, a decoder with a fusion network and dual copy mechanism generates the target summary.

3.2.3 Dual-model frameworks

REDCODER [61] is a retrieval augmented code generation and summarization framework that fine-tunes a retrieval model to extract relevant examples and augment them into the context of a pre-trained generator model.

CoaCor [62] is a RL-based framework for code retrieval that trains two models in the collaborative setting. To train an annotation model, authors replace its optimization criteria with a retrieval model as a rewarding function. This causes the generated annotations to focus on distinguishing code fragments rather than providing human understanding.

3.2.4 Token-level retrieval

Tram [63] is a token-level retrieval augmented Transformer with a separate GAT encoder for AST. Despite fitting the aforementioned Transformer group, it deserves a separate group because it is radically different. In contrast with sentence-level retrieval, Tram retrieves token distribution based on the last layer hidden representations and fuses it into the final next token prediction.

Table 9 shows BLEU scores of retrieval augmented models. Since there is no study presenting results on all datasets, we can only partially analyze results. We see that Transformer models NCS, READSUM, and Tram significantly outperform Rencos and NNGen on JCSD and PCSD, while LSTM-based EditSum excels on Funcom. HGNN and REDCODER only present results on CCSD and CSN surpassing NNGen and CodeBERT by 1.25 and 1.95-5.29 points, respectively.

Despite differences, all these methods retrieve information from a codebase to facilitate code summarization, while following methods leverage other tasks to improve code summarization.

3.3 Multi-task learning

Multi-task learning involves training a single model on multiple related tasks. This approach aims to boost the ability of the model to generalize and gain more knowledge from the data. Different

studies combine various tasks and training objectives in the multitask setting. The comprehensive list of tasks is listed in Table 10.

Text objectives contain functions applicable to any text and widely used in non-code language modeling. In contrast, graph objectives rely on graph structure to compute loss. Finally, code objectives focus on code syntax and grammar or high-level code tasks.

Table 9. BLEU scores of models in the 'Retrieval Augmentation' category.

Model	JCSD	PCSD	Funcom	CCSD	CSN Java	CSN Python
NNGen [39, 60, 26]	18.7	21.6	13.44	12.76		
CodeBERT [64]					17.65	19.06
NCS [50, 1]	44.58	32.52	13.77			
EditSum [60]			16.88			
HGNN [26]		24.42		14.01		
Rencos [39]	20.6	20.7				
READSUM [40]	47.75	34.01				
Re_Trans [50]			16.83			
REDCODER [61]					22.94	21.01
Tram [63]	48.14	35.74				

Table 10. Multi-task models and their training objectives. The abbreviations are explained in Section 3.3.

	Text Objectives						Graph Objectives					Code Objectives										
Model	ULM	MLM	MASS	NSP	RTD	CL	CAP	DEP	NA	TMLM	NOP	IT	MIP	TCM	CS	CG	MNG	ABF	ICM	GATM	ASR	CMG
CodeBERT [65]		✓			✓																	
CuBERT [66]		✓		✓																		
CugLM [67]	✓	✓		✓																		
GraphCodeBERT [55]		✓						✓	✓													
UniXcoder [64]	✓	✓	✓			✓									✓							
CodeT5 [68]			✓									✓	✓		✓	✓						
CodeT5+ [69]	✓		✓			✓								✓								
SPT-Code [35]			✓				✓										✓					
TreeBERT [56]										✓	✓											
DMACOS [59]															✓		✓					
Dual Model [58]															✓	✓						
T5-learning [70]															✓			✓	✓	✓		
CodeTrans [71]															✓						✓	✓

3.3.1 Text objectives

- *Unidirectional (Causal) Language Modeling (ULM)* is to predict the next token given previous context.
- *Masked Language Modeling (MLM)* is to predict a masked token given surrounding context.
- *Masked Sequence-to-sequence (MASS)* (Span Denoising, Masked Span Prediction) is to predict continuous spans, each of which was masked with a single mask token.
- *Next Segment Prediction (NSP)* is to detect whether the second segment logically continues the first one or belongs to a different context.
- *Replaced Token Detection (RTD)* is to discriminate original tokens from those masked and generated by an n-gram language model.
- *Contrastive learning (CL)* is to bring closer the code and summary embeddings of the same sample, while moving them away from all other in-batch "negative" samples.

3.3.2 Graph objectives

- *Code-AST Prediction (CAP)* is to detect whether a code and an AST correspond to each other or not.
- *DFG Edge Prediction (DEP)* is to predict whether two nodes are connected in DFG.
- *Node Alignment (NA)* is to detect whether a node and a variable correspond to each other.
- *Tree Masked Language Modeling (TMLM)* is to predict a masked token in a tree path based on the surrounding context.
- *Node Order Prediction (NOP)* is to detect whether the order of nodes in a tree path is corrupted or not.

3.3.3 Code objectives

- *Identifier Tagging (IT)* is to tag every token that is a part of code identifier.
- *Masked Identifier Prediction (MIP)* is to predict identifier names in the code, where each distinct identifier is masked with a unique token.
- *Text-Code Matching (TCM)* is to detect whether a summary corresponds to the given code or not.
- *Code Summarization (CS)* is to generate a natural language description for a given method body.
- *Code Generation (CG)* is to generate a method body for a given natural language description.
- *Method Name Generation (MNG)* is to generate a method name for a given method body.
- *Automatic Bug Fixing (ABF)* is to repair a code snippet with a bug.
- *Injection of Code Mutants (ICM)* is to inject a bug into a code snippet.
- *Generation of Assertions in Test Methods (GATM)* is to generate an assert statement in a test method for a given functionality.
- *API Sequence Recommendation (ASR)* is to generate several API calls relevant to a natural language query.
- *Commit Message Generation (CMG)* is to generate a commit message for given git commit changes.

CodeBERT [65], CuBERT [66], and CugLM [67] rely solely on text objectives. CodeBERT is a RoBERTa pre-trained on unimodal and bimodal data with MLM and RTD objectives in six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go; while CuBERT is a RoBERTa pre-trained with MLM and NSP objectives in Python only. In contrast, CugLM is a decoder model pre-trained on Java code. GraphCodeBERT [55] (mentioned above, see Section

3.1.5) is an improvement over CodeBERT that optimizes MLM, DEP and NA objectives during pre-training. To implement DEP, GraphCodeBERT has additional input tokens corresponding to DFG nodes. NA is then required for this setting to align code tokens with DEP nodes, input is further extended with every variable mention, and the model should match mentions with nodes. CodeBERT, CuBERT and GraphCodeBERT are encoder models which means they are designed for sequence understanding, not token generation. To use encoder models in token generation, e.g. code summarization, authors use their weights to initialize the encoder of an encoder-decoder model and then train in a sequence-to-sequence manner.

In contrast to all other models, UniXcoder [64] can operate in encoder, decoder and encoder-decoder setups, as listed in Table 11. While a decoder under the hood, UniXcoder was trained on encoding, decoding, and transforming examples with different attention masks. Specifically, UniXcoder applies cross-attention for encoding with MLM objective, causal attention for decoding with ULM objective, and a hybrid attention for encoder-decoder setup with MASS objective. Apart from that, the model adapts representation learning: multi-modal CL and CS.

Table 11. Multi-task models by architectures.

	Architecture		
Model	Encoder	Decoder	Encoder-Decoder
CodeBERT [65]	✓		
CuBERT [66]	✓		
CugLM [67]		✓	
GraphCodeBERT [55]	✓		
UniXcoder [64]		✓	
CodeT5 [68]			✓
CodeT5+ [69]			✓
SPT-Code [35]			✓
TreeBERT [56]			✓
DMACOS [59]			✓
Dual Model [58]			✓
T5-learning [70]			✓
CodeTrans [71]			✓

CodeT5 [68] introduced an identifier-aware pretraining to better handle code-specific tokens by two new code objectives: IT and MIP. Furthermore, CS and CG helped to improve semantic alignment of code and comments. CodeT5+ [69] expanded on CodeT5 with a scalable architecture supporting encoder-only, decoder-only, and encoder-decoder tasks. It incorporated ULM and MASS during the first stage, and ULM, CL, and TCM during the second stage.

T5-learning [70] leveraged T5 training framework to handle both code and natural text for tasks by pretraining on four code objectives, including CS, ABF, ICM, and GATM. SPT-Code [35] aims to enhance code representation learning by leveraging the seq2seq pretraining paradigm, inspired by T5-learning, to unify the encoder and decoder training for tasks in code understanding and generation. To capture both syntactic and semantic properties of code, SPT-Code adopts MASS, CAP, and MNG objectives.

TreeBERT [56] (mentioned above, see Sections 3.1.5, 3.1.7) represented program code by integrating its hierarchical and syntactic structures through ASTs. TreeBERT adapted to AST paths input with two training objectives TMLM and NOP to capture deep syntactical and semantic information more effectively than traditional linearized models. DMACOS [59] strengthens code summarization through new objectives exploiting the relationship between method names and

method implementations. By employing MNG as an auxiliary task to CS, and refining its outputs through a two-pass mechanism, it achieves superior performance.

Dual model [58] harnessed the complementary nature of CS and CG within a dual learning framework of an Attentional LSTM Encoder-Decoder. By jointly training these tasks, using probabilistic and attention-based constraints, it enhances their mutual performance. On the other hand, CodeTrans [71] utilized a T5 model trained on code tasks in transfer learning, single and multi-task settings on CS, ASR, and CMG objectives.

Tables 12, 13 and 14 show BLEU scores of multi-task models. Since there is no study presenting results on all datasets, we can only partially analyze results. Among all multi-task models evaluated on JCSD and PCSD, SPT-Code leveraging text, graph and code objectives is superior. GraphCodeBERT and CugLM show similar scores beating NCS baseline among text-objective models, while the gap in Transformer and CuBERT scores indicate a different evaluation setup. TreeBERT is close to SPT-Code score and beats GraphCodeBERT among graph-objective models. Finally, T5-learning and NCS perform similarly, while slightly surpassing Dual Model and DeepCom among code-objective models. However, we see significant gaps in PCSD and Funcom scores which are not reflected in JCSD scores, suggesting different evaluation setups.

CodeSearchNet results show that CodeT5+ excels across models with text and code objectives, while no model with graph objectives is evaluated on this dataset. CodeT5 and UniXcoder show comparable results beating CodeBERT and Transformer on CSN.

Reviewed models leverage a wide range of text, graph, and code objectives to implement multi-task learning. Multi-task learning aims to generalize model capabilities by optimizing the model weights on several tasks jointly. These tasks complement each other and improve the resulting model on a subset of tasks.

4. Open Issues

This section discusses open challenges and future research directions in code summarization.

Table 12. BLEU scores of models with text objectives.

Model	JCSD	PCSD	CSN Java	CSN Python	CSN Go	CSN JavaScript	CSN PHP	CSN Ruby
Transformer [56, 65]	13.32		16.26	15.81	16.38	11.59	22.12	11.18
NCS [35]	44.5	32.3						
CodeBERT [64, 65]			17.65	19.06	18.07	14.9	25.16	12.16
UniXcoder [64]			20.31	19.13	19.07	15.85	26.54	14.87
GraphCodeBERT [35]	46.0	33.9						
CuBERT [56]	17.41							
CugLM [35]	46.1	34.3						
SPT-Code [35]	49.1	36.1						
CodeT5 [68]			20.31	20.01	19.56	16.16	26.03	15.24
CodeT5+ [69]			20.53	20.16	19.6	16.27	26.78	15.51

Table 13. BLEU scores of models with graph objectives.

Model	JCSD	PCSD
NCS [35]	44.5	32.3
GraphCodeBERT [35]	46.0	33.9
TreeBERT [35]	47.9	34.7
SPT-Code [35]	49.1	36.1

Table 14. BLEU scores of models with code objectives.

Model	JCSD	PCSD	Funcom	CSN Java	CSN Python	CSN Go	CSN JavaScript	CSN PHP	CSN Ruby
NCS [35]	44.5	32.3							
CodeBERT [64]				17.65	19.06	18.07	14.9	25.16	12.16
DeepCom [58, 59]	39.75	20.78	7.7						
UniXcoder [64]				20.31	19.13	19.07	15.85	26.54	14.87
T5-learning [35]	44.2	34.1							
SPT-Code [35]	49.1	36.1							
CodeT5 [68]				20.31	20.01	19.56	16.16	26.03	15.24
CodeT5+ [69]				20.53	20.16	19.6	16.27	26.78	15.51
Dual Model [58]	42.39	21.8							
DMACOS [59]		13.2	12.9						

4.1 Introducing context

Most models in this survey support context lengths up to 512 tokens (≈50 lines of code). While they generate reasonable summaries for many functions, this input is often insufficient for practical use cases. Models typically lack information about callee bodies, argument descriptions and valid values, caller requirements, project comment style, and tasks beyond the analyzed function. Including such context could improve summarization quality.

Haque et al. [72] improved code summarization by incorporating file context. They used a separate RNN to encode file context and appended the final state of each function to the decoder attention. Models with file context consistently outperformed baselines by 1.4–6.7% on the Funcom dataset.

Zhang et al. [73] proposed a method for summarizing overriding methods. Their model uses three BiGRU encoders to process method code, overridden method comments, and child class names with auxiliary features such as keyword or overlap indicators. A BiGRU decoder with attention and a pointer network then generates the comment. The model outperformed Seq2Seq, DeepCom-Hybrid, and two IR baselines by over 4 points in BLEU-4, METEOR, and ROUGE-L.

Although adding context improves results, increasing context length introduces challenges. RNN-based models tend to forget earlier inputs and are difficult to train on long sequences. Transformer-based models have quadratic time and memory complexity with respect to input length, though optimization techniques can reduce these costs. Few LLMs support contexts up to 1M tokens, but performance often degrades when the context exceeds hundred thousand tokens.

Common approaches to handle long context in code summarization include recurrent models, sparse or dilated attention mechanisms, and retrieval-based methods. Recurrent models process sequences token by token while maintaining internal state without additional memory. Sparse or dilated attention reduces Transformer complexity by computing attention over only a subset of tokens. Retrieval-Augmented Generation addresses large contexts by retrieving relevant embeddings representing sentences or documents, allowing the model to focus on the most relevant information. Recent advances in large language models have also extended context windows to up to a million tokens, enabling more comprehensive code summarization.

4.2 Class-level summarization

In recent studies, class-level code summarization rarely appears due to several reasons. First, there are fewer classes than functions, and thus less data to train models. Second, one class may inherit another one, thus one needs to consider all its super-classes in summarization. Finally, classes are longer than functions ranging from a single overriding method up to tens of methods and hundreds of lines, leading to context length problem discussed above.

Malhotra and Chhabra [7] generate class summaries by detecting micro-patterns and inter-class relations, and filling templates with them without machine learning. The generated summaries are longer than usual comments and contain structural information about the class. However, this approach cannot extract semantics and was only tested on four programs.

Li et al. [6] developed a neural class summarization model. To overcome long context issues, they reduce code and AST by removing all method bodies. The model consists of two LSTM encoders for code and AST traversal, and a LSTM decoder with attention and switch network. Switch network is composed of three linear layers: to transform lexical and syntactical contexts, and to compute their weights. The model outperforms CODE-NN, DeepCom, ast-attendgru, Transformer, PLBART [74] by 0.2-2.6 points in BLEU, METEOR, ROUGE-L on hybrid and cross-project datasets.

Du et al. [75] manually crafted a benchmark for class-level code generation. They designed it to evaluate Large Language Models on generation of entire classes mitigating the pretraining phase data leakages by manual crafting. Despite being designed for generation, this benchmark also allows to evaluate class summarization by swapping inputs and targets.

4.3 Dealing with comment types

Comments describe code to facilitate understanding but rarely provide deep insight. Most comments only describe code from one point of view. Chen et al. [76] categorized function comments into six types: “what”, “why”, “how-it-is-done”, “property”, “how-to-use”, and “others”. The authors proposed a composite model, which selects the best model for each comment category. The category is predicted by a classifier trained on manually labeled samples. The authors show that different models are better at generating different categories of comments. The composite model showed consistent improvements by 8-14%.

There is no single guideline for writing code comments, and programmers highlight what they think is important, which leads to the aforementioned comment types. However, we expect a model to describe the code functionality, yet we train it on the data with various comment types. A possible solution would be to have a comment type classifier, and only train the model on data of required comment type. However, recent code large language models understand the required comment type well from a prompt.

4.4 Question Answering over Code

The problem of various comment types arises due to different user needs, and the solution is to develop models capable of answering questions over code rather than summarizing it. This approach focuses on providing precise, user-specific insights about code snippets rather than generating generic comments. Comments answer different questions, and after reading such a biased comment a programmer may still have questions about the code snippet. Since a deep learning model can generate a comment, an improved version of this model might generate a user-specific comment. Using such a model, programmers could ask specific questions about code and get answers instantly to build versatile understanding [77].

Many Large Language Models for code have been released that are able to answer questions, e.g., Copilot [78], Phind [79], CodeLlama [80]. These models can summarize code; however, the answers are broad and detailed when using a straightforward prompt, in contrast with usual code comments. Moreover, Phind can access search results while answering questions, benefiting from dependencies documentation and question-answering forums.

4.5 Evaluation issues

To improve a code summarization model, one needs a robust metric to compare it with the baselines. The ground truth evaluation is human labeling; however, conducting a human evaluation for every trained checkpoint is infeasible. Therefore, automatic metrics are used; most popular of them are

BLEU, METEOR, and ROUGE, however, more researchers start evaluating models with other models: BERTScore [33] or reference-free metrics [81].

BLEU, METEOR, and ROUGE-L are n-gram based and suffer from a corresponding deficiency – they require rigorous matches. These automatic metrics do not recognize synonyms or synonymous constructs, while a paraphrased comment can be as good as an original one. They are sensitive to usage of words from the original comment, including prepositions and articles. As a result, these metrics stop correlating with human evaluation after some threshold.

Another problem concerning BLEU metric specifically is a variety of implementations resulting in different scores. Shi et al. [18] identified six variants, conducted human evaluation and found that BLEU-DC correlates most with human evaluation. The differences among the BLEU variants could affect the validity of the experiments and conclusions.

4.6 Advances of Large Language Models

The emergence of LLMs has reshaped research in deep learning for code. OpenAI's Codex [82] (2021) demonstrated strong coding abilities (70.2% pass@100 on HumanEval). Subsequent models—including AlphaCode [83], CodeLlama [80], StarCoder [84-85], PaLM-Coder [86], WizardCoder [87], DeepSeek-Coder [88], OpenCodeInterpreter [89], and Qwen2.5-Coder [90] — have achieved substantial gains on benchmarks such as HumanEval through scaling, instruction tuning, synthetic data generation, and execution-feedback refinement.

Research in code summarization has declined, as LLMs provide strong out-of-the-box summarization and structure-following abilities. Traditional approaches focused on architectural innovation under fixed datasets and model sizes, whereas LLM-based research emphasizes scaling and instruction tuning. However, systematic comparison between LLMs and specialized models remains inherently challenging. N-gram-based metrics (e.g., BLEU, ROUGE) are biased toward lexical overlap and may underestimate semantically correct but paraphrased LLM outputs (Section 4.5). Additionally, differences in model scale and training data—potentially including overlap with evaluation benchmarks—further complicate fair assessment. Existing studies therefore report mixed results: Sun et al. [91] reported specialized models often achieve higher scores on n-gram metrics, while LLMs tend to align better with human evaluation [92]. Overall, these factors highlight the lack of standardized and fair evaluation protocols for code summarization.

5. Related Work

Our survey identifies the "bricks" of code summarization—the design building blocks composing reviewed models. We build a taxonomy of these blocks and show how different models apply the same approach, or how a single model combines multiple approaches.

Sharma et al. [93] review approaches and datasets across twelve Machine Learning for Software Engineering (ML4SE) tasks, whereas our survey focuses specifically on code summarization. They analyze machine learning models from decision trees to Transformers, while we investigate sequence-to-sequence deep learning models. Additionally, Sharma et al. discuss general patterns in data collection, feature extraction, and model training, whereas we highlight individual model differences and systematize them into a unified taxonomy.

Song et al. [94] discuss code summarization models from vector space models to deep learning approaches. Zhu and Pan [95] and Yang et al. [96] provide systematic literature reviews with high-level method categorization, grouping all deep learning approaches together. However, these works were published in 2019, and most recent models have appeared since then.

Zhang et al. [97] review code summarization including source code modeling, comment generation, and quality evaluation. They describe comment generation approaches from template-based methods to deep learning models, focusing on common patterns rather than constructing a full taxonomy or highlighting individual model features.

6. Conclusion

In this survey, we addressed the task of code summarization and focused on deep learning methods. We reviewed various approaches to improve the quality of summarization and developed a taxonomy of these methods, categorizing them into leveraging code graphs, retrieval augmentation, and multi-task learning. We collected results from all studies into Tables 15, 16, and 17 for BLEU, ROUGE, and METEOR, respectively.

We discussed the open issues and future research directions in the field, including the potential of large language models in code summarization. Our taxonomy provides a structured overview of the methods used to improve code summarization, which can serve as a foundation for further research and development.

Table 15. BLEU scores of discussed models on evaluation datasets from original papers.

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
Baselines											
NNGen [26, 60, 39]	18.7	21.6	18.89								12.76
CODE-NN [58, 45, 43, 23, 48]	27.6	17.36	16.25	36.51	15.37						8.24
Seq2seq+Attn [20, 25, 17, 65]	35.5	1.33		35.51	15.93	9.64	10.21	13.98	15.09	21.08	
Transformer [53, 45, 36, 44, 41]	44.87	32.85	27.41	38.69	18.19	13.9	14.61	18.08	18.2	23.12	24.26
NCS [1, 45, 51, 35]	45.15	32.52	28.26		9.4				10.7		
RoBERTa [64]					18.14	11.7	11.9	17.72	16.47	24.02	
CodeBERT [53, 35, 44, 65]	45.2	33.47			19.06	12.16	14.9	18.07	17.65	25.16	22.98
Code graph models											
AST+GCN [37]	45.3	32.41									
AST-Trans [38]	48.29	34.72									
BASTS [48]				36.38	15.97						
Code2Seq (Transformer) [38]	35.08	29.79									
Code2seq [45, 43]	38.86	21.86	21.5	19.96							
CodeBERT+PDG [44]	40.75										23.41
CodeScribe [46]	49.19	35.11									
CodeTransformer [38]	45.74	30.93									
ComFormer [36]				48.44							
DRL-HAN [43]		33.16		38.25							
DeepCom [58, 59, 25]	40.51	20.78	14.21	38.22							
Graph-Transformer [38]	44.68	32.55									
GraphCodeBERT [35, 64]	46.0	33.9			18.06	12.39	14.81	18.41	19.0	25.59	
Hybrid-DeepCom [45, 25, 48]	42.26	21.14	20.74	39.51	10.24						
M2TS [45]	46.84	33.84	31.63								
MASS pre-training [56]	18.92										
Multi-way TreeLSTM [49]	20.15										
RL+Hybrid2Seq [58, 59, 48, 26]	38.22	19.28	12.1	20.16	14.09						8.42
RST [41]			21.7		17.93	14.81	14.96	18.62	18.63	23.77	

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
SCRIPT [52]	46.89	34.0									
SG-Trans [51, 45]	45.89	33.04	30.49								
SiT [53, 44]	45.76	34.11									25.0
SiT+PDG [44]	46.86										27.63
TL-CodeSum [58, 21]	41.98	15.36									
Transformer+PDG [44]	46.07										26.83
Tree2seq+Attn [58]	37.88	20.07									
TreeBERT [35]	47.9	34.7			11.1	7.4	10.3	17.1	13.8	18.0	
ast-attendgru [59, 22, 48]	40.82	8.71	19.67	27.5	14.47						
code+gnn+BiLSTM [47]			19.93								
code+gnn+GRU [45]	40.24	23.48	22.13								
mAST+GCN [37]	45.49	32.82									
Retrieval models											
EditSum [60]			16.88								
HGNN [26]		24.42									14.01
READSUM [40]	47.75	34.01									
REDCODER [61]					21.01				22.94		
Re Trans [50]			16.83								
Rencos [35, 50, 48, 26]	36.9	28.9	12.3	32.9	15.03						12.59
Tram [63]	48.14	35.74									
Multi-task models											
CodeT5-base [68]					20.01	15.24	16.16	19.56	20.31	26.03	
CodeT5-base+multi-task [68]					20.36	15.69	16.24	19.79	20.46	26.09	
CuBERT [59]	17.41										
CugLM [38]	46.1	34.3			11.4	7.4	10.0	17.0	13.2	17.8	
DMACOS [59]		13.2	12.9								
Dual Model [58, 59, 26]	42.39	21.8	12.3								9.61
PLBART [74]					19.3	14.11	15.56	18.91	18.45	23.58	
SPT-Code [35]	49.1	36.1			12.8	8.4	12.8	18.8	16.8	20.4	
T5-learning [35]	44.2	34.1			9.7	7.8	9.0	15.3	13.5	18.4	
UniXcoder [64]					19.13	14.87	15.85	19.07	20.31	26.54	

Table 16. ROUGE scores of discussed models on evaluation datasets from original papers.

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
Baselines											
NNGen [39, 60, 26]	36.3	40.2	38.57								23.93
CODE-NN [58, 45, 48, 26]	41.1	37.81	38.24	33.18	17.02						22.28
Seq2seq+Attn [20, 36]		30.83		51.84							
Transformer [53, 45, 36, 44, 48]	54.95	46.93	45.47	55.26	17.91						26.5
NCS [35, 45, 51]	54.84	46.8	47.14		17.0				25.9		
Code graph models											
AST+GCN [37]	54.45	46.35									
AST-Trans [38]	55.85	47.77									
BASTS [48]				47.85	22.72						

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
Code2Seq(Transformer) [38]	42.77	40.59									
Code2seq [45, 47, 43]	51.85	38.13	49.69	23.56							
CodeBERT+PDG [44]	52.85										29.45
CodeScribe [46]	59.59	50.46									
CodeTransformer [38]	54.96	43.67									
ComFormer [36]				63.25							
DRL-HAN [43]		46.23		37.19							
DeepCom [58, 60, 36]	52.67	37.35	42.45	52.18							
Graph-Transformer [38]	54.98	39.66									
GraphCodeBERT [35]	56.5	50.4			22.1	18.0	24.9	30.1	33.5	42.8	
Hybrid-DeepCom [45, 36, 48]	51.36	37.8	40.31	54.33	12.41						
M2TS [45]	57.87	47.92	49.31								
Multi-way TreeLSTM [49]	32.59										
RL+Hybrid2Seq [39, 59, 48, 58, 26]	51.91	42.2	40.7	29.22	17.77						28.64
SCRIPT [52]	56.69	48.15									
SG-Trans [51, 45]	55.79	47.01	47.28								
SiT [53, 44]	55.58	48.35									26.83
TL-CodeSum [39, 58]	52.25	35.3									
Transformer+PDG [44]	56.68										30.14
Tree2seq+Attn [58]	51.5	35.64									
TreeBERT [35]	56.6	50.5			21.6	15.5	22.0	30.1	34.3	42.4	
ast-attendgru [59, 47, 48]		34.8	49.75	35.32	15.76						
code+gnn+BiLSTM [47]			56.08								
code+gnn+GRU [45, 47]	52.83	39.85	57.15								
mAST+GCN [37]	54.82	46.81									
Retrieval models											
EditSum [60]			53.17								
HGNN [26]		39.91									30.89
READSUM [40]	59.34	50.49									
Re Trans [50]			42.64								
Rencos [39, 60, 48, 35, 26]	51.2	47.5	46.35	42.84	19.4						28.45
Tram [63]	57.89	49.63									
Multi-task models											
CodeBERT [53, 35, 44]	54.7	49.35			21.0	17.2	22.2	30.4	32.2	42.7	29.06
CugLM [35]	55.8	49.7			22.0	15.3	21.8	29.7	31.7	41.9	
DMACOS [59]		38.5	42.3								
Dual Model [58, 59, 26]	53.61	39.45	41.2								26.4
SPT-Code [35]	58.2	52.0			27.1	20.0	28.8	38.1	36.3	43.7	
T5-learning [35]	53.9	50.1			16.3	15.5	19.6	25.9	34.2	35.6	

Table 17. METEOR scores of discussed models on evaluation datasets from original papers.

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
Baselines											
NNGen [40]	15.0	17.1	29.97								11.58
CODE-NN [26, 60, 25, 45, 48]	13.97	13.68	29.35	18.04	7.25						
Seq2seq+Attn [20, 36]		6.49		24.53							

Model	JCSD	PCSD	Funcom	EMSE-Deepcom	CSN Python	CSN Ruby	CSN JavaScript	CSN Go	CSN Java	CSN PHP	CCSD
Transformer [53, 45, 36, 51, 26, 48]	26.83	19.86	16.47	29.06	7.6						12.02
NCS [51, 50, 35]	27.46	19.96	21.05		4.9				12.9		
CodeBERT [53, 35]	26.2	21.69			6.0	7.5	10.5	11.7	16.4	19.3	
Code graph models											
AST+GCN [37]	27.26	19.77									
AST-Trans [38]	30.94	20.71									
BASTS [48]				24.12	9.79						
Code2Seq (Transformer) [38]	21.69	16.73									
Code2seq [51, 45, 43]	23.76	13.81	10.23	10.17							
CodeScribe [46]	32.27	23.48									
CodeTransformer [38]	29.65	18.42									
ComFormer [36]				34.18							
DRL-HAN [43]		9.43		20.01							
DeepCom [59, 60, 36, 58]	23.06	13.4	31.79	25.18							
Graph-Transformer [38]	29.29	19.58									
GraphCodeBERT [35]	26.5	22.1			7.4	8.1	11.4	12.5	17.8	20.2	
Hybrid-DeepCom [45, 36, 48]	24.86	9.64	9.41	27.38	5.03						
M2TS [45]	28.93	21.83	20.06								
Multi-way TreeLSTM [49]	17.52										
RL+Hybrid2Seq [39, 59, 48, 58, 26]	22.75	17.9	18.6	13.15	7.8						11.73
SCRIPT [52]	28.48	20.84									
SG-Trans [51, 45]	27.85	20.52	18.81								
SiT [53]	27.58	21.11									
TL-CodeSum [39, 58]	23.73	13.6									
Tree2seq+Attn [58]	22.55	8.96									
TreeBERT [35]	27.2	23.0			7.2	6.9	10.5	12.8	17.1	19.1	
ast-attendgru [59, 60, 48]	24.54	16.5	36.99	16.44	6.42						
code+gnn+BiLSTM [50]			20.12								
code+gnn+GRU [45]	24.25	12.31	11.62								
mAST+GCN [37]	27.17	20.12									
Retrieval models											
EditSum [60]			42.93								
HGNN [26]		19.48									14.5
READSUM [40]	31.85	22.86									
Re Trans [50]			22.15								
Rencos [39, 60, 48, 35, 26]	26.5	21.1	34.53	21.52	8.73						13.21
Tram [63]	29.38	21.87									
Multi-task models											
CugLM [35]	26.3	21.6			7.5	6.8	10.4	12.7	16.1	18.2	
DMACOS [59]		19.4	20.1								
Dual Model [59, 58, 26]	25.77	17.5	19.2								11.87
SPT-Code [35]	32.4	26.9			14.2	11.1	17.2	16.6	20.6	20.6	
T5-learning [35]	26.5	22.6			5.5	6.7	9.2	11.0	16.8	17.9	

Список литературы / References

[1]. Ahmad W.U., Chakraborty S., Ray B., Chang K.-W. A Transformer-based Approach for Source Code Summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, pp. 4998-5007. Association for Computational Linguistics, 2020. DOI: 10.18653/v1/2020.acl-main.449.

[2]. Iyer S., Konstas I., Cheung A., Zettlemoyer L. Summarizing Source Code using a Neural Attention Model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, Berlin, Germany, August 7-12, 2016, Volume 1: Long Papers. The Association for Computer Linguistics, 2016. DOI: 10.18653/v1/p16-1195.

[3]. Jiang S., Armaly A., McMillan C. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 135-146. IEEE Computer Society, 2017. DOI: 10.1109/ASE.2017.8115626.

[4]. Loyola P., Marrese-Taylor E., Matsuo Y. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers, pp. 287-292. Association for Computational Linguistics, 2017. DOI: 10.18653/v1/P17-2045.

[5]. Liu Z., Xia X., Hassan A.E., Lo D., Xing Z., Wang X. Neural-Machine-Translation-Based Commit Message Generation: How Far Are We? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp. 373-384. ACM, 2018. DOI: 10.1145/3238147.3238190.

[6]. Li M., Yu H., Fan G., Zhou Z., Huang J. ClassSum: a deep learning model for class-level code summarization. Neural Computing and Applications, 2023, vol. 35, no. 4, pp. 3373-3393. DOI: 10.1007/s00521-022-07877-z.

[7]. Malhotra M., Chhabra J. Class Level Code Summarization Based on Dependencies and Micro Patterns. 2018, pp. 1011-1016. DOI: 10.1109/ICICCT.2018.8473199.

[8]. Makharev V., Ivanov V. Code Summarization Beyond Function Level. 2025. DOI: 10.48550/arXiv.2502.16704.

[9]. Ma Y., Yang Q., Cao R., Li B., Huang F., Li Y. Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration. 2025. DOI: 10.48550/arXiv.2406.01422.

[10]. Lomshakov V., Podivilov A., Savin S., Baryshnikov O., Lisevych A., Nikolenko S. ProConSuL: Project Context for Code Summarization with LLMs. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, Miami, Florida, US, November 2024, pp. 866-880. Association for Computational Linguistics, 2024. DOI: 10.18653/v1/2024.emnlp-industry.65.

[11]. GitHub. Available at: https://github.com/. Accessed 05.06.2026.

[12]. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A.N., Kaiser L., Polosukhin I. Attention Is All You Need. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA, December 4-9, 2017, pp. 5998-6008, 2017.

[13]. Hochreiter S., Schmidhuber J. Long Short-Term Memory. Neural Computation, 1997, vol. 9, no. 8, pp. 1735-1780.

[14]. Cho K., van Merrienboer B., Gulcehre C., Bahdanau D., Bougares F., Schwenk H., Bengio Y. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, Doha, Qatar, October 25-29, 2014, pp. 1724-1734. ACL, 2014.

[15]. Schuster M., Paliwal K.K. Bidirectional Recurrent Neural Networks. IEEE Transactions on Signal Processing, 1997, vol. 45, no. 11, pp. 2673-2681.

[16]. Luong T., Pham H., Manning C.D. Effective Approaches to Attention-based Neural Machine Translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015, pp. 1412-1421. The Association for Computational Linguistics, 2015. DOI: 10.18653/V1/D15-1166.

[17]. Hu X., Li G., Xia X., Lo D., Jin Z. Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension, ICPC 2018, pp. 200-210. IEEE, 2018.

[18]. Shi E., Wang Y., Du L., Chen J., Han S., Zhang H., Zhang D., Sun H. On the Evaluation of Neural Code Summarization. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 1597-1608. ACM, 2022. DOI: 10.1145/3510003.3510060.

[19]. Barone A.V.M., Sennrich R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. CoRR, 2017, abs/1707.02275. DOI: 10.48550/arXiv.1707.02275.
[20]. Wan Y., Zhao Z., Yang M., Xu G., Ying H., Wu J., Yu P.S. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp. 397-407. ACM, 2018. DOI: 10.1145/3238147.3238206.
[21]. Hu X., Li G., Xia X., Lo D., Lu S., Jin Z. Summarizing Source Code with Transferred API Knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sweden, July 13-19, 2018, pp. 2269-2275. ijcai.org, 2018. DOI: 10.24963/ijcai.2018/314.
[22]. LeClair A., Jiang S., McMillan C. A neural model for generating natural language summaries of program subroutines. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 795-806. IEEE/ACM, 2019. DOI: 10.1109/ICSE.2019.00087.
[23]. UCI Source Code Data Sets. Available at: https://ics.uci.edu/~lopes/datasets/. Accessed 05.06.2026.
[24]. Husain H., Wu H.-H., Gazit T., Allamanis M., Brockschmidt M. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. CoRR, 2019, abs/1909.09436. DOI: 10.48550/arXiv.1909.09436.
[25]. Hu X., Li G., Xia X., Lo D., Jin Z. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. Empirical Software Engineering, 2020, vol. 25, no. 3, pp. 2179-2217. DOI: 10.1007/s10664-019-09730-9.
[26]. Liu S., Chen Y., Xie X., Siow J.K., Liu Y. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.
[27]. Shi L., Mu F., Chen X., Wang S., Wang J., Yang Y., Li G., Xia X., Wang Q. 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, ESEC/FSE 2022, Singapore, Singapore, pp. 107-119. Association for Computing Machinery, 2022. DOI: 10.1145/3540250.3549145.
[28]. Papineni K., Roukos S., Ward T., Zhu W.J. BLEU: a Method for Automatic Evaluation of Machine Translation. 2002. DOI: 10.3115/1073083.1073135.
[29]. NLTK Source. Available at: https://github.com/nltk/nltk. Accessed 05.05.2026.
[30]. Lin C.-Y. ROUGE: A Package for Automatic Evaluation of Summaries. In: Text Summarization Branches Out, Barcelona, Spain, July 2004, pp. 74-81. Association for Computational Linguistics, 2004
[31]. Banerjee S., Lavie A. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005, pp. 65-72. Association for Computational Linguistics, 2005.
[32]. Vedantam R., Zitnick C.L., Parikh D. CIDEr: Consensus-Based Image Description Evaluation. CoRR, 2014, abs/1411.5726. DOI: 10.48550/arXiv.1411.5726.
[33]. Zhang T., Kishore V., Wu F., Weinberger K.Q., Artzi Y. BERTScore: Evaluating Text Generation with BERT. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020.
[34]. Allamanis M., Barr E.T., Devanbu P., Sutton C. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 2018, vol. 51, no. 4, article 81.
[35]. Niu C., Li C., Ng V., Ge J., Huang L., Luo B. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In: 44th IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 1-13. ACM, 2022. DOI: 10.1145/3510003.3510096.
[36]. Yang G., Chen X., Cao J., Xu S., Cui Z., Yu C., Liu K. ComFormer: Code Comment Generation via Transformer and Fusion Method-Based Hybrid Code Representation. In: 8th International Conference on Dependable Systems and Their Applications, DSA 2021, Yinchuan, China, August 5-6, 2021, pp. 30-41. IEEE, 2021. DOI: 10.1109/DSA52907.2021.00013.
[37]. Choi Y., Bak J., Na C., Lee J.-H. Learning Sequential and Structural Information for Source Code Summarization. In: Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021, pp. 2842-2851. Association for Computational Linguistics, 2021. DOI: 10.18653/v1/2021.findings-acl.251.

[38]. Tang Z., Shen X., Li C., Ge J., Huang L., Zhu Z., Luo B. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In: 44th IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 150-162. ACM, 2022. DOI: 10.1145/3510003.3510224.
[39]. Zhang J., Wang X., Zhang H., Sun H., Liu X. Retrieval-Based Neural Source Code Summarization. In: ICSE '20, Seoul, South Korea, pp. 1385-1397. New York, NY, USA, Association for Computing Machinery, 2020. DOI: 10.1145/3377811.3380383.
[40]. Choi Y., Na C., Kim H., Lee J.-H. READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization. IEEE Access, 2023, vol. 11, pp. 51155-51165.
[41]. Villmow J., Ulges A., Schwanecke U. A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks. In: International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021, pp. 1-10. IEEE, 2021.
[42]. Shi Y., Yin Y., Wang Z., Lo D., Zhang T., Xia X., Zhao Y., Xu B. How to Better Utilize Code Graphs in Semantic Code Search? In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, pp. 722-733. Association for Computing Machinery, 2022. DOI: 10.1145/3540250.3549087.
[43]. Wang W., Zhang Y., Sui Y., Wan Y., Zhao Z., Wu J., Yu P., Xu G. Reinforcement-learning-guided Source Code Summarization via Hierarchical Attention. IEEE Transactions on Software Engineering, 2020.
[44]. Son J., Hahn J., Seo H., Han Y.-S. Boosting Code Summarization by Embedding Code Structures. In: Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022, pp. 5966-5977. International Committee on Computational Linguistics, 2022.
[45]. Gao Y., Lyu C. M2TS: multi-scale multi-modal approach based on transformer for source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022, pp. 24-35. ACM, 2022.
[46]. Guo J., Liu J., Wan Y., Li L., Zhou P. Modeling Hierarchical Syntax Structure with Triplet Position for Source Code Summarization. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, ACL 2022, Dublin, Ireland, May 22-27, 2022, pp. 486-500. Association for Computational Linguistics, 2022. DOI: 10.18653/v1/2022.acl-long.37.
[47]. LeClair A., Haque S., Wu L., McMillan C. Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184-195, 2020.
[48]. Lin C., Ouyang Z., Zhuang J., Chen J., Li H., Wu R. Improving Code Summarization with Block-Wise Abstract Syntax Tree Splitting. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension, ICPC 2021, pp. 184-195. IEEE, 2021.
[49]. Shido Y., Kobayashi Y., Yamamoto A., Miyamoto A., Matsumura T. Automatic Source Code Summarization with Extended Tree-LSTM. In: International Joint Conference on Neural Networks, IJCNN 2019, Budapest, Hungary, July 14-19, 2019, pp. 1-8. IEEE, 2019. DOI: 10.1109/IJCNN.2019.8851751.
[50]. Zhang C., Zhou Q., Qiao M., Tang K., Xu L., Liu F. Re_Trans: Combined Retrieval and Transformer Model for Source Code Summarization. Entropy, 2022, vol. 24, no. 10, article 1372.
[51]. Gao S., Gao C., Zeng J., Nie L.Y., Xia X., Lyu M. Code Structure Guided Transformer for Source Code Summarization. ACM Transactions on Software Engineering and Methodology, 2022. DOI: 10.1145/3522674.
[52]. Gong Z., Gao C., Wang Y., Gu W., Peng Y., Xu Z. Source Code Summarization with Structural Relative Position Guided Transformer. In: IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022, pp. 13-24. IEEE, 2022. DOI: 10.1109/SANER53432.2022.00013.
[53]. Wu H., Zhao H., Zhang M. Code Summarization with Structure-induced Transformer. In: Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021, pp. 1078-1090. Association for Computational Linguistics, 2021. DOI: 10.18653/v1/2021.findings-acl.93.
[54]. Zügner D., Kirschstein T., Catasta M., Leskovec J., Günnemann S. Language-Agnostic Representation Learning of Source Code from Structure and Context. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.
[55]. 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.B., Drain D., Sundaresan N., Yin J., Jiang D., Zhou M. GraphCodeBERT: Pre-training Code Representations with Data Flow. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.

[56]. Jiang X., Zheng Z., Lyu C., Li L., Lyu L. TreeBERT: A Tree-Based Pre-Trained Model for Programming Language. In: Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, July 27-30, 2021, vol. 161 of Proceedings of Machine Learning Research, pp. 54-63. AUAI Press, 2021.

[57]. Alon U., Brody S., Levy O., Yahav E. code2seq: Generating Sequences from Structured Representations of Code. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

[58]. Wei B., Li G., Xia X., Fu Z., Jin Z. Code Generation as a Dual Task of Code Summarization. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, December 8-14, 2019, pp. 6559-6569, 2019.

[59]. Xie R., Ye W., Sun J., Zhang S. Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-task Learning Approach. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021, pp. 138-148. IEEE, 2021. DOI: 10.1109/ICPC52881.2021.00022.

[60]. Li J., Li Y., Li G., Hu X., Xia X., Jin Z. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, pp. 155-166. IEEE, 2021. DOI: 10.1109/ASE51524.2021.9678724.

[61]. Parvez M.R., Ahmad W.U., Chakraborty S., Ray B., Chang K.-W. Retrieval Augmented Code Generation and Summarization. In: Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, November 16-20, 2021, pp. 2719-2734. Association for Computational Linguistics, 2021. DOI: 10.18653/v1/2021.findings-emnlp.232.

[62]. Yao Z., Peddamail J.R., Sun H. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. CoRR, 2019, abs/1904.00720. DOI: 10.48550/arXiv.1904.00720.

[63]. Ye T., Wu L., Ma T., Zhang X., Du Y., Liu P., Wang W., Ji S. Tram: A Token-level Retrieval-augmented Mechanism for Source Code Summarization. CoRR, 2023, abs/2305.11074. DOI: 10.48550/arXiv.2305.11074.

[64]. Guo D., Lu S., Duan N., Wang Y., Zhou M., Yin J. UniXcoder: Unified Cross-Modal Pre-Training for Code Representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, ACL 2022, Dublin, Ireland, May 22-27, 2022, pp. 7212-7225. Association for Computational Linguistics, 2022.

[65]. Feng Z., Guo D., Tang D., Duan N., Feng X., Gong M., Shou L., Qin B., Liu T., Jiang D., Zhou M. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, November 16-20, 2020, pp. 1536-1547. Association for Computational Linguistics, 2020. DOI: 10.18653/v1/2020.findings-emnlp.139.

[66]. Kanade A., Maniatis P., Balakrishnan G., Shi K. Learning and Evaluating Contextual Embedding of Source Code. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, Virtual Event, July 13-18, 2020, vol. 119 of Proceedings of Machine Learning Research, pp. 5110-5121. PMLR, 2020.

[67]. Liu F., Li G., Zhao Y., Jin Z. Multi-task Learning Based Pre-trained Language Model for Code Completion. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020, pp. 473-485. IEEE, 2020. DOI: 10.1145/3324884.3416591.

[68]. Wang Y., Wang W., Joty S.R., Hoi S.C.H. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, November 7-11, 2021, pp. 8696-8708. Association for Computational Linguistics, 2021. DOI: 10.18653/v1/2021.emnlp-main.685.

[69]. Wang Y., Le H., Gotmare A.D., Bui N.D.Q., Li J., Hoi S.C.H. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. CoRR, 2023, abs/2305.07922. DOI: 10.48550/arXiv.2305.07922.

[70]. Mastropaolo A., Scalabrino S., Cooper N., Nader-Palacio D., Poshyvanyk D., Oliveto R., Bavota G. Studying the Usage of Text-to-Text Transfer Transformer to Support Code-Related Tasks. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, May 22-30, 2021, pp. 336-347. IEEE, 2021. DOI: 10.1109/ICSE43902.2021.00041.

[71]. Elnaggar A., Ding W., Jones L., Gibbs T., Feher T., Angerer C., Severini S., Matthes F., Rost B. CodeTrans: Towards Cracking the Language of Silicon's Code through Self-Supervised Deep Learning and High Performance Computing. CoRR, 2021, abs/2104.02443. DOI: 10.48550/arXiv.2104.02443.

[72]. Haque S., LeClair A., Wu L., McMillan C. Improved Automatic Summarization of Subroutines via Attention to File Context. In: 17th International Conference on Mining Software Repositories, MSR 2020, Seoul, Republic of Korea, June 29-30, 2020, pp. 300-310. ACM, 2020. DOI: 10.1145/3379597.3387449.

[73]. Zhang J., Panthaplackel S., Nie P., Mooney R.J., Li J.J., Gligoric M. Learning to Generate Code Comments from Class Hierarchies. CoRR, 2021, abs/2103.13426. DOI: 10.48550/arXiv.2103.13426.

[74]. Ahmad W.U., Chakraborty S., Ray B., Chang K.-W. Unified Pre-training for Program Understanding and Generation. 2021. DOI: 10.48550/arXiv.2103.06333.

[75]. Du X., Liu M., Wang K., Wang H., Liu J., Chen Y., Feng J., Sha C., Peng X., Lou Y. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-Level Code Generation. CoRR, 2023, abs/2308.01861. DOI: 10.48550/arXiv.2308.01861.

[76]. Chen Q., Xia X., Hu H., Lo D., Li S. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. ACM Transactions on Software Engineering and Methodology, 2021, vol. 30, no. 2, pp. 25:1-25:29. DOI: 10.1145/3434280.

[77]. Andryushchenko, G., Ivanov, V., Makharev, V., Tukhtina, E. and Valeev, A. Leveraging large language models in code question answering: Baselines and issues. In International Conference on Analysis of Images, Social Networks and Texts, pp. 3-17. Springer Nature Switzerland, 2024.

[78]. GitHub Copilot. Available at: https://github.com/features/copilot. Accessed 10.06.2026.

[79]. Phind. Available at: https://huggingface.co/Phind. Accessed 05.06.2026.

[80]. Rozière B., Gehring J., Gloeckle F., Sootla S., Gat I., Tan X.E., Adi Y., Liu J., Remez T., Rapin J., Kozhevnikov A., Evtimov I., Bitton J., Bhatt M., Canton-Ferrer C., Grattafiori A., Xiong W., Défossez A., Copet J., Azhar F., Touvron H., Martin L., Usunier N., Scialom T., Synnaeve G. Code Llama: Open Foundation Models for Code. CoRR, 2023, abs/2308.12950. DOI: 10.48550/arXiv.2308.12950.

[81]. Ito T., van Deemter K., Suzuki J. Reference-free Evaluation Metrics for Text Generation: A Survey. 2025. DOI: 10.48550/arXiv.2501.12011.

[82]. Chen M., Tworek J., Jun H., Yuan Q., Pondé de Oliveira Pinto H., Kaplan J., Edwards H., Burda Y., Joseph N., Brockman G., Ray A., Puri R., Krueger G., Petrov M., Khlaaf H., Sastry G., Mishkin P., Chan B., Gray S., Ryder N., Pavlov M., Power A., Kaiser L., Bavarian M., Winter C., Tillet P., Such F.P., Cummings D., Plappert M., Chantzis F., Barnes E., Herbert-Voss A., Guss W.H., Nichol A., Paino A., Tezak N., Tang J., Babuschkin I., Balaji S., Jain S., Saunders W., Hesse C., Carr A.N., Leike J., Achiam J., Misra V., Morikawa E., Radford A., Knight M., Brundage M., Murati M., Mayer K., Welinder P., McGrew B., Amodei D., McCandlish S., Sutskever I., Zaremba W. Evaluating Large Language Models Trained on Code. CoRR, 2021, abs/2107.03374. DOI: 10.48550/arXiv.2107.03374.

[83]. Li Y., Choi D.H., Chung J., Kushman N., Schrittwieser J., Leblond R., Eccles T., Keeling J., Gimeno F., Dal Lago A., Hubert T., Choy P., de Masson d'Autume C., Babuschkin I., Chen X., Huang P.-S., Welbl J., Gowal S., Cherepanov A., Molloy J., Mankowitz D.J., Sutherland Robson E., Kohli P., de Freitas N., Kavukcuoglu K., Vinyals O. Competition-Level Code Generation with AlphaCode. CoRR, 2022, abs/2203.07814. DOI: 10.48550/arXiv.2203.07814.

[84]. Li R., Ben Allal L., Zi Y., Muennighoff N., Kocetkov D., Mou C., Marone M., Akiki C., Li J., Chim J., Liu Q., Zheltonozhskii E., Zhuo T.Y., Wang T., Dehaene O., Davaadorj M., Lamy-Poirier J., Monteiro J., Shliazhko O., Gontier N., Meade N., Zebaze A., Yee M.-H., Umapathi L.K., Zhu J., Lipkin B., Oblokulov M., Wang Z., Murthy R.V., Stillerman J.T., Patel S.S., Abulkhanov D., Zocca M., Dey M., Zhang Z., Fahmy N., Bhattacharyya U., Yu W., Singh S., Luccioni S., Villegas P., Kunakov M., Zhdanov F., Romero M., Lee T., Timor N., Ding J., Schlesinger C., Schoelkopf H., Ebert J., Dao T., Mishra M., Gu A., Robinson J., Anderson C.J., Dolan-Gavitt B., Contractor D., Reddy S., Fried D., Bahdanau D., Jernite Y., Muñoz Ferrandis C., Hughes S., Wolf T., Guha A., von Werra L., de Vries H. StarCoder: may the source be with you! Transactions on Machine Learning Research, 2023, vol. 2023.

[85]. Lozhkov A., Li R., Ben Allal L., Cassano F., Lamy-Poirier J., Tazi N., Tang A., Pykhtar D., Liu J., Wei Y., Liu T., Tian M., Kocetkov D., Zucker A., Belkada Y., Wang Z., Liu Q., Abulkhanov D., Paul I., Li Z., Li W.-D., Risdal M., Li J., Zhu J., Zhuo T.Y., Zheltonozhskii E., Dade N.O.O., Yu W., Krauß L., Jain N., Su Y., He X., Dey M., Abati E., Chai Y., Muennighoff N., Tang X., Oblokulov M., Akiki C., Marone M., Mou C., Mishra M., Gu A., Hui B., Dao T., Zebaze A., Dehaene O., Patry N., Xu C., McAuley J.J., Hu H., Scholak T., Paquet S., Robinson J., Anderson C.J., Chapados N., et al. StarCoder 2 and The Stack v2: The Next Generation. CoRR, 2024, abs/2402.19173. DOI: 10.48550/arXiv.2402.19173.

[86]. Chowdhery A., Narang S., Devlin J., Bosma M., Mishra G., Roberts A., Barham P., Chung H.W., Sutton C., Gehrmann S., Schuh P., Shi K., Tsvyashchenko S., Maynez J., Rao A., Barnes P., Tay Y., Shazeer N., Prabhakaran V., Reif E., Du N., Hutchinson B., Pope R., Bradbury J., Austin J., Isard M., Gur-Ari G., Yin P., Duke T., Levskaya A., Ghemawat S., Dev S., Michalewski H., Garcia X., Misra V., Robinson K., Fedus L., Zhou D., Ippolito D., Luan D., Lim H., Zoph B., Spiridonov A., Sepassi R., Dohan D., Agrawal S., Omernick M., Dai A.M., Sankaranarayana Pillai T., Pellat M., Lewkowycz A., Moreira E., Child R., Polozov O., Lee K., Zhou Z., Wang X., Saeta B., Diaz M., Firat O., Catasta M., Wei J., Meier-Hellstern K., Eck D., Dean J., Petrov S., Fiedel N. PaLM: Scaling Language Modeling with Pathways. Journal of Machine Learning Research, 2023, vol. 24, pp. 240:1-240:113.

[87]. Luo Z., Can Xu, Zhao P., Sun Q., Geng X., Hu W., Tao C., Ma J., Lin Q., Jiang D. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net, 2024.

[88]. Guo D., Zhu Q., Yang D., Xie Z., Dong K., Zhang W., Chen G., Bi X., Wu Y., Li Y.K., Luo F., Xiong Y., Liang W. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. CoRR, 2024, abs/2401.14196. DOI: 10.48550/arXiv.2401.14196.

[89]. Zheng T., Zhang G., Shen T., Liu X., Lin B.Y., Fu J., Chen W., Yue X. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In: Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024, pp. 12834-12859. Association for Computational Linguistics, 2024.

[90]. Hui B., Yang J., Cui Z., Yang J., Liu D., Zhang L., Liu T., Zhang J., Yu B., Dang K., Yang A., Men R., Huang F., Ren X., Ren X., Zhou J., Lin J. Qwen2.5-Coder Technical Report. CoRR, 2024, abs/2409.12186. DOI: 10.48550/arXiv.2409.12186.

[91]. Sun W., Miao Y., Li Y., Zhang H., Fang C., Liu Y., Deng G., Liu Y., Chen Z. Source Code Summarization in the Era of Large Language Models. 2024. DOI: 10.48550/arXiv.2407.07959.

[92]. Sun W., Fang C., You Y., Miao Y., Liu Y., Li Y., Deng G., Huang S., Chen Y., Zhang Q., Qian H., Liu Y., Chen Z. Automatic Code Summarization via ChatGPT: How Far Are We? CoRR, 2023, abs/2305.12865. DOI: 10.48550/arXiv.2305.12865.

[93]. Sharma T., Kechagia M., Georgiou S., Tiwari R., Sarro F. A Survey on Machine Learning Techniques for Source Code Analysis. CoRR, 2021, abs/2110.09610. DOI: 10.48550/arXiv.2110.09610.

[94]. Song X., Sun H., Wang X., Yan J. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. IEEE Access, 2019, vol. 7, pp. 111411-111428.

[95]. Zhu Y., Pan M. Automatic Code Summarization: A Systematic Literature Review. CoRR, 2019, abs/1909.04352. DOI: 10.48550/arXiv.1909.04352.

[96]. Yang B., Liping Z., Fengrong Z. A survey on research of code comment. In: Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, ICMSS 2019, Wuhan, China, January 12-14, 2019, pp. 45-51. ACM, 2019.

[97]. Zhang C., Wang J., Zhou Q., Xu T., Tang K., Gui H., Liu F. A Survey of Automatic Source Code Summarization. Symmetry, 2022, vol. 14, no. 3, article 471. DOI: 10.3390/sym14030471.

Информация об авторах / Information about authors

Айдар Ильфатович ВАЛЕЕВ – аспирант АНО ВО «Университет Иннополис» по направлению «Математическое моделирование, численные методы и комплексы программ»; исследователь-разработчик направления GigaCode R&D в ПАО «Сбербанк». Его научные интересы включают суммаризацию и генерацию исходного года с помощью больших языковых моделей (LLM) и агентов на их основе.

Aidar Ilfatovich VALEEV – postgraduate student at Autonomous Non-Profit Organization of Higher Education “Innopolis University”, educational program: “Mathematical modeling, numerical analysis, and software systems”; Research Engineer, GigaCode R&D, Public JSC Sberbank. His research interests include code summarization and generation using large language models (LLM) and agents based on them.

Алексей Олегович КОРШУК – выпускник АНО ВО «Университет Иннополис» по направлению «Информатика и вычислительная техника» (бакалавриат); ведущий инженер-основатель компании Coframe. Сфера научных интересов: большие языковые модели (LLM) для работы с исходным кодом, автономные агенты для программирования и автоматизации веб-интерфейсов, мультиагентное обучение с подкреплением (MARL), A/B-тестирование и персонализация.

Aliaksei Olegovich KORSHUK – graduate of Autonomous Non-Profit Organization of Higher Education "Innopolis University", Bachelor's degree in Computer Science and Engineering; Founding Engineer at Coframe. Research interests include large language models (LLM) for code, autonomous coding agents for frontend and browser automation, multi-agent reinforcement learning (MARL), A/B testing, and personalization.

Владимир Владимирович ИВАНОВ – кандидат физико-математических наук, доцент Центра образовательных программ топ-уровня в сфере искусственного интеллекта АНО ВО «Университет Иннополис». Сфера научных интересов: обработка естественного языка (NLP), большие языковые модели (LLM), генерация кода.

Vladimir Vladimirovich IVANOV – Cand. Sci. (Phys.-Math.), Associate Professor at the Center for Top-Level Educational Programs in Artificial Intelligence, Autonomous Non-Profit Organization of Higher Education “Innopolis University”. Research interests: natural language processing (NLP), large language models (LLM), code generation.