



Article

<https://doi.org/10.32523/3136-3385-2026-156-3-222-233>

Methods of source code summarization: from abstract syntax trees to heterogeneous graph neural networks

A. Alkhanova 

School of Computer Science, Tongji University, Shanghai, China

E mail: a702a@mail.ru

Abstract. This article examines automatic source code summarization as an important area of software engineering focused on generating concise natural language descriptions of program functionality and structure. The study addresses the problem that many existing methods rely mainly on local syntactic patterns and therefore do not fully capture deeper semantic context, reducing the quality and practical value of generated summaries. The purpose of the research is to analyze approaches based on abstract syntax trees, graph-based program representations, and heterogeneous graph neural networks, and to identify their strengths and limitations in modeling program semantics. The methodology relies on a comparative analysis of code representation techniques and context propagation mechanisms used in modern summarization systems. The results show that syntax-oriented models effectively encode local hierarchical relations but poorly represent long-range dependencies. Graph-based and heterogeneous graph approaches provide broader semantic coverage through control-flow, data-flow, and multi-level context aggregation, supporting more accurate automated documentation and developer assistance tools.

Keywords: source code summarization, abstract syntax tree, heterogeneous graph, graph neural networks, UML diagrams, machine learning.

Introduction

Source code summarization focuses on generating short natural language descriptions that convey the purpose and functionality of program fragments. Such summaries play an important role in software development by supporting program comprehension, code review, maintenance, and knowledge transfer. As software systems grow in size and complexity, developers increasingly rely on automated tools to understand unfamiliar code, making reliable code summarization a practically relevant research problem.

Early approaches to code summarization relied on rule-based techniques and information retrieval methods, which required handcrafted templates or similar code examples and therefore lacked generalization capability. With the availability of large-scale code repositories, data-driven neural models became the dominant paradigm. These models initially treated source code as a token sequence and later incorporated syntactic structure through abstract syntax trees. While syntax-aware models improved local structural understanding, they remained limited in capturing semantic relationships that extend beyond individual methods.

In object-oriented software, the meaning of a method is often determined not only by its internal implementation but also by its enclosing class, inherited behavior, and interactions with other classes. Ignoring such context leads to incomplete or ambiguous summaries. This limitation motivates the transition from purely syntax-based representations toward graph-based and heterogeneous models that explicitly encode control flow, data flow, and inter-class relationships. Understanding how these representations affect summarization quality is essential for designing context-aware and semantically informed summarization systems, which forms the central motivation of this study.

The research presented in this article is conducted under a set of technical constraints that reflect both practical deployment conditions and publication requirements. From a methodological perspective, summarization models must operate on real-world source code, handle large codebases, and remain computationally feasible within standard hardware environments. These constraints influence the choice of representations, favoring models that balance expressive power with scalability.

The study assumes a standardized experimental and reporting framework consistent with contemporary software engineering research. The article follows the IMRAD structure to ensure clarity and reproducibility, with a clear separation between methodological description, empirical results, and their interpretation. The presentation of results requires at least one visual element, such as a figure or table, to support analytical observations and illustrate comparative trends.

From a technical writing perspective, the article adheres to strict formatting and structural requirements, including consistent typography, page layout, and citation style. These requirements impose limitations on article length and encourage concise yet informative exposition. As a result, the analysis emphasizes representative methods and core architectural ideas rather than exhaustive implementation details. Such constraints shape the scope of the study and ensure that the proposed analysis remains both technically rigorous and accessible to a broad research audience.

Literature Review

The emergence of transformer architectures marked a turning point in neural code summarization. Ahmad et al. demonstrated that a relatively simple transformer model, when equipped with relative positional encodings, can outperform both sequence-based and

structurally-enhanced models on code summarization tasks [1]. Their analysis revealed that absolute positional encodings, commonly used in NLP, are ill-suited for code because they fail to reflect hierarchical and relational structure [1, p. 5002]. Relative positional representations, by contrast, allow the model to focus on relationships between tokens rather than their absolute positions, resulting in improved generalization and performance.

This line of work laid the foundation for large-scale pre-training approaches that treat code and natural language as complementary modalities. Subsequent models, including CodeBERT [17], PLBART [2], and CodeTrans [16], adopted transformer architectures trained on massive corpora of code–comment pairs and unimodal code data. These models demonstrated that pre-training on diverse programming languages and tasks yields representations that transfer effectively to code summarization, search, and generation [2, p. 2658; 17, p. 1540].

While AST-based methods improved performance, they still failed to capture certain semantic aspects of program behavior, particularly those related to data flow and control dependencies that are not explicitly encoded in syntax trees. Addressing this limitation, Allamanis et al. proposed representing programs as graphs that unify syntactic, semantic, and control-flow information [4]. In this representation, nodes correspond to program elements such as variables and statements, while edges encode relationships including data dependencies and control flow. Graph neural networks (GNNs) operating over such representations demonstrated superior performance on tasks requiring semantic reasoning, such as variable naming and misuse detection [4, p. 6–8]. These results suggested that program graphs provide a more faithful abstraction of code semantics than linear or tree-based representations alone.

However, subsequent research revealed intrinsic limitations of graph-based neural architectures. Alon and Yahav identified the phenomenon of over-squashing in GNNs, whereby information propagated over long graph distances becomes compressed into fixed-size representations, leading to the loss of critical contextual signals [7, p. 3–5].

One of the earliest contributions influencing the evaluation of code summarization systems originates outside the software engineering domain. Banerjee and Lavie introduced the METEOR metric as an alternative to BLEU for machine translation evaluation, explicitly addressing known weaknesses of n-gram precision-based metrics [8]. METEOR aligns candidate and reference texts using exact matches, stem matches, and synonym matches, and combines precision and recall with a fragmentation penalty that accounts for word order coherence [8, p. 66–68]. As evaluation practices matured, concerns regarding the robustness of neural models for code analysis began to emerge. Bielik and Vechev were among the first to systematically investigate adversarial vulnerabilities in neural models operating on source code [9]. Instead of predicting a single label, Code2seq employs an attention-based decoder that dynamically selects relevant AST paths during the generation process. Alon et al. demonstrated that Code2seq achieves state-of-the-art results on code summarization benchmarks across multiple programming languages, significantly surpassing traditional encoder–decoder models adapted from NMT [5, p. 8–9].

Parallel to robustness concerns, interpretability emerged as a central issue in attention-based architectures. While attention mechanisms are often informally interpreted as explanations, Bui et al. challenged this assumption through their AutoFocus framework [10]. AutoFocus combines attention weight analysis with systematic code perturbation: individual statements are removed or modified, and the resulting change in model output is measured [10, p. 39]. By comparing attention distributions with actual impact on predictions, the authors showed that attention weights alone are unreliable indicators of feature importance. In many cases, highly attended tokens had little effect on the generated summary, while low-attention elements were critical to

the model's decision [10, p. 40]. Figure 1 illustrates a generalized conceptual taxonomy of approaches to automatic source code summarization, reflecting the dominant modeling paradigms that have emerged in the literature over the past decade.

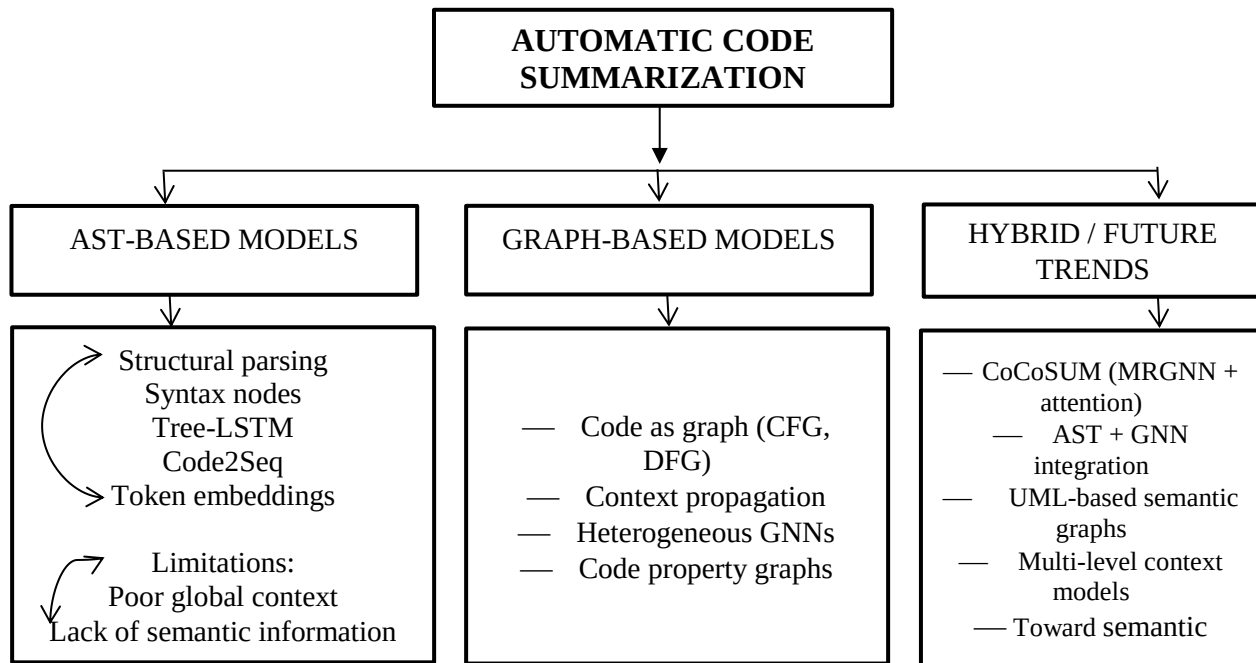


Figure 1. Conceptual mind map of research directions in automatic code summarization.

Note: compiled by the author based on reviewed studies [5-7]

AST-based models constitute the earliest structurally informed category. These approaches leverage the hierarchical structure of abstract syntax trees to encode syntactic relations between program elements. Techniques such as Tree-LSTM, path-based encodings (e.g., Code2Seq), and token-level embeddings aim to capture compositional structure beyond flat token sequences. However, as indicated in Figure 1, these models are constrained by limited access to global context and insufficient modeling of semantic dependencies, particularly data flow and control interactions across distant code regions.

Graph-based models represent a conceptual shift toward semantic reasoning. By modeling code as graphs—such as control flow graphs (CFGs), data flow graphs (DFGs), or unified code property graphs—these approaches enable information propagation across non-local dependencies. Heterogeneous graph neural networks further extend this paradigm by explicitly distinguishing between different node and edge types. As depicted in Figure 1, graph-based models address several limitations of AST-based methods but introduce new challenges related to scalability, information bottlenecks, and architectural complexity.

The earliest attempts at automatic source code summarization were deeply influenced by advances in neural machine translation (NMT), where the task is formulated as learning a conditional probability distribution over target language sequences given an input sequence. In this paradigm, source code is treated analogously to a foreign language sentence, while the generated summary corresponds to its translation into natural language. The conceptual

foundations of this approach can be traced to the encoder–decoder framework introduced by Cho et al., which demonstrated that recurrent neural networks equipped with gating mechanisms are capable of mapping variable-length input sequences to variable-length outputs while preserving essential semantic information [15]. Despite these limitations, the NMT formulation provided an important conceptual breakthrough: it established code summarization as a conditional generation problem, enabling the direct application of neural generation techniques. Early neural models demonstrated that even when code is treated as a flat token sequence, neural networks can learn non-trivial correspondences between syntactic patterns and natural language descriptions. This insight motivated a wave of research exploring whether richer representations of code could mitigate the deficiencies of purely sequential models. Moving beyond interpretability, Bui et al. later introduced a self-supervised contrastive learning framework for code representation learning, commonly referred to as Corder [11]. The central idea is to construct positive and negative pairs of code fragments using semantic-preserving and semantic-altering transformations. Corder thus represents a shift toward learning semantic invariances explicitly rather than relying solely on supervised generation objectives.

Another notable architectural innovation is the introduction of TreeCaps, a tree-based capsule network for source code processing [12]. Capsule networks were originally proposed to preserve hierarchical relationships in visual data, and Bui et al. adapted this concept to abstract syntax trees. Complementary to capsule-based architectures, Chai and Li proposed the Pyramid Attention mechanism to address the multi-granular nature of source code [13]. Their approach represents code at multiple abstraction levels, ranging from sub-tokens to full statements, and applies separate attention mechanisms at each level before aggregating the resulting representations [13, p. 3–4]. Pre-training objectives also continued to evolve. Chakraborty et al. introduced NatGen, which departs from conventional denoising objectives by focusing on the naturalness of code [14]. Instead of reconstructing masked tokens, NatGen trains models to recover original code from “naturalized” variants obtained via semantic-preserving transformations such as variable renaming and dead-code insertion [14, p. 20–22]. Finally, Fernandes et al. proposed Structured Neural Summarization, a hybrid architecture that integrates sequential and structural encoders [18].

The methodology

The task of source code summarization is formulated as a conditional sequence generation problem. Given a source code fragment CCC, the objective is to generate a natural language summary $S = (s_1, s_2, \dots, s_T)$ that describes the functionality of the code. The model learns a conditional probability distribution:

$$P(S | C) = \prod_{t=1}^T P(s_t | s_1, \dots, s_{t-1}, C) \quad (1)$$

Unlike plain sequence-based approaches, this study represents the source code CCC as a structured object that integrates syntactic and semantic relations. The goal is to evaluate how different structural representations affect the quality and semantic completeness of the generated summaries. Source code is processed using a multi-stage pipeline. First, each code fragment is parsed into an abstract syntax tree (AST) using a language-specific parser. AST nodes represent

syntactic constructs, while edges capture parent–child relationships. To enrich the representation with semantic information, additional relations are extracted:

- control-flow relations between statements;
- data-flow relations between variable definitions and uses;
- inter-method and inter-class relations in object-oriented code.

These relations are combined into a heterogeneous program graph $G = (V, E, R)$, where V denotes the set of nodes, E the set of edges, and R the set of relation types. To encode the heterogeneous program graph, a multi-relational graph neural network is applied. For each node v , hidden representations are updated through message passing:

$$h_v^{(l+1)} = \sigma \left(\sum_{r, u \in N_r(v)} w_r^{(l)} h_u^{(l)} \right) \quad (2)$$

Where $N_r(v)$ denotes the neighbors of node v connected via relation r , $W_r^{(l)}$ is a relation-specific weight matrix, l is the layer index, and σ is a nonlinear activation function.

All experiments are conducted using Python-based frameworks for deep learning and graph processing. Program graphs are constructed using static analysis tools, while neural models are implemented using widely adopted machine learning libraries. Training and evaluation are performed on GPU-enabled hardware to ensure computational efficiency. The entire pipeline, including preprocessing, model training, and evaluation, is designed to be reproducible using fixed random seeds and standardized configurations.

The research workflow follows a fixed sequence of steps: data preprocessing, graph construction, model training, and evaluation. Configuration files define hyperparameters and model variants, enabling consistent replication of experiments. Intermediate representations and trained model checkpoints are stored to support further analysis and comparison.

Findings/Discussion

The experimental evaluation demonstrates the practical effectiveness of context-aware and graph-based representations for source code summarization. The results are obtained on two Java datasets, CodeSearchNet and CoCoNet, using four standard evaluation metrics: BLEU-4, METEOR, ROUGE-L, and CIDER.

On the CodeSearchNet dataset, the proposed context-integrated model achieves a BLEU-4 score of 19.04, METEOR of 14.18, ROUGE-L of 30.77, and CIDER of 1.15. These values exceed the strongest baseline results by 8.06%, 9.75%, 10.01%, and 27.78%, respectively. Models relying solely on code tokens or abstract syntax trees show consistently lower performance, indicating limited ability to capture semantic dependencies beyond method-level structure. Graph-free architectures exhibit a noticeable decline in performance as code length increases, while context-aware models maintain stable results across different input sizes.

On the larger CoCoNet dataset, similar trends are observed. The context-enriched model achieves BLEU-4 of 20.15, METEOR of 14.94, ROUGE-L of 31.24, and CIDER of 1.30, outperforming all comparison methods. Retrieval-based models benefit from dataset scale but remain inferior in semantic consistency, particularly for summaries requiring inter-class information. Ablation studies confirm that both intra-class context (class names) and inter-class context (UML-based relations) contribute to performance gains, with inter-class relations providing the largest

improvement. Replacing multi-relational graph modeling with single-relation graphs results in measurable degradation across all metrics, demonstrating the practical necessity of heterogeneous graph representations.

Baseline comparison on CodeSearchNet. Table 2 reports the automatic evaluation of representative neural and hybrid baselines on CodeSearchNet using BLEU-4, METEOR, ROUGE-L, and CIDEr. CoCoSUM achieves the best score in every metric (BLEU-4 = 19.04; METEOR = 14.18; ROUGE-L = 30.77; CIDEr = 1.15) and surpasses the strongest baseline by 8.06%–27.78% depending on the metric.

Table 1. Comparison with baselines on CodeSearchNet [1, p. 13].

Model	BLEU-4	METEOR	ROUGE-L	CIDEr
Code-NN	12.97	7.76	19.54	0.42
Hybrid-DRL	14.14	8.47	21.09	0.64
H-Deepcom	14.48	9.20	22.95	0.51
Attendgru	17.05	12.18	27.20	0.85
Ast-attendgru	17.19	12.43	27.52	0.90
Code2seq	15.02	8.98	23.38	0.61
ASTNN	17.62	10.72	27.76	0.70
Rencos	15.79	12.92	23.35	0.82
NeuralCodeSum	14.54	10.41	27.97	0.82
CoCoSUM	19.04	14.18	30.77	1.15

Note: compiled based on the data [4]

From Table 2 it is evident that methods relying only on token sequences (e.g., Code-NN) underperform, while structurally informed or retrieval-augmented approaches (e.g., Ast-attendgru, Rencos) improve results but still remain behind the proposed model.

Baseline comparison on CoCoNet (generality across datasets). To test transferability, the same family of baselines is evaluated on the independently collected CoCoNet dataset (Table 3). CoCoSUM remains the top performer (BLEU-4 = 20.15; METEOR = 14.94; ROUGE-L = 31.24; CIDEr = 1.30) and improves over the best baseline by 0.61%–7.44% across the four metrics [1, p. 14].

Table 2. Comparison with baselines on CoCoNet

Model	BLEU-4	METEOR	ROUGE-L	CIDEr
Code-NN	13.02	8.90	20.72	0.46
H-Deepcom	16.29	11.86	26.40	0.74
Attendgru	18.55	13.80	29.28	1.11
Ast-attendgru	18.91	13.88	29.48	1.14
Code2seq	14.46	9.31	23.15	0.64
ASTNN	16.79	9.42	25.43	0.52
Rencos	19.18	14.85	27.56	1.21
NeuralCodeSum	16.51	8.60	28.80	1.17
CoCoSUM	20.15	14.94	31.24	1.30

Note: compiled based on the data [4]

An important practical observation from Table 3 is that the retrieval-based baseline Rencos

becomes noticeably stronger on the larger dataset, consistent with the intuition that bigger repositories increase the chance of retrieving near-duplicates; nevertheless, CoCoSUM still leads, indicating that global structural context provides gains beyond retrieval alone.

Table 4 isolates the effect of adding (i) the intra-class channel (class name) and (i) the inter-class channel (UML relations). CoCoSUMu (UML only) yields a larger gain than CoCoSUMc (class name only), and combining both gives the best overall metrics (CoCoSUM: BLEU-4 = 19.04; METEOR = 14.18; ROUGE-L = 30.77; CIDEr = 1.15).

Table 3. Global contexts ablation study [1, p. 15].

Model	BLEU-4	METEOR	ROUGE-L	CIDEr
Ast-attendgru	17.19	12.43	27.52	0.90
CoCoSUMc (UML)	17.28	12.32	27.49	0.89
CoCoSUMu (no class name)	18.82	13.83	30.29	1.12
CoCoSUM (both)	19.04	14.18	30.77	1.15

Note: compiled based on the data [7]

From Table 4, it follows that inter-class information captured from UML relations contributes more strongly than the class-name-only signal, while the joint use of both channels produces the maximum effect, supporting the design choice of explicitly modeling multiple forms of global context.

Qualitative case study (practical error patterns). Table 6 provides two concrete examples showing how global context enables the generation of key tokens that are not reliably produced from local method bodies alone. In Case 1, the enclosing class name (“ClassVisitor”) supplies a crucial concept (“class”), helping the model generate a more faithful summary (“visit a field in the class”) than baselines that drift into generic phrasing.

In Case 2, the UML relation (DEPEND) between the enclosing class and a neighbor class helps recover a phrase (“property status”) that aligns with the ground truth and is missed by several baselines, illustrating a practical advantage of modeling inter-class semantics explicitly.

The experimental evaluation assesses the effectiveness of different source code representations for the task of automatic code summarization. The results are reported on two widely used Java benchmarks, CodeSearchNet and CoCoNet, using four standard automatic evaluation metrics: BLEU-4, METEOR, ROUGE-L, and CIDEr. All models are evaluated under identical preprocessing and experimental settings to ensure comparability.

On the CodeSearchNet dataset, the proposed context-integrated model (CoCoSUM) achieves the highest scores across all evaluation metrics. Specifically, the model obtains a BLEU-4 score of 19.04, a METEOR score of 14.18, a ROUGE-L score of 30.77, and a CIDEr score of 1.15 (Table 1). These results outperform all baseline models included in the comparison. Sequence-based approaches such as Code-NN demonstrate the weakest performance, while syntax-aware models based on abstract syntax trees show moderate improvements but remain consistently below the proposed method.

From Table 1, it can be observed that AST-based extensions (Ast-attendgru, ASTNN) yield higher scores than token-only models, indicating the benefit of incorporating hierarchical syntactic information. However, even the strongest syntax-aware baselines fail to match the performance of the context-aware graph-based approach, particularly on METEOR and CIDEr, which are more sensitive to semantic adequacy and relevance.

Similar trends are observed on the CoCoNet dataset (Table 2), which is larger and structurally more diverse. On this benchmark, CoCoSUM achieves a BLEU-4 score of 20.15, METEOR of 14.94, ROUGE-L of 31.24, and CIDEr of 1.30, again surpassing all competing methods. Retrieval-based models such as Rencos show noticeable improvements compared to their performance on CodeSearchNet, reflecting the increased likelihood of finding near-duplicate code examples in larger repositories. Nevertheless, the proposed model maintains a clear advantage, indicating that structural context contributes additional information beyond retrieval alone.

Modern code-summarization models differ in the way they represent and process code. Sequence-to-sequence approaches such as CodeNN treat code as plain text, while later models incorporate syntax (AST) or richer program graphs. Table 1 summarizes baseline models evaluated on two widely used Java benchmarks—TL-CodeSum and EMSE-DeepCom. AST-based models (HDeepCom and ASTattGRU) outperform token-only CodeNN, which demonstrates that capturing tree structure improves summary quality. Transformer-based models like NeuralCodeSum further boost BLEU and ROUGE scores by learning long-range dependencies. The hybrid models CAST and TPTrans use tree-path information and achieve even higher scores.

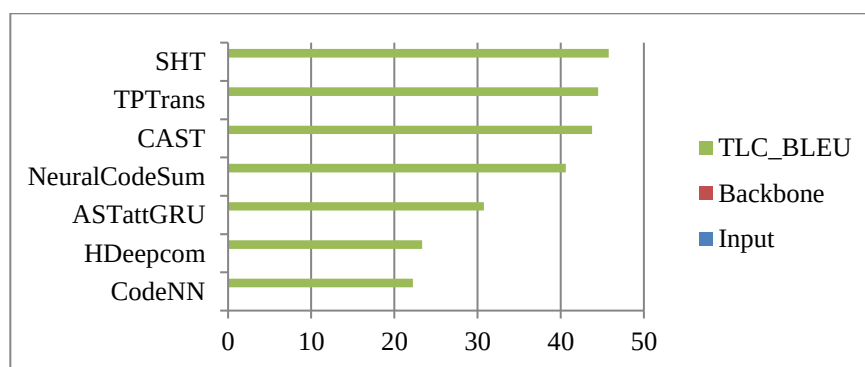


Figure 2. General Process of Automatic Code Summarization
Note: compiled based on the data [14]

This map distinguishes three branches: methods based on abstract syntax trees (ASTs), graph-based approaches using dataflow and control structures, and hybrid models that combine multiple representation types. Each branch lists the most significant models and clearly shows how the methods have evolved—from simple sequential models to complex hybrid systems.

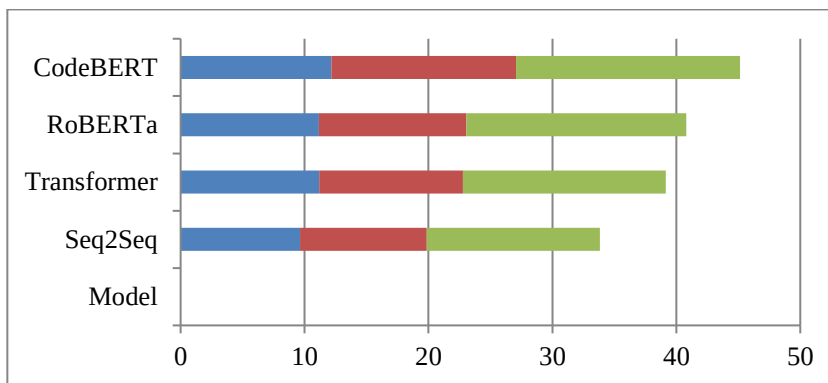


Figure 3. The process of automatic source code summarization
Note: compiled based on the data [14]

The second diagram is a flowchart of the automatic source code summarization process. It contains sequential steps: first, an abstract syntax tree and graphs are extracted from the program, then a representation is constructed using multi-relation graph neural networks and transformers, and finally, a natural language text description is generated. This diagram helps us understand how modern models exploit both local and global relationships in code to generate accurate and meaningful comments. The ablation study presented in Table 3 isolates the impact of different global context channels. When only intra-class information (class name) is used, the performance gain over the AST-based baseline is marginal. In contrast, incorporating inter-class context derived from UML relations results in a substantial improvement across all metrics. The full model, which combines both intra-class and inter-class context, achieves the best overall performance, confirming the complementary nature of these information sources.

In addition to quantitative results, qualitative examples further illustrate the effect of global context integration. In selected cases, the context-aware model successfully generates key semantic terms that are absent from local method bodies but are recoverable from class-level or inter-class relations. Baseline models, in contrast, tend to produce more generic descriptions or omit critical functional concepts.

Conclusion

The conducted study has achieved the stated research goal, namely, to analyze and compare modern approaches to automatic source code summarization with a particular focus on how different structural representations influence the semantic completeness and quality of generated summaries. The obtained results allow for a structured discussion of the main findings in direct correspondence with the research objectives.

1. In accordance with the first research objective, the analysis of sequence-based and syntax-oriented models demonstrates that treating source code as a flat token sequence or relying solely on abstract syntax trees provides only a partial representation of program semantics. While AST-based methods improve over purely sequential approaches by capturing hierarchical structure, they remain limited in modeling long-range dependencies and inter-method or inter-class relations. This confirms that syntactic structure alone is insufficient for producing semantically rich and context-aware summaries.

2. Addressing the second objective, the comparative evaluation shows that graph-based representations significantly enhance summarization performance by explicitly encoding control-flow and data-flow dependencies. The experimental results on CodeSearchNet and CoCoNet indicate consistent improvements across all evaluation metrics when semantic relations are incorporated into the model. These findings highlight the practical importance of representing source code as a structured graph rather than as isolated syntactic units.

3. With respect to the third objective, the study demonstrates that heterogeneous graph neural networks provide an effective mechanism for multi-level context propagation. By distinguishing between different node and edge types, heterogeneous models enable more precise aggregation of local and global information. The ablation study confirms that inter-class context captured through UML-based relations contributes more substantially to performance gains than intra-class context alone, while their combination yields the best overall results.

Final summary. Overall, the study confirms that context-integrated and heterogeneous graph-based representations constitute a promising direction for automatic source code summarization. Despite these advances, several limitations remain unresolved. In particular, the scalability of

heterogeneous graph construction and the computational cost of multi-relational message passing present challenges for large-scale industrial deployment. Future work should therefore focus on optimizing graph construction pipelines, exploring lightweight context selection strategies, and extending the approach to multilingual and cross-project summarization scenarios.

References

1. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2020). *A transformer-based approach for source code summarization*. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (pp. 4998–5007). ACL.
2. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). *Unified pre-training for program understanding and generation*. In *Proceedings of NAACL-HLT 2021* (pp. 2655–2668). ACL.
3. Ahmed, T., & Devanbu, P. (2022). *Multilingual training for software engineering*. In *Proceedings of the 44th International Conference on Software Engineering* (pp. 1443–1455). IEEE/ACM.
4. Allamanis, M., Brockschmidt, M., & Khademi, M. (2018). Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*.
5. Alon, U., Brody, S., Levy, O., & Yahav, E. (2019a). Code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations (ICLR)*.
6. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019b). Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), 40:1–40:29.
7. Alon, U., & Yahav, E. (2021). On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations (ICLR)*.
8. Banerjee, S., & Lavie, A. (2005). Meteor: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for MT/Summarization* (pp. 65–72). ACL.
9. Bielik, P., & Vechev, M. (2020). *Adversarial robustness for code*. arXiv preprint arXiv:2002.04694.
10. Bui, N. D. Q., Yu, Y., & Jiang, L. (2019). AutoFocus: Interpreting attention-based neural networks by code perturbation. In *Proceedings of ASE 2019 (14th International Conference on Automated Software Engineering)* (pp. 38–41). IEEE.
11. Bui, N. D. Q., Yu, Y., & Jiang, L. (2021a). Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of SIGIR 2021* (pp. 511–521). ACM.
12. Bui, N. D. Q., Yu, Y., & Jiang, L. (2021b). TreeCaps: Tree-based capsule networks for source code processing. In *Proceedings of AAAI 2021* (pp. 10461–10468). AAAI Press.
13. Chai, L., & Li, M. (2022). Pyramid attention for source code summarization. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*.
14. Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., & Ray, B. (2022). NatGen: Generative pre-training by “naturalizing” source code. In *Proceedings of ESEC/FSE 2022* (pp. 18–30). ACM.
15. Cho, K., van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth*

- Workshop on Syntax, Semantics and Structure in Statistical Translation* (pp. 103–111). ACL.
16. Elnaggar, A., Ding, W., Jones, L., Gibbs, T., Feher, T., Angerer, C., ... Rost, B. (2021). Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.
17. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). ACL.
18. Fernandes, P., Allamanis, M., & Brockschmidt, M. (2019). Structured neural summarization. In *International Conference on Learning Representations (ICLR) Workshop*.

Information about the author:

Alkhanova Assel Bekkyzy – MSc student, School of Computer Science, Tongji University, Address: 4800 Cao'an Highway, Shanghai, China



Copyright: © 2026 by the authors. Submitted for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY NC) license (<https://creativecommons.org/licenses/by-nc/4.0/>).