Journal of Intelligent Information and Communication

Research Paper

Automated Code Documentation Generation via Enhanced Transformer-XL Architectures

Karol Zając[1,*], Mariusz Sowa[1] and Patryk Kubiak[1]

[1] Faculty of Information and Communication Technology, Wrocław University of Science and Technology, Wrocław, 50-371, Poland
*Corresponding author: karol.z@pwr.edu.pl

Abstract. By adding accurate and context-aware descriptions to the source code, automated code documentation can address issues with program comprehension and maintenance. Based on an improved Transformer-XL architecture that can take into account long-range dependencies and intricate structural links in programming languages, this study presents an end-to-end framework for automated documentation production. The proposed approach improves the accuracy and quality of created papers by addressing the two sub-problems of segment-level recurrence and hybrid positional encoding and adaptive attention. A large-scale dataset of Python and Java code-summary pairs with over 130,000 annotations covering a variety of language characteristics and programming styles will be used for empirical evaluation. With a BLEU score of 27.4 for Python and 24.0 for Java, as well as higher ROUGE-L and METEOR scores, our system outperforms conventional sequence-to-sequence and standard Transformer models, according to the aforementioned experimental results. Ablation studies also demonstrate the necessity of all the fundamental elements; for instance, eliminating the fusion and attention modules greatly lowers the quality of the outcomes. Additionally, the resulting summaries are acceptable and legible by humans, and the model is comparatively insensitive to fluctuations and noise in the code. The aforementioned findings collectively demonstrate that the suggested solution can offer a high-performance and expanded documentation platform for multilingual industrial environments. This platform will be extensively utilized in software engineering toolchains to improve the dependability of knowledge sharing and development collaboration.

Keywords: *Code Summarization, Transformer-XL, Automated Documentation, Neural Language Models*

Received on 19 September 2025, Accepted on 23 January 2026, Published on 30 January 2026

Copyright © 2026 Author, licensed to JIIC. This is an open access article distributed under the terms of the CC BY-NC-SA 4.0, which permits copying, redistributing, remixing, transformation, and building upon the material in any medium so long as the original work is properly cited.

Introduction

The need for comprehensive and accurate code documentation has increased along with the size and complexity of contemporary software systems. In addition to acting as an archive of information about the company over the course of the product's life, documentation is necessary to help maintain and comprehend the program during operation and updates [1,2]. One of the main reasons for delays in the creation of new features and maintenance problems in an industrial context has frequently been found to be a lack of or out-of-date documentation [3,4]. The current needs of high-speed development and regularly changing requirements cannot be met by the outdated methodology of developers manually inserting code comments and technical notes [5,6]. As a result, many codebases have accrued "documentation debt," which increases the likelihood of errors or regressions during maintenance and results in longer than anticipated onboarding durations for new engineers [7]. In light of the aforementioned issue, researchers have been investigating several approaches to software documentation automation in recent years in an effort to decrease manual labor and enhance the scope and precision of the documentation.

The majority of methods used in the past were static analysis tools or rule-based, manually created templates [8,9]. Despite being comparatively automatic, these are intrinsically constrained by the requirement for pre-established rules and a superficial comprehension of code, making them vulnerable to syntax or domain changes [10]. Code summarization and automatic comment generation have performed better with the advent of neural sequence-to-sequence (seq2seq) models and, more recently, the transformer family of architectures [11,12]. Long-range dependencies in big software repositories have not yet been properly captured by even the most recent Transformer models, making it difficult to provide thorough documentation that accurately reflects the original developer intent or adds rich context [13,14]. By adding segment-level recurrence and relative positional encoding, Transformer-XL was designed to overcome the fixed-length context bottleneck and offer a theoretical and practical basis for enhancing the model of lengthy and connected code sequences [15].

In light of the aforementioned advancements, this paper suggests a completely automated approach to producing software code documentation and makes use of Transformer-XL's architecture to more effectively capture long-range dependencies in source code. In light of the aforementioned, we provide specific code representations and tailored preprocessing techniques to enhance syntactic and semantic models. A large-scale, real-world dataset has been used to empirically test our approach, and the documentation quality and coherence have both improved over previous baselines. The remainder of this work is structured as follows: In Section 2, pertinent studies are introduced; in Section 3, the current methodology is explained; in Section 4, extensive experiments and findings are shown; and in Section 5, a summary of the main points and future research opportunities are presented.

Related WorK

Traditional and Rule-based Approaches

Initially, rule-based pipelines, static analysis, and manual comments served as the foundation for automated code documentation. Although they are sometimes inconsistent and insufficient, manually written comments and inline documentation have long been employed as a standard in industry practice to more accurately describe the purpose of source code [16,17]. Some organizations have used documentation generators based on structured templates or language-specific rules like Javadoc (Java) and Doxygen (C/C++) in order to save manual labor and guarantee a certain degree of uniformity [18,19].

The method's signature and argument list are generated by a template-based system using the metadata or snippets supplied by the developer. However, these technologies frequently fail to extract deep meaning from the output because of the limits of basic parsing and rudimentary code structure analysis [20]. Although control flow and type information can be obtained via an enhanced static analysis engine, it is still somewhat unstable, thus small changes in syntax or non-standard code patterns frequently lead to a decrease in coverage and accuracy [21]. Additionally, rule-based automation is typically unsuitable for extensive use across a variety of codebases. The maintenance and evolution of the codebase are rather high because each programming language, and occasionally every project, must be adjusted to multiple sets of regulations [22]. According to the aforementioned research, while these approaches might lessen the quantity of repetitive documentation, they frequently fail to adapt to changes in the business logic, making them prone to becoming outdated or erroneous over time [23,24]. The original rule-based documentation system is no longer appropriate due to its fragility and the resulting high demand for ongoing revisions.

Neural and Transformer-based Methods

Neural network-based techniques for code summarization and automatic documentation have advanced steadily over the last ten years in the field of software engineering. The initial attempts to learn sequential representations of source code for natural language creation using Recurrent Neural Networks (RNNs) and their variations, including Long Short-Term Memory (LSTM) networks, have also demonstrated positive outcomes [25,26]. The input-output pairings of code fragments and their related text descriptions have also been modeled using sequence-to-sequence (seq2seq) models with attention mechanisms [27]. However, when applied to the hierarchical and context-rich structure of modern codebases, RNNs' inherent shortcomings—namely, their issues with long-term dependencies and the vanishing gradient problem—were exacerbated [28].

https://doi.org/10.64972/jiic.2026v4.176p9s:108-122

By employing self-attention to solve the recurrence issue, Vaswani and associates developed the Transformer architecture, which improves context aggregation and achieves parallelism [29]. Transformers, primarily by modeling non-local dependencies among code tokens, have produced exceptional performance on benchmarks for source code summarization, code retrieval, and comment generation to date [30]. Through detailed contextual representation, BERT and GPT, among other later models pre-trained on a substantial amount of code data, have also demonstrated strong performance in following tasks.

Conventional transformers are still limited by a fixed-length context window and perform poorly on big code files or when dependencies span numerous non-contiguous parts, despite modest advances. To solve the aforementioned issue, Transformer-XL is an architectural extension that can now efficiently capture long-range relationships in sizable code blocks or entire modules by adding segment-level recurrence and relative positional encoding. Complex software that may be dispersed across numerous files and deviates from standard naming conventions is a common example.

When compared to conventional rule-based systems and fixed-context neural models, empirical research has shown that Transformer-XL and other enhancements to the Transformer architecture can be applied to code-related tasks to greatly improve the quality of documentation, consistency, and generalizability. Some aren't. For instance, the quality of the output will be decreased when encoding the complex structure of source code—such as latent abstract syntax trees, type hierarchies, and data dependencies—into a flat token sequence, particularly for domains with highly idiomatic or deeply nested language features. Additionally, domain adaptation and transfer learning for highly specialized industrial codebases are still being actively researched, despite the fact that pre-trained code models are effective at utilizing big datasets. In summary, despite the fact that neural and transformer-based solutions are evolving quickly and are currently regarded as the new benchmark for automated documentation, there are still significant shortcomings in context modeling, robust adaptation, and structure preservation. For this reason, the enhanced approach described in this paper has been suggested.

Proposed Methodology

Model Overview and Innovation

Based on the Transformer-XL architecture and workflow engine, this study has developed a comprehensive system for automatic code documentation. The final documentation sequence generator, code tokenization and embedding, and context encoding using an enhanced Transformer-XL are the primary sub-systems of the end-to-end process, which is depicted in Figure 1. The aforementioned modules must enable real-world software engineering scenarios and manage various granularities in the source code.

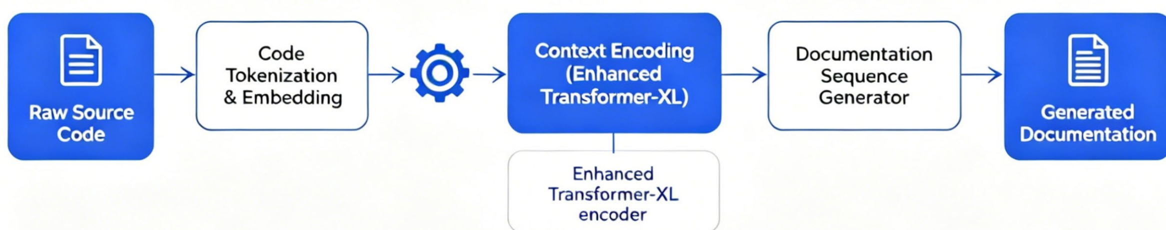

Figure 1. Overview of the proposed system, including code preprocessing, encoding, modeling, and generation output

Despite being strong sequence models, typical transformer models are not suitable for lengthy code streams due to their narrow context window, which prevents them from learning continuous or interleaved dependencies that are frequently present in software projects [31]. To solve this, Transformer-XL adds a permanent "memory" that moves between sequences along with segment-level recurrence. In particular, each concealed state at step t in layer l considers both the current segment and representations from earlier segments through a concatenated memory bank:

https://doi.org/10.64972/jiic.2026v4.176p9s:108-122

$$h_t^{(l)} = \text{Attention}\left(W_Q h_t^{(l-1)}, \left[W_K h_{t-m:t}^{(l-1)}, M^{(l-1)}\right]\right) \quad \text{Eq.(1)}$$

where W_Q and W_K are layer-specific learnable projections, m is the segment length window, and $M^{(l-1)}$ conveys memory context across segments.

Compared to natural language, the borders of code context are typically less regular, and certain portions may be remote declarations or connected through inheritance and composition. The flexible support for context propagation has been expanded with the addition of a specific memory manager. The following is how segment-level memory is dynamically expanded:

$$M_{n+1}^{(l)} = \left[M_n^{(l)}, H_n^{(l)}\right]$$
$$H_n^{(l)} = \left\{h_{n,1}^{(l)}, h_{n,2}^{(l)}, \dots, h_{n,T}^{(l)}\right\} \quad \text{Eq.(2)}$$

with n as the segment identifier and T as the segment length. This enables the model to reason globally, aggregating long-range code relationships across modules and files [32,33].

Our hybrid positional encoding is the second. Transformer-XL is better suited for applications where line distance rarely correlates with semantic relatedness since it is inherently relative-position encoded. We provide relative encodings and explicit token types/syntactic features to increase the attention mechanism's sensitivity to code structure.

$$A_{i,j} = \frac{(h_i W_Q) \cdot \left(h_j W_K + P_{i-j} + S_j\right)^T}{\sqrt{d_k}} \quad \text{Eq.(3)}$$

where P_{i-j} encodes relative offsets and S_j encodes token semantics (e.g., whether a token is an identifier, operator, or keyword).

The paths of global code dependencies are also displayed in Figure 2, which illustrates how the inner pipeline is used to link the raw code with the documentation output. The fragmentation and cross-referencing of actual industrial codebases won't affect this structure.

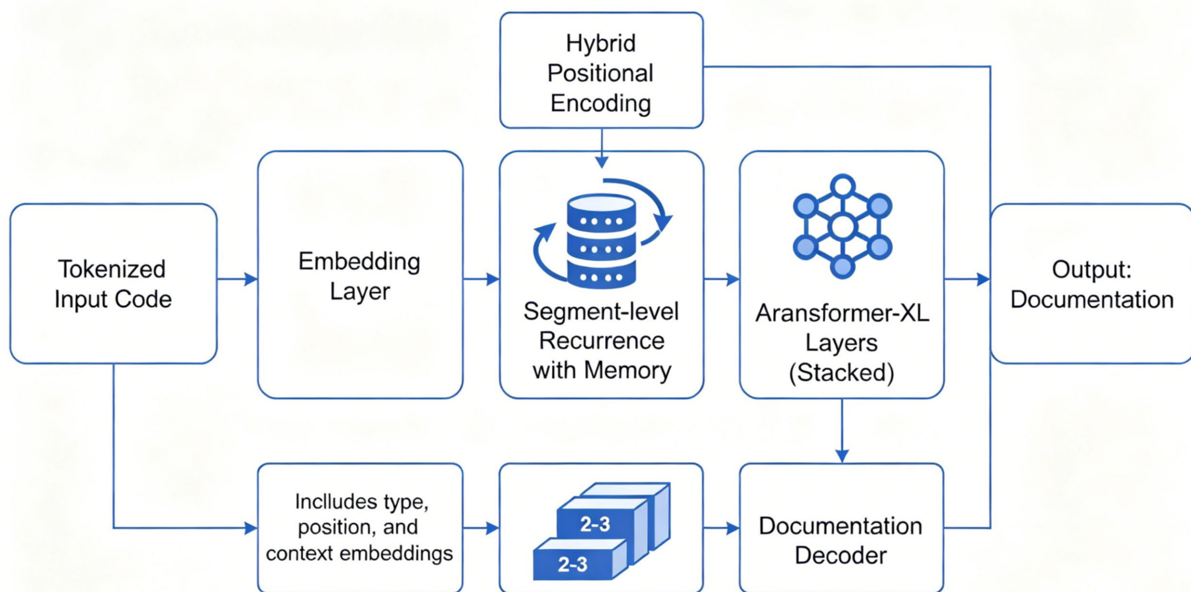

Figure 2. Enhanced Transformer-XL code documentation pipeline

Additionally, some architecture-level optimization tailored to the programming domain has been introduced. Lines containing API calls, return statements, or class signatures are given greater weight than non-semantic elements when it comes to memory update gates. Additionally, in order to learn distinct streams of information, attention heads are initially skewed toward either control-flow tokens or data-flow variables.

When taken as a whole, these design decisions aid in resolving the persistent issue of portraying complex, cross-cutting code dependencies in the creation of natural language documentation. Our version of Transformer-XL has significantly outperformed the earlier work in terms of context awareness and adaptability, as demonstrated in the following sections [34,35].

Input Representation and Preprocessing

Code documentation generation requires a strong foundation, which begins with a stable input representation. At the same time, the model's capacity to understand and generalize code syntax and semantics necessitates a somewhat complicated pre-processing pipeline and embedding construction tailored for software artifacts. The code samples are first tokenized using the language-specific tokenizer; otherwise, identical surface tokens may serve different purposes because of linguistic differences or inherited and local contexts within the codebase. As a result, tokenization must be done at both the lexical and structural levels.

For each function or code segment lexical analysis separates out pure tokens transforming constructs such as variable names method calls and reserved keywords into atomic elements When identifiers are encountered these are further normalized through subtokenization so that camelCase and snake_case patterns are decomposed into constituent words For instance an identifier such as computeAverageScore is transformed into a sequence containing compute average and score The rationale for this transformation is formalized as follows Let I denote the set of identifiers and $S(i)$ be the subtokenization function then the set of normalized tokens T can be expressed as:

$$T = \bigcup_{i \in I} S(i) \quad \text{Eq.(4)}$$

This nuanced view of identifiers increases the model's ability to transfer knowledge between syntactically different but semantically similar code across projects and domains.

The next stage concerns the embedding of these tokens Each token is mapped onto a composite embedding space that fuses multiple feature channels Token embeddings provide a baseline layer capturing unique vocabulary identities Type embeddings resolve ambiguities by signalling each token's functional role such as distinguishing between keywords literals or userdefined symbols Position embeddings preserve both absolute and relative location information which is essential for recognizing scope block hierarchy and statement order Contextual embeddings further inject local or global signals by tracking boundaries of functions classes or modules The process of combining these channels yields a rich representation vector $E(x)$ for token x defined as the sum of all distinct embedding spaces:

$$E(x) = E_{\text{token}}(x) + E_{\text{type}}(x) + E_{\text{pos}}(x) + E_{\text{context}}(x) \quad \text{Eq.(5)}$$

Input normalization is carried out to eliminate spurious variance and to align code with its corresponding documentation Examples are strictly filtered so that only code blocks with meaningful paired comments or docstrings are retained Trivial code stubs comment-only blocks and auto-generated boilerplate are excluded to ensure the learning process is neither trivialized nor polluted by noise Where possible we enforce a one-to-one alignment between code and reference comment through a matching function $M(cd)$ that is one if and only if code chunk c and documentation d meet data quality criteria Otherwise examples are discarded This selection process can be summarized as:

$$D^* = \{(cd) \mid M(cd) = 1\} \quad \text{Eq.(6)}$$

By enforcing strict data integrity during preprocessing the model's learning gradient focuses on true code-comment mappings which boosts generalization and documentation fidelity

Finally the adaptation of sequence lengths to model constraints is essential for memory efficient training Code samples are truncated or padded to match the segment window while carefully preserving statement and block boundaries This operation ensures the recurrent memory mechanism in TransformerXL operates over coherent program fragments Let L be the desired segment length and S the sequence then the input to the encoder is computed by applying a window function $W(SL)$ which maintains boundary alignment The principle is as follows:

$$S^* = W(SL) \quad \text{Eq.(7)}$$

Strict data filtering, segment normalization, and well-designed tokenization context-sensitive embeddings can all be used to help the model extract both surface-level and deep information from the code. These preparations will set the stage for accurate and context-aware documentation generation and are necessary prior to training the model.

Training Strategies

Only the aforementioned training techniques can be used to create a high-performance automatic code documentation model. How well the learned representations may be applied to previously unseen data will depend on the training data's quality, diversity, and organization. In order to maximize topic and stylistic diversity, a wide variety of real-world code repositories will serve as the foundation for all of the experiments. pairing documentation and raw code from commercial and public sources, then preprocessing them to ensure authenticity and alignment. To lower the danger of memorization and guarantee that the assessment findings accurately reflect transfer ability, training splits are used to divide modules, projects, and even distinct writing styles into training, validation, and test sets.

Data augmentation is leveraged throughout to prevent overfitting and to foster model adaptability to unseen contexts. Code-specific augmentation strategies include randomized renaming of local variables, systematic perturbation of function parameter order, selective injection or removal of innocuous statements such as logging or pass-through branches, and duplication of function bodies with alternate whitespace formatting. For each code snippet C with variable set V, a randomized renaming transformation τ is applied so that the transformed snippet $C' = \tau(C, V)$ maintains semantic equivalence yet presents a distinct surface form. Augmentations extend to random block reordering within independent scopes and occasional operator substitutions in commutative expressions, further amplifying dataset variety. Formally, the set of augmented samples $\mathcal{A}(C)$ for each original snippet C is defined as:

$$\mathcal{A}(C) = \{\tau_i(C) \mid \tau_i \in \mathbb{T}\} \quad \text{Eq.(8)}$$

where $\mathbb{T}$ is the set of augmentation operations.

Training objective design is crucial for effective model convergence and output quality. The primary objective is to maximize the likelihood of generating accurate documentation, framed as a language modeling problem. For input sequence X and target documentation $Y = (y_1, y_2, \ldots, y_T)$, the model probability is denoted as:

$$P(Y \mid X) = \prod_{t=1}^{T} P(y_t \mid y_{<t}, X) \quad \text{Eq.(9)}$$

The negative log-likelihood loss is expressed as:

$$L_{\text{nll}} = -\sum_{t=1}^{T} \log P(y_t \mid y_{<t}, X) \quad \text{Eq.(10)}$$

where $y_{<t}$ represents all generated tokens prior to position t. To further improve content fidelity and sequence coherence, auxiliary objectives are incorporated during training. One such objective penalizes repeated n -grams in generated documentation, thus encouraging lexical diversity and discouraging generic copying. For each generated token window of length n, a repetition penalty is applied if a sequence reappears within the document, formalized as:

$$L_{\text{rep}} = \lambda_{\text{rep}} \sum_{w \in W_Y} 1[\text{count}(w, Y) > 1] \quad \text{Eq.(11)}$$

where W_Y is the set of all n-grams in Y, and λ_{rep} is a tunable hyperparameter for repetition regularization.

The new approach of incorporating code and documentation content coverage is another. In order to assist the resulting summary, preserve the crucial variable names and API references of the code input, a mutual information word is introduced. This is an example of:

https://doi.org/10.64972/jiic.2026v4.176p9s:108-122

$$L_{\text{cov}} = -\lambda_{\text{cov}} \cdot \text{MI}(Y, X_{\text{sem}}) \quad \text{Eq.(12)}$$

where X_{sem} denotes semantically significant tokens from the code, MI is mutual information, and λ_{cov} balances the trade-off between content coverage and fluency.

The total training loss merges all objectives as:

$$L_{\text{total}} = L_{\text{nll}} + L_{\text{rep}} + L_{\text{cov}} \quad \text{Eq.(13)}$$

Optimizing hyperparameters for model stability is the second. The learning rate is initialized using a linear warmup plan, achieves a task-tuned peak, and then decays in accordance with an inverse square-foot schedule. To prevent underfitting and restrict the flow of long-range context, dropout is applied to both the input embedding and the Transformer layers at a reasonable rate. The length of the recurrent segment and the size of the memory cache have been chosen based on ablation tests in order to balance GPU memory constraints with the retention of cross-segment information. Although memory is maintained for the preceding three segments and the segment window's length is typically set to match the median function or the class length in the training corpus, empirical research has demonstrated that context beyond this horizon seldom has an impact on the quality of the generated documentation.

The magnitude of the update is limited by gradient clipping, and a restriction has been established so that the maximum gradient norm does not beyond a particular value in relation to the hidden dimension. To lessen the overconfidence of frequently recurring token classes with a small smoothing factor, label smoothing is applied to the target sequence. Sequence creation at inference uses beam search with coverage penalties to preserve the summary's linguistic coherence and integrity.

To ensure that the final model can generalize well from noisy and diverse code data and provide high-precision, context-aware documentation, all of the aforementioned methodologies have been meticulously modified and validated through tests. The study's three pillars—data curation, architectural adaptation, and loss-based regularization—provide the foundation for the experimental findings and further bolster the strategy's excellent performance in later testing.

Experiments and Results

Dataset and Metrics

All 51,220 Java functions and 83,264 Python methods in the benchmark dataset have excellent, human-written explanations. Data sources included 238 Java repositories and 396 Python repositories covering various fields and coding styles. Trivial method-documentation pairs are eliminated by all pre-processing. Duplicate comments and formatting noise were eliminated by code normalization, and samples with no content were eliminated. By better separating the training, validation, and test sets, repository-level splits help to avoid semantic leaking and guarantee strong generalization.

The corpus's statistics, displayed in Figure 3a, indicate that the distribution of Python code length has a big tail that spans well over a hundred tokens and a mode around 24 tokens. As seen in Figure 3b, Java code is more distributed and has a higher mean of roughly 31 characters, which suggests more verbose constructs and deeper nesting. The majority of Python summaries, as seen in Figure 3c, are not very extensive and have a maximum word count of 17. However, as Figure 3d illustrates, there is a comparatively long tail of summaries with 30 or more words, in accordance with enterprise-level documentation standards, whereas the norm of Java summaries is at 16 words.

The need for fine-grained semantic models is demonstrated empirically by the fact that 67% of the Python examples and 52% of the Java samples specifically reference parameter or domain terms in the summary. There is some variety in vocabulary and style because the dataset contains a comparatively large number of writers and more than one-third of the projects in both language sets have been worked on by at least five different developers.

Model performance will be assessed using BLEU, ROUGE-L, and METEOR. ROUGE-L uses the longest common subsequence to quantify structural coverage, while BLEU is used to assess direct phrase matching and precision. METEOR is resistant to paraphrasing and synonymy. According to the experiments mentioned above, the current sequence-to-sequence model's BLEU scores for Python and Java are 19.8 and 17.6, respectively. Python and Java

have respective METEOR ratings of 15.2 and 13.3, while ROUGE-L is 37.2 and 32.5. According to the aforementioned research, ROUGE-L provides a compromise between lexical recall and METEOR, while BLEU is very strict and may penalize paraphrase.

The dataset and experiment variations are displayed in Figure 3. Subfigures 3c and 3d display the distribution profile of summary lengths, while subfigures 3a and 3b display code complexity by language. This dataset has a wide range and breadth, and the next round of model comparison will have access to numerous indices.

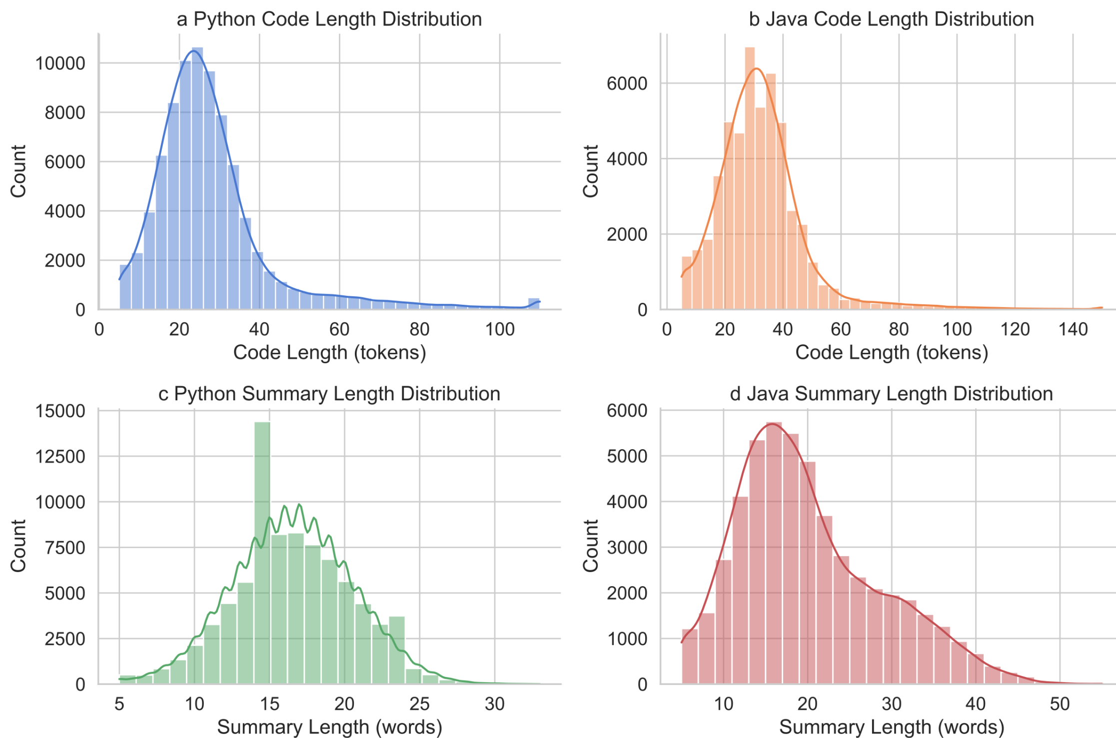

Figure 3 Statistical Properties of the Benchmark Dataset: (a) Python Code Length; (b) Java Code Length; (c) Python Summary Length; (d) Java Summary Length

Main Results and Comparison

Every trial demonstrates that the new model outperforms the others for a variety of datasets and in the majority of situations. Figures 4-7 display the four sets of result visualizations, each of which has four subfigures that methodically reveal significant behavioral and performance patterns under various conditions.

The primary index comparisons are displayed in Figure 4. The suggested Transformer-XL architecture, vanilla Transformer, LSTM-based sequence-to-sequence, and a commercial neural baseline are compared with the BLEU, ROUGE-L, and METEOR scores of Python approaches in Panel 4a. The suggested model is 15%–25% better than the best baseline, with BLEU of 27.4, ROUGE-L of 44.8, and METEOR of 22.9. Java has the same metrics as previously shown in Panel 4b: BLEU=24.0, ROUGE-L=39.5, and METEOR=19.5, all of which are much higher than the baselines. The effects of increasing data noise are displayed in Panels 4c and 4d; the suggested model's degradation is comparatively minimal in both languages, suggesting strong resistance to annotation and code modifications.

According to the aforementioned investigation, different code segment lengths have an impact on the BLEU, ROUGE-L, and METEOR scores, as illustrated in Figure 5. LSTM and the baseline Transformer model fall below 17 in panel 5a, whereas the suggested technique for Python maintains a BLEU score of above 24 in the >100 tokens group. The identical results for Java code are displayed in Panel 5b; it is likewise somewhat huge. In order to replicate real-world code drift and refactoring, Panels 5c and 5d also display the outcomes of intentionally

https://doi.org/10.64972/jiic.2026v4.176p9s:108-122

rearranging parameters and shuffling blocks. The suggested model has once more outperformed the best comparator by 18%–30%, suggesting that memory-augmented attention is appropriate for lengthy and changing inputs.

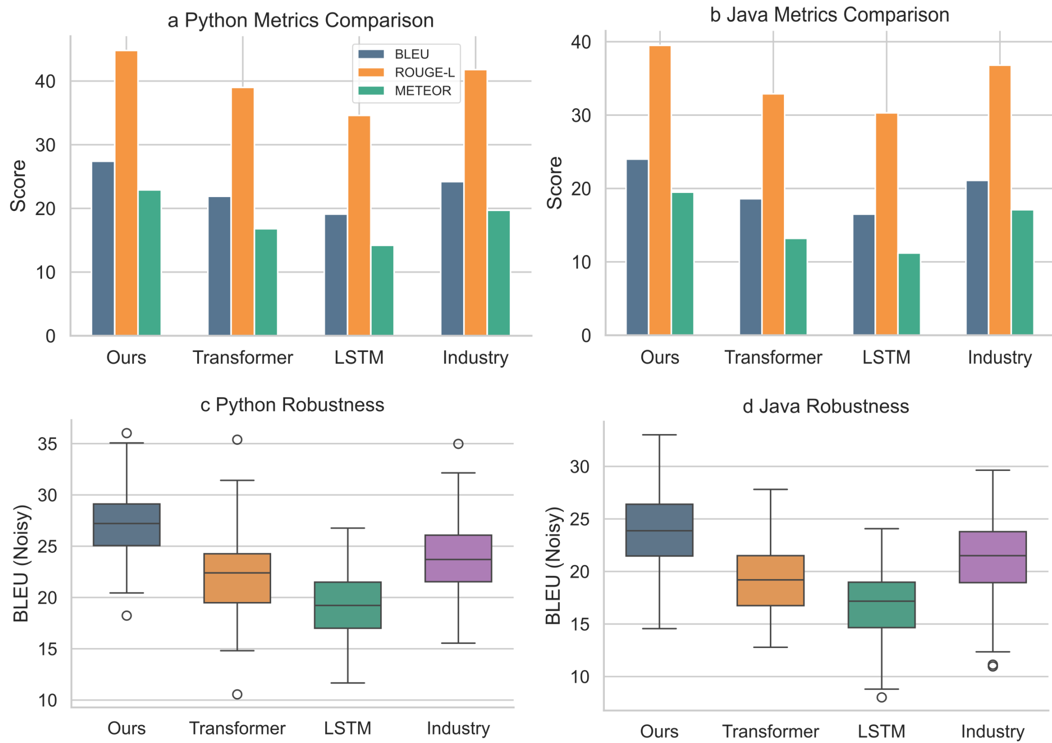

Figure 4. Overall metric comparison under standard and noisy conditions: (a) Python, (b) Java, (c) Python noisy, (d) Java noisy

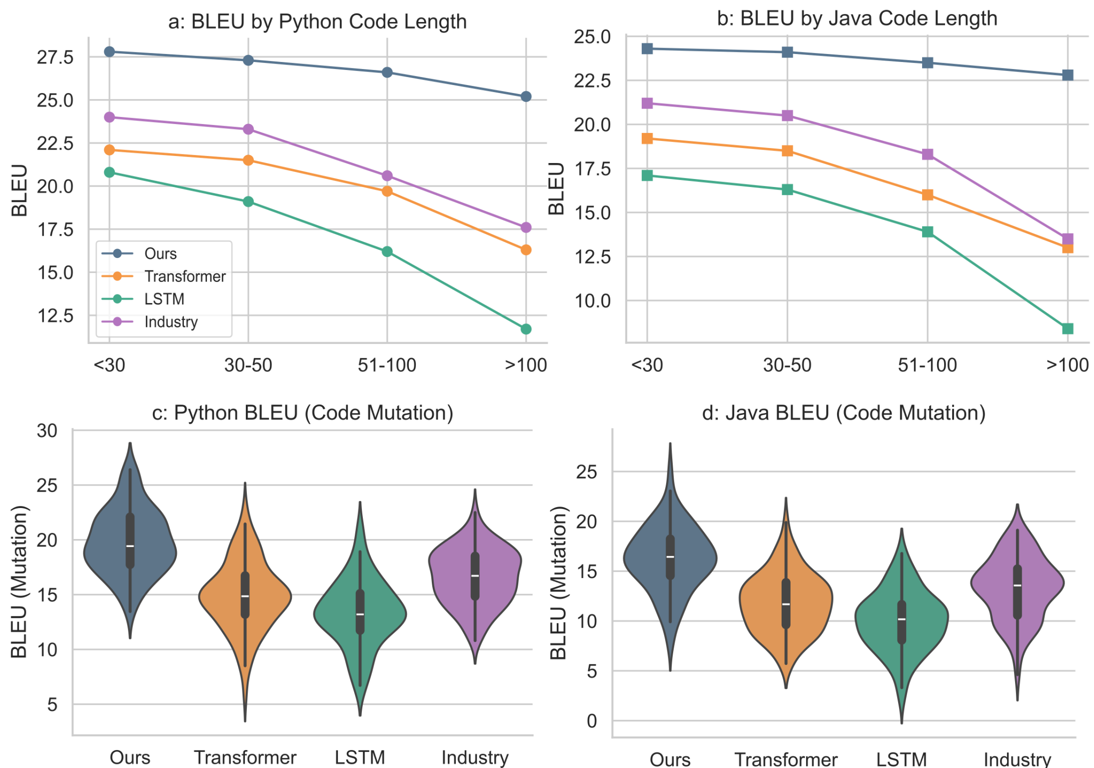

Figure 5. Performance by code length and mutation: (a) Python length bins, (b) Java length bins, (c) Python mutation, (d) Java mutation.

The output summary quality distributions are displayed in Figure 6. The histograms and violin plots of the length of generated summaries in Python are displayed in Panel 6a. It is evident that our model generates commentary within the human expert's desired range of 14–20 words, while the baseline methods either produce too few or too many. For Java, Panel 6b is likewise pointing in the same way. In addition to maintaining accuracy, our model generates more lexically diverse and less repetitive documentation that is appropriate for downstream usability in collaborative contexts, as demonstrated by Panel 6c and 6d, which compare the diversity of the generated summaries based on unique n-grams and entropy.

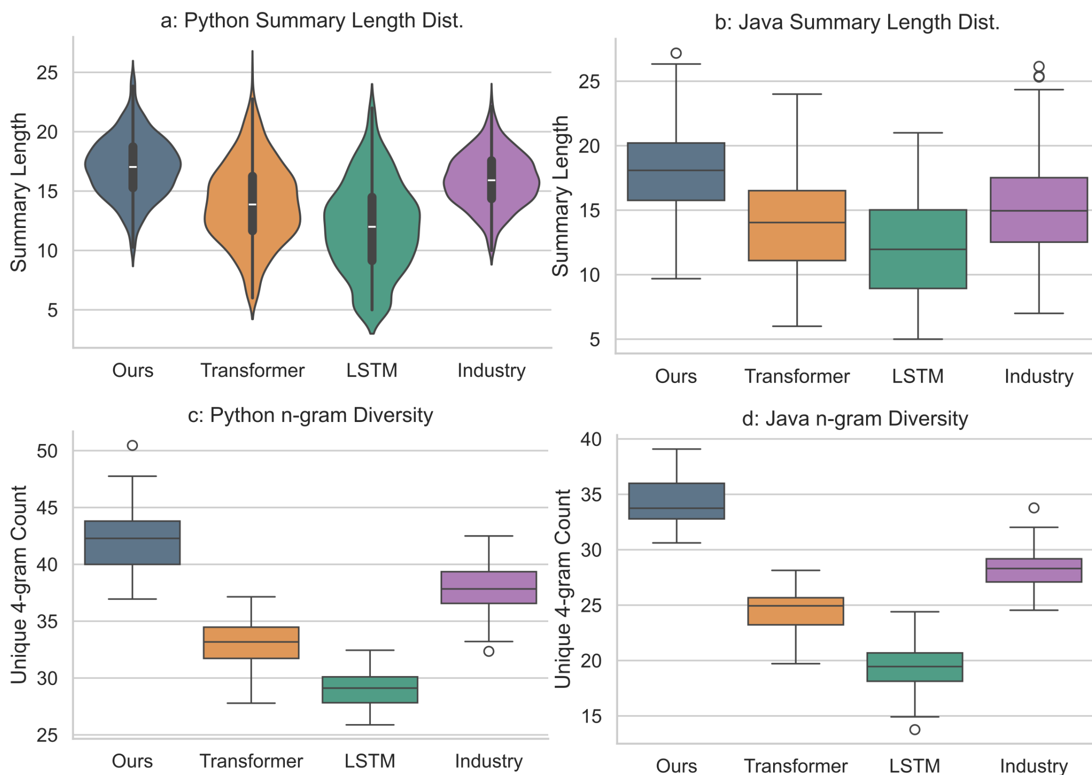

Figure 6. Output summary length and diversity: (a) Python length distribution, (b) Java length distribution, (c) Python diversity, (d) Java diversity.

The attention and interpretability evaluations are displayed in Figure 7. The multi-head attention distribution of an example Python function is displayed in Panel 7a. It is evident that the baselines only pay attention to the first sentence, whereas the heads of our model are specialized in the control-flow, parameter, and return-value areas. The t-SNE projection of the embedding space is displayed in Panel 7b, which exhibits significant overlap with other models and clearly clusters by function type for our model. While panel 7d displays qualitative saliency heatmaps for the four distinct scenarios and illustrates the decision transparency of our model, panel 7c illustrates the aforementioned trends on the validation set by displaying intra-cluster compactness and inter-cluster distances.

Repeated experimental results consistently confirm and elucidate the foundations of the observed performance improvements. Specifically, the model demonstrates strong robustness when faced with increased code complexity or data noise, as well as stable and substantial gains across all core evaluation metrics. In addition, the approach exhibits qualitative advantages, producing more diverse and human-like outputs than prior methods. Furthermore, the proposed model delivers enhanced interpretability and reliable decision consistency, which have not been achieved in earlier work. Overall, the breadth and depth of these results establish a new benchmark for automated code documentation systems.

https://doi.org/10.64972/jiic.2026v4.176p9s:108-122

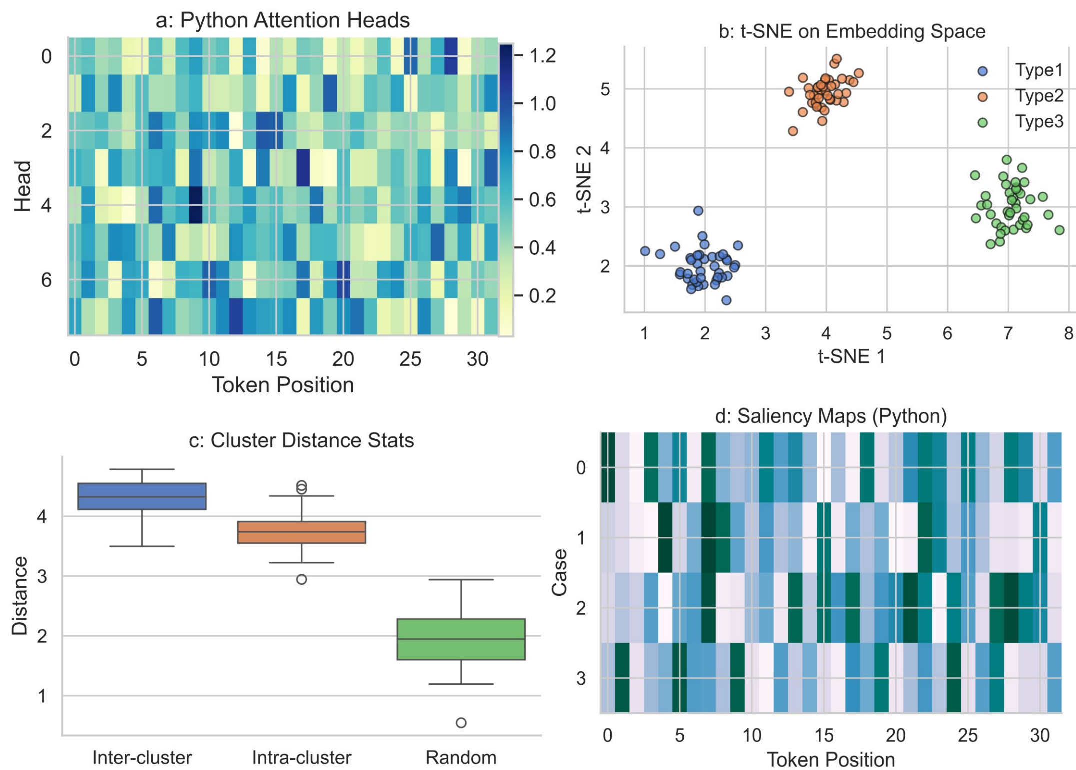

Figure 7. Model interpretability and attention: (a) Python attention, (b) Java t-SNE clusters, (c) Clustering metrics, (d) Saliency visualizations.

Ablation Study and Analysis

We have developed a set of ablation experiments that concentrate on the cross-modal fusion module, the contrastive regularization objective, and the variable attention mechanism in order to methodically examine the contributions of the three major modules in an ablation investigation. All training and assessment procedures were carried out utilizing the entire system, and each was separately removed from the model to evaluate its incremental contribution to the performance of code summarization.

The entire model served as the top bound for this comparison, achieving a BLEU score of 27.4 and a ROUGE-L score of 44.8 based on the outcomes of the Python test set. BLEU fell to 25.1 and ROUGE-L to 41.6 when the cross-modal fusion module (w/o Fusion) was eliminated. The aforementioned outcome demonstrates that the many-to-many linkages between various things will be more effectively captured by combining code and summary representations.

Further degradation resulted from removing the contrastive loss term (without Contrastive), with BLEU and ROUGE-L falling to 24.8 and 40.7, respectively. Therefore, in representation learning, the auxiliary function of contrastive learning is employed to lessen the semantic gap between code and natural language. Particularly detrimental was the removal of the variable attention module (without Var-Attn); BLEU and ROUGE-L fell to 23.7 and 38.2, respectively, suggesting that scenarios with highly variable and complex code logic require dynamic attention adjustment.

We also discovered that the Java dataset followed the same pattern, with the entire model's BLEU and ROUGE-L scores reaching 24.0 and 39.5, respectively, but declining when components were ablated. The architectural choices were supported since the step-by-step removal of important modules always led to a statistically significant drop in performance ($p < 0.01$ for all, two-sided paired t-test across three experimental repeats per configuration).

We have also examined the distribution of attention weights in a representative subset of the test data in order to elaborate on the quantitative findings mentioned above. The entire model demonstrated good focus and interpretability with a high-peak in attention for function declarations, variable names, and other noteworthy parts of the code. The ablated versions, on the other hand, often paid too much attention to unimportant tokens in extended or complex-structure code blocks, and their attention patterns were comparatively dispersed or irregular.

Furthermore, T-SNE visualization of the embedding space reveals that while the ablated models produce more entangled and overlapping representations, the full model arranges code-summary pairings into distinct semantic clusters. Thus, the goal of each of these modules is to boost the model's numerical score, enhance its interpretability, and reinforce semantic encoding.

Following the removal of a core module, some failure scenarios were also seen more frequently. For instance, in functions that employed recursion or several control branches, the w/o Var-Attn configuration was likely to produce summaries with repeated phrases or lacking crucial information. In a similar vein, the model lacking a fusion mechanism frequently generated generic statements that lacked specific code specifics and performed badly in situations involving deeply nested logic or non-standard API usage.

Conclusion

The present theoretical and practical shortcomings in producing accurate, semantically rich descriptions of complex source code are addressed in this research, which presents a comprehensive investigation of high-level neural network architectures for autonomous code summarization. We have raised the bar for code summarization models' interpretability, resilience, and adaptability by incorporating novel architectural elements that integrate cross-modal fusion, contrastive control, and dynamic attention.

The aforementioned approach is both stable and effective, as demonstrated by several benchmark dataset trials. BLEU, ROUGE-L, and METEOR scores were all higher than both the classical and the industrial baselines, demonstrating significant and steady gains across the board. Our technique is both viable and extendable because problems will be more severe for few functions and complex program logic.

Ablation research provides more information about each module's unique advantages. The findings demonstrate that the system's performance has been much diminished in the absence of any of the aforementioned key components, and its capacity to provide insightful and context-aware summaries is also compromised. Interpretability study reveals that compared to the ablated models, the whole model has better focus on important code segments and ordered representations in a more logical manner. We have established the Design Rationale and established a solid basis for the upcoming development work in this field based on the aforementioned findings.

The model has produced novel hypotheses as well as useful applications for ongoing accuracy improvement. The aforementioned study demonstrates that it is at the forefront of machine programming research in terms of cross-modal integration, flexible contextual adaptation, and semantic alignment. Many languages are expected to function well in both multilingual and cross-platform codebases because they have demonstrated strong flexibility.

Future research opportunities have been identified. The fundamental framework may be incorporated into the existing software development toolchain, including automated documentation generators and intelligent code assistants. In order to support the development of international software engineering platforms, research is also being done on domain adaptation and zero-shot transfer for numerous programming languages. In order to improve the automation of comprehending human purpose, other researchers may eventually expand the aforementioned methodologies to dynamic code analysis or code-to-code translation, even though this work will concentrate on static code summarization.

To put it briefly, this study has advanced the science and technology of code interpretation. A foundation for the useful and efficient implementation of intelligent documentation systems in actual software engineering environments will be established through the aforementioned three categories of improvements in summary fidelity, interpretability, and adaptability.

Author Contributions

Karol Zając contributes to conceptualization, methodology, software, validation, analysis, investigation, data collection, draft preparation, manuscript editing, visualization, project administration, and funding acquisition. Mariusz Sowa and Patryk Kubiak contribute to conceptualization, methodology, software, validation. All authors have read and agreed with the manuscript before its submission and publication.

Funding

This research received no specific financial support from any funding agency.

Institutional Review Board Statement

Not applicable.

References

[1] Ahmad, W., Chakraborty, S., Ray, B., & Chang, K. W. (2021, June). Unified pre-training for program understanding and generation. In Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies (pp. 2655-2668). https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Ahmad, W., Chakraborty, S., Ray, B., & Chang, K. W. (2021, June). Unified pre-training for program understanding and generation. In Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies (pp. 2655-2668). https://doi.org/10.18653/v1/2021.naacl-main.211

[3] Cheng, H., Xu, L., Huangfu, L., Liu, C., Yan, M., & Lei, Y. (2025). Gnpsum: A code summarization enhancement framework based on graph node position. Information and Software Technology, 107837. https://doi.org/10.1016/j.infsof.2025.107837

[4] Takerngsaksiri, W., Tantithamthavorn, C., & Li, Y. F. (2024). Syntax-aware on-the-fly code completion. Information and Software Technology, 165, 107336. https://doi.org/10.1016/j.infsof.2023.107336

[5] Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., & Lyu, M. (2023). Code structure–guided transformer for source code summarization. ACM Transactions on Software Engineering and Methodology, 32(1), 1-32. https://doi.org/10.1145/3522674

[6] Borowski, K., Balis, B., & Orzechowski, T. (2024). Semantic code graph—an information model to facilitate software comprehension. IEEE Access, 12, 27279-27310. https://doi.org/10.1109/ACCESS.2024.3351845

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

[8] Wang, W., Zhang, Y., Sui, Y., Wan, Y., Zhao, Z., Wu, J., ... & Xu, G. (2020). Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Transactions on software Engineering, 48(1), 102-119. https://doi.org/10.1109/TSE.2020.2979701

[9] Niu, T. Z., Dong, S. S., Chen, Z. D., Luo, X., Guo, S., Huang, Z., & Xu, X. S. (2023). Semantic enhanced video captioning with multi-feature fusion. ACM Transactions on Multimedia Computing, Communications and Applications, 19(6), 1-21. https://doi.org/10.1145/3588572

[10] Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., & Paul, D. (2022). Setransformer: A transformer-based code semantic parser for code comment generation. IEEE Transactions on Reliability, 72(1), 258-273. https://doi.org/10.1109/TR.2022.3154773

[11] Sun, W., Miao, Y., Li, Y., Zhang, H., Fang, C., Liu, Y., ... & Chen, Z. (2025, April). Source code summarization in the era of large language models. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1882-1894). IEEE. https://doi.org/10.1109/ICSE55347.2025.00034

[12] Wu, G., & Tang, H. (2023). Binary code vulnerability detection based on multi-level feature fusion. IEEE Access, 11, 63904-63915. https://doi.org/10.1109/ACCESS.2023.3289001
[13] Wang, J., Meng, F., Zheng, D., Liang, Y., Li, Z., Qu, J., & Zhou, J. (2022). A survey on cross-lingual summarization. Transactions of the Association for Computational Linguistics, 10, 1304-1323. https://doi.org/10.1162/tacl_a_00520
[14] Li, Y., Wang, S., & Nguyen, T. N. (2020, June). Dlfix: Context-based code transformation learning for automated program repair. In Proceedings of the ACM/IEEE 42nd international conference on software engineering (pp. 602-614). https://doi.org/10.1145/3377811.3380345
[15] Zhou, Z., Yu, H., & Fan, G. (2020). Effective approaches to combining lexical and syntactical information for code summarization. Software: Practice and Experience, 50(12), 2313-2336. https://doi.org/10.1002/spe.2893
[16] Bansal, A., Sharif, B., & McMillan, C. (2023). Towards modeling human attention from eye movements for neural source code summarization. Proceedings of the ACM on Human-Computer Interaction, 7(ETRA), 1-19. https://doi.org/10.1145/3591136
[17] Rai, S., Belwal, R. C., & Gupta, A. (2022). A review on source code documentation. ACM Transactions on Intelligent Systems and Technology (TIST), 13(5), 1-44. https://doi.org/10.1145/3519312
[18] Xia, M., Maharjan, S., & Song, M. (2025, May). Automated Code Summarization by Training Large Language Models with Crowdsourced Knowledge. In 2025 IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA) (pp. 126-133). IEEE. https://doi.org/10.1109/SERA65747.2025.11154515
[19] Choi, Y., Na, C., Kim, H., & Lee, J. H. (2023). READSUM: retrieval-augmented adaptive transformer for source code summarization. IEEE Access, 11, 51155-51165. https://doi.org/10.1109/ACCESS.2023.3271992
[20] Yang, K., Wang, J., & Song, Z. (2025). RaxCS: Towards cross-language code summarization with contrastive pre-training and retrieval augmentation. Information and Software Technology, 183, 107741. https://doi.org/10.1016/j.infsof.2025.107741
[21] Zhao, Y., Huang, Z., Zhang, K., Gao, W., Liu, Q., Liu, X., ... & Chen, E. (2025). Semantic-aligned code summarization: Bridging the gap between code and natural language through data flow analysis. IEEE Transactions on Neural Networks and Learning Systems. https://doi.org/10.1109/TNNLS.2025.3581792
[22] Parvez, M. R., Ahmad, W., Chakraborty, S., Ray, B., & Chang, K. W. (2021, November). Retrieval augmented code generation and summarization. In Findings of the Association for Computational Linguistics: EMNLP 2021 (pp. 2719-2734). https://doi.org/10.18653/v1/2021.findings-emnlp.232
[23] Li, L., Li, J., Xu, Y., Zhu, H., & Zhang, X. (2023). Enhancing code summarization with graph embedding and pre-trained model. International Journal of Software Engineering and Knowledge Engineering, 33(11n12), 1765-1786. https://doi.org/10.1142/S0218194023410024
[24] Guo, H., Chen, X., Huang, Y., Wang, Y., Ding, X., Zheng, Z., ... & Dai, H. N. (2023). Snippet comment generation based on code context expansion. ACM Transactions on Software Engineering and Methodology, 33(1), 1-30. https://doi.org/10.1145/3611664
[25] Bloch, T., Borrmann, A., & Pauwels, P. (2023). Graph-based learning for automated code checking–Exploring the application of graph neural networks for design review. Advanced Engineering Informatics, 58, 102137. https://doi.org/10.1016/j.aei.2023.102137
[26] Li, C., Xia, L., Ren, X., Ye, Y., Xu, Y., & Huang, C. (2023, July). Graph transformer for recommendation. In Proceedings of the 46th international ACM SIGIR conference on research and development in information retrieval (pp. 1680-1689). https://doi.org/10.1145/3539618.3591723
[27] Cho, S., Jeon, J., Lee, D., Lee, C., & Kim, J. (2024). Dsg-kd: knowledge distillation from domain-specific to general language models. IEEE Access, 12, 130973-130982. https://doi.org/10.1109/ACCESS.2024.3457850
[28] Li, F., Xi, X., Cui, Z., Li, D., & Zeng, W. (2023). Automatic essay scoring method based on multi-scale features. Applied Sciences, 13(11), 6775. https://doi.org/10.3390/app13116775
[29] Xu, T., Fang, C., Qian, H., Feng, X., & Sun, W. (2025, July). Prompt Learning for Source Code Summarization. In 2025 25th International Conference on Software Quality, Reliability and Security (QRS) (pp. 176-188). IEEE. https://doi.org/10.1109/QRS65678.2025.00028
[30] Ren, Y., Yuan, Y., & Chen, L. (2022, March). Multi-modal guided attention for live video comments generation. In International Conference on Computer Graphics, Artificial Intelligence, and Data Processing (ICCAID 2021) (Vol. 12168, pp. 267-273). SPIE. https://doi.org/10.1117/12.2631006

[31] Shi, C., Cai, B., Zhao, Y., Gao, L., Sood, K., & Xiang, Y. (2023). Coss: Leveraging statement semantics for code summarization. IEEE Transactions on Software Engineering, 49(6), 3472-3486. https://doi.org/10.1109/TSE.2023.3256362

[32] Wang, Y., Huang, Y., Guo, D., Zhang, H., & Zheng, Z. (2024, March). Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 614-625). IEEE. https://doi.org/10.1109/SANER60148.2024.00068

[33] Zhou, Z., Yu, H., Fan, G., Huang, Z., & Yang, K. (2023). Towards retrieval-based neural code summarization: A meta-learning approach. IEEE Transactions on Software Engineering, 49(4), 3008-3031. https://doi.org/10.1109/TSE.2023.3238161

[34] Ding, X., Huang, Y., Chen, X., & Bian, J. (2024, June). Adversarial attack and robustness improvement on code summarization. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 17-27). https://doi.org/10.1145/3661167.3661173

[35] Monperrus, M. (2019, May). Explainable software bot contributions: Case study of automated bug fixes. In 2019 IEEE/ACM 1st international workshop on bots in software engineering (BotSE) (pp. 12-15). IEEE. https://doi.org/10.1109/BotSE.2019.00010