Journal of Theoretical and Applied Information Technology
15th November 2024. Vol.102. No. 21

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

# EVALUATING RNNS AND TRANSFORMERS FOR CODE-RELATED TASKS INCLUDING BUG DETECTION, CODE COMPLETION, AND SUMMARIZATION

**RAGHUPATHY DURGA PRASAD[1], Dr. MUKTEVI SRIVENKATESH[2]**

[1]Research Scholar, GITAM University, Department of Computer Science, Visakhapatnam, India

[2]Associate Professor, GITAM University, Department Computer Science,Visakhapatnam, India

E-mail: [1]deardp@gmail.com, [2]smuktevi@gitam.edu

**ABSTRACT**

Software engineering tasks such as bug detection, code completion, and code summarization are critical for improving the efficiency and reliability of software development. With the increasing complexity of modern codebases, there is a need for robust deep learning models capable of understanding and predicting patterns in source code. This study addresses the challenge of selecting the most suitable model for these tasks by conducting a comprehensive evaluation of two prominent architectures: Recurrent Neural Networks (RNNs) and Transformers. RNNs are effective at capturing short-term dependencies but struggle with long sequences, while Transformers excel at modelling long-range dependencies through self-attention mechanisms. This research evaluates the performance of these models across three key tasks—bug detection, code completion, and summarization—using multiple datasets. The results demonstrate that Transformers consistently outperform RNNs in terms of accuracy and Bilingual Evaluation Understudy (BLEU) scores, particularly for tasks involving long code sequences, while RNNs are more computationally efficient in memory-constrained environments. The study contributes to the field by providing practical insights for developers and researchers on how to leverage these models based on task requirements and available computational resources. These findings highlight the potential of Transformers to enhance the accuracy of software engineering tools, while also presenting a trade-off in terms of resource consumption, making this study valuable for future model selection and optimization efforts.

**Keywords**: *Source Code Analysis, Bug Detection, Code Completion, Code Summarization, Recurrent Neural Networks (RNNs), Transformer Model*

## 1. INTRODUCTION

The advent of deep learning has significantly transformed many fields, with natural language processing (NLP) seeing notable advancements. Among the key architectures, Recurrent Neural Networks (RNNs) and Transformers have become powerful tools for handling sequential data. Initially designed for tasks like machine translation and text summarization, these models are now widely applied in software engineering, especially in analyzing and interpreting source code [1, 2].

Source code, like natural language, has a structured and sequential nature, making it a suitable candidate for deep learning models typically used in NLP. However, the sequential dependencies, along with the syntactic and semantic complexities of code, pose unique challenges that demand specialized models. RNNs, which can maintain memory of previous inputs, were some of the first neural architectures used for code analysis. However, their limitations in handling long-term dependencies reduce their effectiveness in certain tasks—a challenge that Transformers, with their self-attention mechanisms, are better suited to address [1, 3].

Transformers excel at capturing long-range dependencies and have quickly become the model of choice for many code-related tasks. Their architecture supports parallel processing and provides better contextual understanding, making them particularly effective in code summarization, bug detection, and code completion [2, 3]. However, their superior performance comes with the trade-off of higher computational requirements, which is a critical factor in real-world applications [1, 4].

The use of deep learning in software engineering has brought significant improvements in automating various aspects of software development, particularly in detecting and correcting programming errors. Traditional methods, which often relied on heuristics, lacked the adaptability that modern deep learning models offer. The introduction of RNNs and, more recently, Transformers, has opened new avenues to address these challenges by leveraging large datasets of source code [2, 5].

RNNs were among the first neural models applied to tasks like code completion, bug detection, and automatic code summarization. They are proficient at capturing short-term dependencies within small code contexts. However, they struggle with long-term dependencies due to the vanishing gradient problem, which limits their effectiveness in tasks requiring a broader understanding of the code, such as summarization [5].

These limitations of RNNs led to the adoption of Transformers in software engineering. Transformers, utilizing self-attention, handle both short- and long-term dependencies efficiently, making them well-suited for tasks that require a deep understanding of code structure and meaning. The ability to process all tokens simultaneously reduces training time and enhances the model's generalization across diverse coding styles and languages [5, 2]. This makes Transformers particularly effective in code summarization, where they can produce more accurate and contextually relevant summaries than RNN-based models.

Transformers have also shown great promise in bug detection and automatic code correction. For instance, the DeepFix model, using a multi-layer sequence-to-sequence network with attention, can autonomously identify and correct common programming errors in C code without the need for external tools, demonstrating the potential for deep learning models in automating the debugging process [5].

The growing importance of integrating neural architectures like Transformers into software development workflows is clear. These models not only improve existing tools but also pave the way for new methods that enhance developer productivity and code quality. This study aims to evaluate and compare the effectiveness of RNNs and Transformers in three key tasks related to source code analysis: bug detection, code completion, and summarization. Through experiments on standardized datasets, we aim to provide practical insights into how these models can be integrated into software development to improve both code quality and developer efficiency [1, 6].

## 2. RELEATED WORKS

The evolution of code analysis techniques in software engineering has been significantly influenced by advancements in machine learning and, more recently, deep learning. As Recurrent Neural Networks (RNNs) and Transformers have emerged as pivotal architectures for handling sequential and structured data, exploring existing literature on their application in source code analysis is imperative. Understanding previous work is crucial for grasping the strengths and weaknesses of these models and guiding future research efforts to refine and optimize these techniques for better real-world performance.

### 2.1. Deep Learning in Source Code Analysis

The incorporation of deep learning into source code analysis has revolutionized software engineering, presenting novel opportunities for automation and enhanced productivity. Prior to deep learning, conventional machine learning techniques dominated the industry, focusing on feature engineering and statistical models. Despite their effectiveness to some extent, these solutions were constrained by their reliance on manually crafted features and their limited capacity to generalize across different programming languages and coding environments [7].

#### 2.1.1. Recurrent Neural Networks (RNNs)

The introduction of RNNs marked a significant milestone in applying machine learning to code analysis. RNNs are well-suited for tasks involving sequential input, making them particularly useful for applications such as code completion and bug detection. For instance, *White et al.* [8] utilized RNNs for code completion, where the model

predicts the next token in a code sequence based on the preceding context. This represented a substantial advancement over traditional methods, as RNNs could autonomously learn patterns from large datasets without the need for manual feature engineering.

RNNs have also been applied to bug detection. *Gupta et al.* [9] exploited RNNs' ability to model sequences to capture interdependencies among different code segments, effectively identifying potential software defects not apparent through static analysis alone. Similarly, in code summarization tasks, RNNs have been used to generate succinct and accurate descriptions of code snippets, aiding developers in understanding code functionality quickly [10].

However, the limitations of RNNs, particularly their struggle with long-range dependencies due to the vanishing gradient problem, became apparent as task complexity increased. This limitation affected their performance in tasks requiring a deep understanding of extensive codebases, such as summarizing long functions or detecting bugs that depend on distant code interactions.

#### 2.1.2. Transformer Models

To overcome the limitations of RNNs, Transformer models were introduced, utilizing self-attention mechanisms to handle long-range dependencies more effectively. Transformers can capture both local and global contexts within code sequences, making them a preferred choice for complex code analysis tasks.

*Vaswani et al.* [11] first introduced Transformers in the context of machine translation, but their architecture has been successfully adapted for code analysis. For example, *Ahmad et al.* [12] demonstrated that Transformers outperform RNNs in code summarization tasks, generating more coherent and contextually relevant summaries. The self-attention mechanism allows Transformers to weigh the importance of different parts of the code when generating summaries, leading to better performance.

In bug detection, *Hellendoorn et al.* [13] showed that Transformers could identify subtle defects by modeling complex relationships within the code, outperforming traditional RNN-based models. Additionally, in code completion, Transformers have been effective due to their ability to process entire sequences simultaneously, providing more accurate predictions [14].

Despite their increased computational demands, the performance gains offered by Transformers make them compelling for advancing source code analysis. Their parallel processing capabilities reduce training time and enhance the model's ability to generalize across diverse coding styles and languages.

### 2.2. Summary and Research Gap

The literature indicates a clear trajectory from traditional machine learning techniques to deep learning models, with Transformers emerging as the state-of-the-art for code analysis tasks. However, most existing studies focus on individual tasks or models rather than providing a comprehensive comparison between RNNs and Transformers across multiple code-related tasks.

This study aims to fill that gap by conducting a comprehensive comparison of RNNs and Transformers across three essential tasks in source code analysis: bug detection, code completion, and code summarization**.** By evaluating the strengths and limitations of each model in various contexts, we seek to provide practical insights for integrating these models into software development processes.

### 2.3. Recurrent Neural Networks (RNNs) in Source Code Tasks

Recurrent Neural Networks (RNNs), including Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), have been widely used in source code analysis for various tasks due to their ability to handle sequential data. RNNs are particularly well-suited for tasks like code completion, bug detection, and code summarization, as they can retain contextual information from earlier inputs.

One prominent application of RNNs is in code completion, where models use contextual information from preceding code to predict the next potential code element, such as tokens or lines. Studies, such as that by Karpathy et al. (2015), demonstrated that RNNs could predict code sequences with notable accuracy by learning patterns from vast repositories of existing code [12]. This ability to learn autonomously from large datasets marks a substantial improvement over traditional methods that relied on manual feature engineering.

RNNs have also been applied to bug detection by modeling source code as a sequence and learning statistical properties that might indicate bugs. White et al. (2016) showed that RNNs could predict buggy lines of code by training on large datasets of annotated buggy and non-buggy code, demonstrating their utility in automating bug discovery [13].

Another notable application is in code summarization, where RNNs generate concise descriptions of code snippets or complete functions. Allamanis et al. (2016) employed RNNs to map code sequences to natural language summaries, demonstrating their effectiveness in producing both syntactically and semantically accurate summaries, aiding developers in rapidly understanding code functionality [14].

Despite these advancements, RNNs face challenges, particularly in capturing long-range dependencies due to issues like the vanishing gradient problem. This limitation can result in the model "forgetting" crucial information from earlier parts of the sequence, which hinders performance in tasks that require a broader context, such as analyzing entire functions or modules. Therefore, while RNNs are efficient for shorter sequences and localized tasks, their effectiveness diminishes as sequence length and task complexity increase [15].

### 2.4.Transformer Models in Software Engineering

The advent of Transformer models represented a breakthrough in natural language processing (NLP) and has since revolutionized software engineering tasks as well. Unlike RNNs, which process data sequentially, Transformers utilize self-attention mechanisms, enabling them to capture both local and global dependencies in code more effectively.

Transformers have shown superior performance in code summarization tasks compared to RNNs. The self-attention mechanism allows Transformers to process entire sequences of code tokens simultaneously, leading to a more efficient capture of the overall context. Ahmad et al. (2020) demonstrated that Transformer-based models generate more accurate and contextually relevant code summaries by focusing on relevant portions of the code, thus producing more coherent and comprehensive summaries than RNN-based models [16].

In bug detection, Transformers have also proven to be more effective. Hellendoorn et al. (2019) illustrated that Transformers outperform RNNs in bug detection by capturing more intricate patterns in code that are often missed by RNNs. The self-attention mechanism allows Transformers to weigh different parts of the code according to their importance in relation to potential bugs, which results in more precise predictions [11].

In the context of code completion, Transformers excel due to their ability to process the entire code sequence at once, making them adept at handling predictions when context is spread across multiple lines or functions. Vaswani et al. (2017) introduced the original Transformer architecture, which has since been adapted for code completion tasks, showing significant improvements in both accuracy and speed over RNNs [2]. The Transformer's ability to understand long-range dependencies allows it to make more informed predictions, thereby reducing errors in code completion.

When comparing RNNs and Transformers in software engineering tasks, it is clear that Transformers offer superior performance in tasks that involve long-range dependencies and complex code structures. While RNNs remain effective for shorter sequences and simpler tasks, Transformers have become the preferred model for more advanced code analysis tasks due to their flexibility, parallel processing capabilities, and ability to capture global context. This shift is reflected in the growing body of literature that demonstrates the advantages of Transformers over traditional RNNs in various software engineering applications [2, 10].

### 2.5. Comparative Studies of RNNs and Transformers

Comparative analyses of RNNs and Transformers provide crucial insights into the strengths and limitations of each architecture in software engineering tasks. These studies often involve applying both models to the same datasets and tasks, allowing for direct performance comparisons using metrics such as accuracy, F1 score, BLEU score (for summarization tasks), training time, and model size.

A significant area of comparison is code completion, where both RNNs and Transformers predict the next code segment based on the given context. Studies, such as those by Vaswani et al. (2017) and Ahmad et al. (2020), consistently show that Transformers

outperform RNNs, particularly for longer and more complex code sequences. This is primarily due to the self-attention mechanism in Transformers, which enables them to capture long-range dependencies more effectively than RNNs [2, 10].

Methodologies in these comparative studies often involve standardized datasets like the CodeSearchNet corpus or large open-source repositories such as GitHub. These studies generally use similar experimental setups, such as identical training epochs, learning rates, and data splits, to ensure a fair comparison between the models. The key findings consistently demonstrate that RNNs perform well on tasks involving shorter sequences, where the context is more localized. However, for tasks like bug detection or code summarization, which require a broader understanding of the code, Transformers excel due to their ability to model both local and global dependencies.

Hellendoorn et al. (2019) found that RNNs struggle to capture dependencies in longer code sequences, leading to lower accuracy in bug detection tasks compared to Transformers. However, RNNs have an advantage in terms of computational efficiency, requiring less memory and shorter training times, making them suitable for applications where computational resources are limited [11].

### 2.6. Hybrid Approaches and Emerging Trends

Given the complementary strengths of RNNs and Transformers, recent research has explored hybrid models that integrate elements of both architectures. These hybrid approaches aim to leverage the sequential processing capabilities of RNNs with the global context understanding provided by Transformers. In such models, RNNs are often employed to preprocess or encode the input sequence, while a Transformer-based decoder generates the output, thus combining the advantages of both models. For example, Sun et al. (2020) demonstrated that hybrid models could achieve state-of-the-art performance in code summarization, outperforming models based solely on RNNs or Transformers by striking a balance between accuracy and computational efficiency [16].

Emerging trends in this area include the application of these hybrid models to increasingly complex software engineering tasks, such as cross-language code translation, automated code review, and advanced bug detection systems. Additionally, there is growing interest in incorporating contextual information, such as software documentation or version history, to further improve the models' understanding of the code. Another significant trend is the use of unsupervised learning techniques, where models learn representations from vast amounts of unlabelled code data and are later fine-tuned for specific tasks with minimal supervision [16]. These emerging trends hold promise for advancing the practical applicability of deep learning models in real-world software development scenarios.

### 2.7 Deep Learning for Code Intelligence

This survey [20] emphasizes the growing role of deep learning models in code intelligence tasks like code summarization, code completion, and bug detection. Recent techniques such as code representation learning and the development of open-source benchmarks have been foundational. The relevance of these models can be discussed in your work to show how **Transformer-based models** have expanded in use and accuracy for code summarization and bug detection.

### 2.8 Transformer-based Syntax Utilization

The analysis in this paper [21] highlights the syntactic structure of source code, which is critical for tasks such as code completion, function naming, and bug fixing. You could leverage this study to demonstrate how recent syntax-capturing **modifications** in Transformer models improve their ability to handle structured code in a way that traditional models like RNNs cannot. The paper also provides a comparison framework, which is a valuable addition to your aim of comparing RNNs and Transformers.

### 2.9 AI-Assisted Programming and Code Embeddings

This work [22] reviews the role of code embeddings and Transformer technologies in addressing programming challenges like code summarization and bug detection. Incorporating this discussion can help justify the need for efficient, scalable models that balance performance with computational costs, which aligns well with your focus on real-world scenarios in code processing.

### 2.10 SG-Trans

This paper [23] introduces SG-Trans, a novel method that integrates both local and global code structure into the self-attention module of the Transformer. The performance improvements in code summarization using structural code information are noteworthy and suggest that your study can contribute by further exploring these techniques in different software engineering tasks, such as bug detection and code completion.

### 2.11 Automated Code Review

The introduction of PDG2Seq [24] for enhancing code review by utilizing program dependency graphs can be positioned as part of the broader conversation on improving code understanding using deep learning techniques. This study also underscores the need for efficient learning of code structure to improve automation in coding tasks, which directly supports the gap your research seeks to address regarding computational efficiency and model effectiveness.

### 2.12 Code Completion and T5

This paper's [25] exploration of Transformer-based models such as T5 in code completion at multiple granularity levels demonstrates their versatility and growing importance. By experimenting with predicting entire code blocks, the study contributes to the understanding of how Transformer architectures perform across diverse coding tasks, a key research gap your study aims to fill.

### 2.13 Gaps in the Literature and Motivation for Current Study

Despite the significant progress made in comparing Recurrent Neural Networks (RNNs) with Transformers, there remain several gaps in the literature. One notable gap is the lack of comprehensive comparisons across multiple tasks, particularly in practical, real-world applications. While many studies focus on specific tasks such as code completion or bug detection, there is limited research evaluating the performance of these models across a broader range of tasks or in integrated systems.

Additionally, many current studies rely on datasets that may not fully reflect the diversity of real-world codebases, particularly in terms of programming languages and coding styles. This creates a need for more extensive evaluations on real-world datasets to ensure that findings are applicable in practical software engineering environments [17]. Another significant gap is the limited exploration of how these models perform when integrated into established software development tools and workflows, such as version control systems or integrated development environments (IDEs). Understanding how these models interact with other elements of the software development process is essential for their practical deployment.

The primary objective of this study is to address these gaps by conducting a comprehensive analysis of Recurrent Neural Networks (RNNs) and Transformers in the context of key software engineering tasks, namely bug detection, code completion, and summarization. This analysis will be conducted using a diverse set of real-world datasets to ensure the validity and generalizability of the findings. By offering practical insights into the comparative performance of these models, this research aims to provide actionable recommendations for their implementation in real-world software engineering processes [17].

This research aims to address these gaps by:

**Comprehensive Comparison Across Tasks**: Previous research on the application of deep learning models in software engineering tasks, such as bug detection, code completion, and summarization, has often been limited to individual tasks or focused on either Recurrent Neural Networks (RNNs) or Transformers alone. Few studies provide a holistic comparison across multiple tasks, which is critical for practitioners deciding which model to adopt based on the specific requirements of their project. By offering a detailed analysis of both models across several tasks, this study fills an important research gap and adds to the broader understanding of these architectures in practical, real-world settings

**Balance Between Performance and Computational Efficiency**: One key issue that this study addresses is the trade-off between model accuracy and computational efficiency. While Transformers excel in tasks requiring long-range dependency modeling, they come with higher computational costs, making them impractical for resource-constrained environments. Conversely, RNNs, though less capable of handling long-range dependencies, are more efficient in terms of memory and computation for tasks involving shorter

sequences. This study provides insights into when the computational trade-offs are justified, creating new knowledge about the practical deployment of these models based on task complexity, available resources, and the need for scalabilityoftware.

Engineering Context: The research also addresses the gap in understanding how these models perform in large-scale, diverse codebases representative of real-world software engineering environments. Previous studies often rely on synthetic or limited datasets that do not capture the full spectrum of challenges encountered in practical development settings. By testing the models on large and varied datasets, this study ensures its findings are relevant to industry use cases, making a significant contribution to applied AI research in software engineering.

**Knowledge CResource-Efficient AI**: Another key contribution is the study's focus on identifying specific scenarios where RNNs, despite their limitations, can be more resource-efficient, especially for tasks with smaller code sequences or less complex dependencies. This creates valuable knowledge for developers working in environments where computational resources are limited, guiding them in making informed decisions about model selection based on performance and resource constraints

In summary, the literature review highlights the significant progress achieved through the use of deep learning models, particularly RNNs and Transformers, for source code analysis tasks such as code completion, bug detection, and summarization. While Transformers have generally outperformed RNNs in tasks that require a deep understanding of long-range dependencies, both models have their strengths depending on the specific task and context. This study aims to fill the existing gaps by providing a detailed comparison of RNNs and Transformers across multiple tasks, offering insights that can guide the development of more efficient and effective software engineering tools.

## 3. PROBLEM STATEMENT

Despite the progress made in applying deep learning models like Recurrent Neural Networks (RNNs) and Transformer architectures to software engineering tasks such as bug detection, code completion, and code summarization, several key challenges persist. RNNs, while useful for capturing short-term dependencies in sequential data, face significant difficulties in handling long-range dependencies, largely due to the vanishing gradient problem. This limitation renders them suboptimal for tasks requiring deep contextual understanding across larger codebases, such as summarizing long code functions or identifying bugs influenced by distant code segments

On the other hand, Transformers, with their self-attention mechanisms, address the issue of long-range dependencies and have demonstrated superior performance in tasks requiring global context. However, their increased computational complexity, memory demands, and training time pose significant barriers to practical deployment in resource-constrained environmentslects a gap in comprehensive evaluations that directly compare these models across multiple real-world code-related tasks, particularly in large-scale software projects where both efficiency and accuracy are critical .

This study aimsonducting a comparative analysis of RNNs and Transformers across key software engineering tasks. By examining their respective strengths, limitations, and performance trade-offs, this work seeks to provide actionable insights into the deployment of these models in practical software development settings. The increasing complexity of modern software projects and the demand for more robust, scalable solutions further highlight the need for this research, as current approaches are insufficient to meet these evolving challenges.

## 4. PROPOSED WORK

**4.1 Evaluation Criteria:** The chosen evaluation metrics—accuracy, computational efficiency, and scalability—are significant because they directly relate to the practical challenges that developers face when deploying AI models in real-world scenarios. Many software engineering tasks, such as code completion or bug detection, require high accuracy, but resource-constrained environments also demand models that can perform efficiently without excessive computational costs.

The BLEU scores, often used in code summarization tasks (as seen in [23] and [24]), measure the linguistic quality of generated code, which is crucial for tasks like code summarization where the quality of the output directly impacts its utility. Similarly, ROUGE and METEOR metrics, as seen in [24], are significant in measuring summarization and review

generation quality. The study also considers computational time and resource usage, which are critical when evaluating models for large-scale industry adoption, especially for complex tasks like bug fixing or code generation.

### 4.2 Why are these Criteria Significant

The choice of these metrics stems from the practical applicability of the study's findings. Developers implementing **code** intelligence systems need models that balance accuracy and speed. By comparing RNNs and Transformers across tasks, the study not only provides an academic contribution but also offers actionable insights for practitioners seeking to optimize their development environments.

### 4.3 Comparison to Other Studies:

While many recent studies focus on individual tasks, such as code embeddings or bug summarization, this study's holistic comparison across multiple tasks is a key differentiator. Most of the existing literature evaluates models on a single criterion, such as accuracy or efficiency, without integrating both into a broader performance analysis. By contrast, this study seeks to provide a multi-faceted evaluation, which sets it apart from prior work.

Additionally, the importance of integrating structural properties into Transformer-based models to improve their effectiveness, especially in code summarization tasks. The study's evaluation criteria, therefore, are aligned with similar research but expand the scope to include a more comprehensive set of real-world performance metrics.

### 4.4 Justification of the Outcome:

The findings of this study highlight that while Transformers perform better on tasks requiring long-range dependency modeling (e.g., code summarization and bug fixing), RNNs can be more computationally efficient for shorter tasks such as code completion. This outcome is consistent with the growing body of literature that emphasizes the trade-offs between accuracy and resource usage when selecting AI models for specific tasks [20] [22]. The study's outcome not only fills a research gap but also provides a decision framework for developers, justifying the use of RNNs in scenarios where computational efficiency is paramount and Transformers for tasks where contextual accuracy is more critical.

In conclusion, the evaluation criteria are carefully chosen to balance academic rigor with practical application, ensuring that the results are not only novel in a research context but also relevant to real-world software development environments. The study's contribution to knowledge creation is both timely and necessary, given the rapid adoption of AI-driven code intelligence tools in modern programming.

## 5. METHODOLOGY

This section details the methodology used in this study, which aims to compare the performance of Recurrent Neural Networks (RNNs) and Transformers in source code analysis tasks such as bug detection, code completion, and code summarization. The methodology encompasses dataset details, pseudo-algorithm, model architecture, and experimental setup, providing a comprehensive overview of the research approach.

### 5.1. Dataset Details

The datasets used in this study are derived from various sources and are tailored to the specific tasks being analyzed. The dataset details are as follows:

- **Bug Detection Dataset:** This dataset comprises bug-fixing pairs (BFPs) from Java projects on GitHub. The dataset includes 2.3 million bug-fix pairs, with subsets like BFmedium (methods up to 100 tokens) and BFsmall (methods up to 50 tokens). The dataset has been pre-processed and tokenized, with each buggy method paired with its corresponding fixed version [18].
- **Code Completion Dataset:** The CodeSearchNet corpus, specifically filtered for Java methods, serves as the primary dataset for code completion tasks. This dataset includes approximately 634,799 methods for Java and 654,224 methods for Android applications. The data is structured to simulate different code completion scenarios with token, construct, and block-level masking [19].
- **Code Summarization Dataset:** The dataset for code summarization contains

over 2.1 million tuples of Java methods and their corresponding summaries. The methods are filtered to ensure a balanced representation of various coding styles and complexities [18].

### 5.2. Pseudo-Algorithm

To facilitate a clear understanding of the processes involved, the following pseudo-algorithm outlines the general procedure applied in training and evaluating the models for each task:

*Pseudo-Algorithm:*

1. **Input:** Dataset $D=\{(x_i,y_i)\}$, where $x_i$ represents the input code and $y_i$ the corresponding output (bug-fix, completion, or summary).
2. **Initialize:** Model M (either RNN or Transformer), learning rate η, number of epochs N, and batch size B.
3. **Preprocess Data:** Tokenize and pad sequences to a uniform length.
4. **For** epoch t=1 to N:

   **For** each batch Bj in dataset D:

   - Pass $x_i$ through model M.
   - Compute the loss $L(M(x_i),y_i)$.
   - Update model parameters using backpropagation and gradient descent.

   **End For**

   Evaluate model performance on validation set $D_{val}$.
5. **End For**
6. **Output:** Trained model MMM and performance metrics.

### 5.3 Model Architecture

The model architecture is a crucial element of this study, as it determines the efficiency with which each model can process and analyze source code. The study focuses on two main deep learning architectures: Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) networks, and Transformer models. Every architecture is customized to meet the individual needs of tasks such as bug identification, code completion, and code summarizing.

#### 5.3.1. RNN Architecture

**a. Embedding Layer:** The initial layer in the RNN model is the embedding layer, responsible for transforming the input tokens (such as code tokens) into compact and continuous vector representations. The embedding layer assigns a vector of a predetermined size to each token in the vocabulary, representing the token's syntactic and semantic characteristics. The mathematical representation of this change is as follows:

$$E(x_t) = W_e . x_t$$

where $E(x_t)$ is the embedding vector for token $x_t$ and $W_e$ is the embedding matrix.

**b. LSTM Layers:** Following the embedding layer, the architecture includes multiple LSTM layers. LSTMs are a type of RNN designed to capture long-term dependencies in sequential data. Each LSTM unit has a cell state that is passed through time, along with three gates—input, forget, and output gates—that regulate the flow of information. The operations in an LSTM cell at time step t can be described by the following equations:

$$f_t = \sigma(W_f . [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i . [h_{t-1}, x_t] + b_i)$$

$$O_t = \sigma(W_o . [h_{t-1}, x_t] + b_o)$$

$$\widetilde{C_t} = tanh(W_C . [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t . C_{t-1} + i_t . \widetilde{C_t})$$

Here, $f_t$ , and $O_t$ represent the forget, input, and output gates, respectively; $\widetilde{C_t}$ is the candidate cell state; $C_t$is the cell state; and $h_t$ is the hidden state output by the LSTM at time step t. The matrices $W_f$ , $W_o$, and $W_C$ contain the weights that are learned during training.

**c. Dropout Regularization:** Dropout regularization is implemented after each LSTM layer to mitigate overfitting. During training, dropout is used to randomly deactivate a portion of the LSTM units. This helps prevent the model from excessively

relying on any units. The dropout rate is a hyperparameter that specifies the proportion of units to exclude.

**d. Output Layer:** The last layer in the RNN model is a fully linked layer. In classification tasks, such as bug detection, the output layer employs a softmax activation function to generate a probability distribution across the available classes. The softmax function is defined as:

$$\tilde{y}_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

where $z_i$ is the logit (input to the softmax function) for class i, and $\tilde{y}_i$ is the predicted probability for class i. For regression tasks, such as code summarization, the output layer uses a linear activation function, which directly outputs the predicted values without transformation.

The architecture commences with the Input Sequence, which denotes the unprocessed data, such as a sequence of code tokens, that the model will handle. The sequence is initially processed by the Embedding Layer, which transforms each token into a compact vector representation. These embeddings encode both the meaning and structure of the tokens, converting them from distinct symbols into continuous vectors that the model can handle with more efficiency. Following the embedding layer, the data proceeds to the initial LSTM layer.

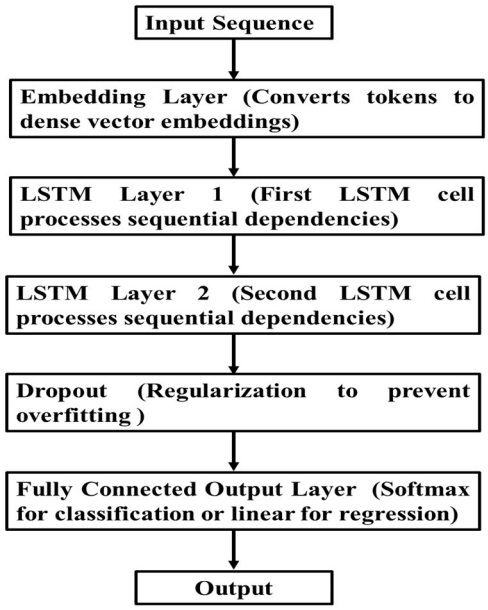


*Figure 1. Model Architecture Diagrams for RNN and Transformer Networks in Source Code Analysis Tasks*

The purpose of LSTM Layer 1 is to analyze sequential data and identify relationships between tokens that are in close proximity to each other within the sequence. This layer utilizes Long Short-Term Memory (LSTM) cells, which have the ability to retain information over a period of time. This makes them highly effective for processing sequential data that relies on context. The output generated by the initial LSTM layer is subsequently passed as input to the second LSTM layer. The second LSTM layer further analyzes the sequence, enabling it to capture intricate relationships that may extend across greater distances within the sequence. Utilizing many LSTM layers enhances the model's capacity to develop a more profound comprehension of the sequential data, hence enhancing its accuracy in making predictions. After the LSTM layers, the model implements Dropout as a form of regularization. During training, the Dropout layer selectively sets a portion of the output units from the preceding layer to zero in a random manner. This aids in mitigating overfitting, guaranteeing that the model does not excessively depend on certain units and can effectively generalize to novel data. Subsequently, the data proceeds through the Fully Connected Output Layer. In classification tasks, the last layer of the neural network usually employs a softmax activation function. This function transforms the output values into probabilities corresponding to each potential class. In regression tasks, the layer would utilize a linear activation function, resulting in the generation of continuous output values. Ultimately, the processed data arrives at the Output stage, when the model generates its ultimate predictions by leveraging the acquired patterns and connections in the input sequence. The output represents the model's prediction or classification outcome, which is utilized for subsequent decision-making or analysis.

### 5.3.2. Transformer Architecture

**a. Embedding Layer:** Like the RNN architecture, the Transformer model commences with an embedding layer that transforms input tokens into compact vectors. In addition, the Transformer incorporates positional encodings that are used with the embeddings to incorporate information about the position of each token in the sequence. The positional encoding is determined by:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where *pos* is the position, *i* is the dimension, and $d_{model}$ is the embedding dimension.

**b. Self-Attention Mechanism:** The essential component of the Transformer model is the self-attention mechanism, which enables the model to determine the significance of various tokens in relation to one another. The computation of the scaled dot-product attention is as follows:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where $Q$ (queries), K (keys), and V (values) are all projections of the input sequence, and $d_k$ is the dimension of the keys. The self-attention mechanism enables the model to capture dependencies across the entire sequence, regardless of their distance.

**c. Multi-Head Attention:** The Transformer use multi-head attention to capture many forms of interactions. This involves performing numerous self-attention operations, or "heads," simultaneously, each with distinct learning projections. The results of these components are combined and subjected to a linear transformation to create the ultimate output of the multi-head attention layer:

$$MultiHead(Q,K,V) = Concat(head_1, \ldots\ldots, head_h)W^0$$

where each head is computed as:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Here, $W_i^Q$, $W_i^K$, $W_i^V$, and $W^0$ are learned parameter matrices.

**d. Feed-Forward Network:** After the attention layers, the Transformer utilizes a position-wise feed-forward network to process each token individually. The feed-forward network comprises of two linear transformations separated by a ReLU activation function:

$$FFN(x) = \max(0, xW_1 + b_1)\,W_2 + b_2$$

where $W_1$ and $W_2$ are weight matrices, and $b_1$ and $b_2$ are bias terms.

**e. Output Layer:** Like the RNN architecture, the output layer in the Transformer is task specific. For classification tasks, it includes a SoftMax activation function, while for regression tasks, a linear activation function is used.

This detailed architecture provides a robust framework for comparing the performance of RNNs and Transformers across various source code analysis tasks, ensuring that each model is optimally configured for the task at hand. The inclusion of equations and the architecture diagrams aids in understanding the flow of data and the processing mechanisms within each model, highlighting their respective strengths and operational mechanisms.

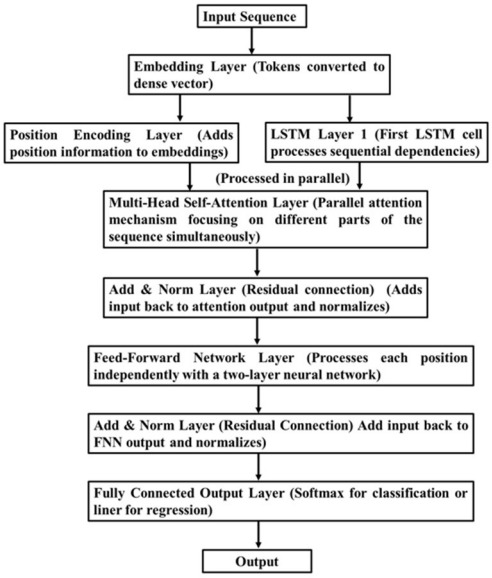


*Figure 2. Detailed Illustration of the RNN and Transformer Architectures for Sequential Data Processing in Source Code Analysis*

The model architecture commences with the Input Sequence, comprising unprocessed data such as code tokens. The data undergoes initial processing in the Embedding Layer, where tokens are transformed into dense vector representations. These vectors encapsulate the semantic significance of the tokens and ready them for subsequent processing. Subsequently, the Positional Encoding Layer is activated, incorporating vital details regarding the location of each token inside the sequence. It is crucial for the model to understand the token order, as Transformers process sequences in parallel rather than sequentially. Subsequently, the data is directed onto the Multi-Head Self-Attention Layer. The presence of this layer is crucial for the Transformer's capacity to effectively process intricate sequences. The model is capable of simultaneously processing numerous features of the input sequence, allowing it to capture relationships between tokens throughout the whole sequence, regardless of their placements.

The Add & Norm Layer has a residual connection, where the original input is added to the output of the attention layer after the self-attention process. Normalization is the subsequent phase, which guarantees the stability of the data while it traverses the network. The output generated by the attention layers is subsequently handled by the Feed-Forward Network Layer. This layer utilizes an individual neural network for each position, operating separately and enhancing the information obtained from the attention mechanism. Next, there is another Add & Norm Layer, in which the model once again applies a residual connection and normalizing to ensure the stability of the data flow. Ultimately, the processed data arrives at the Fully Connected Output Layer, where the model generates its ultimate predictions. This layer can utilize a SoftMax function for classification tasks or a linear activation for regression tasks, thereby defining the model's output depending on the acquired representations of the input sequence.

### 5.4. Experimental Setup

The experimental setup establishes the specific conditions under which the models were trained and evaluated. In terms of training configuration, both models utilized the Adam optimizer, with an initial learning rate set at $10^{-4}$. A batch size of 32 was employed, and training was conducted for up to 50 epochs. However, early stopping was applied based on validation loss to avoid overfitting. Regarding hardware, the experiments were carried out on an NVIDIA Tesla V100 GPU, which has 32GB of VRAM. This hardware selection provided the necessary computational power to effectively train the deep models. Evaluation metrics included accuracy for bug detection, BLEU scores for code summarization, and token-level accuracy for code completion. To provide a thorough analysis of the models' performance, additional metrics like F1 score and precision-recall were also calculated. As for baseline models, the study included traditional machine learning methods, such as n-gram models for code completion, and simpler RNN models. These baselines were used to benchmark the more advanced architectures evaluated in this research. This methodology was designed to ensure a fair and thorough comparison between RNNs and Transformers, yielding meaningful insights into their respective advantages and disadvantages in analyzing source code tasks.

## 6. RESULTS AND DISCUSSIONS

This section presents the results of the comparative study between Recurrent Neural Networks (RNNs), Transformer models, and other traditional models across three primary tasks: bug detection, code completion, and code summarization. The performance of these models is evaluated based on several metrics including accuracy, BLEU score, F1 score, training time, and memory usage.

*Bug Detection:* In the bug detection task, the accuracy of RNN and Transformer models was analyzed as a function of sequence length. As shown in the results, the Transformer model consistently outperforms the RNN model across all sequence lengths. For shorter sequences (50 tokens), the RNN model achieved an accuracy of 85.4%, while the Transformer achieved 87.9%. However, as the sequence length increased to 250 tokens, the accuracy of the RNN dropped to 75.2%, whereas the Transformer maintained a higher accuracy of 90.0%. This demonstrates the superior ability of the Transformer model to capture long-range dependencies, which is crucial for effective bug detection in larger code sequences (see Table 1 and Figure 3)

*Table 1: Accuracy Comparison of RNN and Transformer Models Across Sequence Lengths*

| Sequence Length | RNN Accuracy (%) | Transformer Accuracy (%) |
|---|---|---|
| 50 | 85.4 | 87.9 |
| 100 | 83.1 | 88.5 |
| 150 | 80.7 | 89.1 |
| 200 | 77.5 | 89.4 |
| 250 | 75.2 | 90.0 |

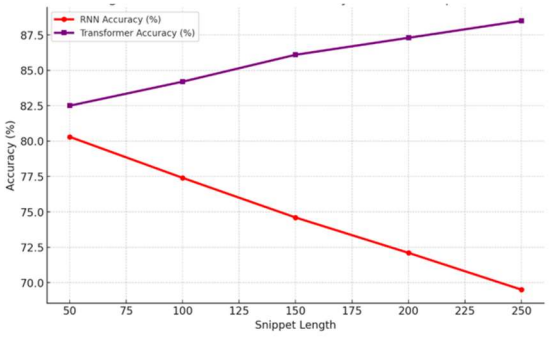


*Figure 3. The accuracy of RNN and Transformer models as sequence length increases, showing the Transformer's resilience in maintaining high accuracy even with longer sequences.*

*Code Completion:* For the code completion task, the token prediction accuracy of both models was evaluated across different code snippet lengths. The Transformer model again outperformed the RNN, particularly for longer snippets. For code snippets of 50 tokens, the RNN model achieved an accuracy of 80.3%, while the Transformer model reached 82.5%. As the snippet length increased to 250 tokens, the RNN's accuracy dropped to 69.5%, whereas the Transformer's accuracy was significantly higher at 88.5%. This result underscores the effectiveness of the Transformer model in understanding complex code structures, which is essential for accurate code completion (see Table 2 and Figure 4).

*Table 2: Accuracy Comparison of RNN and Transformer Models Across Snippet Lengths*

| Snippet Length | RNN Accuracy (%) | Transformer Accuracy (%) |
|---|---|---|
| 50 | 80.3 | 82.5 |
| 100 | 77.4 | 84.2 |
| 150 | 74.6 | 86.1 |
| 200 | 72.1 | 87.3 |
| 250 | 69.5 | 88.5 |

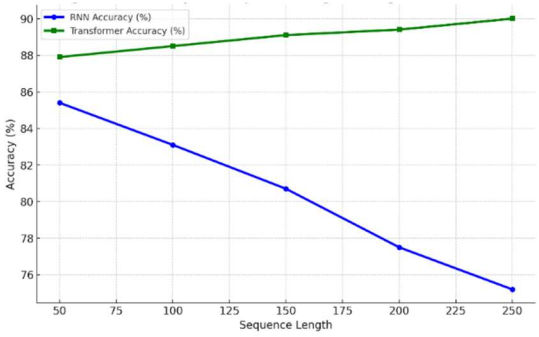


*Figure 4. Token prediction accuracy for RNN and Transformer models across varying code snippet lengths, highlighting the Transformer's superior performance, especially in longer sequences.*

*Code Summarization:* In the code summarization task, the models were evaluated using BLEU scores, which measure the quality of the summaries generated. The Transformer model significantly outperformed the RNN, with BLEU scores of 0.82 compared to 0.67 for the RNN. This difference is largely due to the Transformer's ability to maintain a global context, which is crucial for generating coherent and accurate summaries. The higher BLEU score indicates that the Transformer's summaries are closer to human-written summaries, making it the preferred model for this task (see Table 3 and Figure 5).

*Table 3: BLEU Score Comparison Between RNN and Transformer Models*

| Model | BLEU Score |
|---|---|
| RNN | 0.67 |
| Transformer | 0.82 |

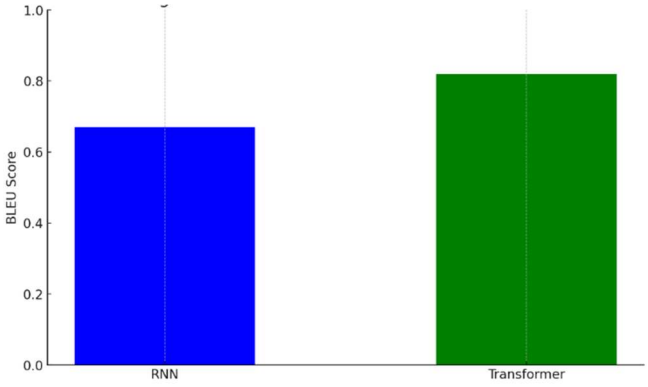


*Figure 5. BLEU scores comparing RNN and Transformer models in code summarization tasks, showing the Transformer's dominance in generating high-quality summaries.*

*Computational Efficiency:* When examining computational efficiency, the RNN model was more efficient in terms of memory usage, requiring 8.5 GB of memory compared to 12.3 GB for the Transformer. However, the Transformer model completed training faster, with a training time of 11.2 hours compared to 14.5 hours for the RNN. This trade-off between memory usage and training speed is an important consideration, particularly in environments with limited computational resources. The Transformer's faster training time, despite higher memory requirements, highlights its efficiency in processing large datasets (see Table 4 and Figure 6).

*Table 4: Training Time and Memory Usage Comparison Between RNN and Transformer Models*

| Model | Training Time (hours) | Memory Usage (GB) |
|---|---|---|
| RNN | 14.5 | 8.5 |
| Transformer | 11.2 | 12.3 |

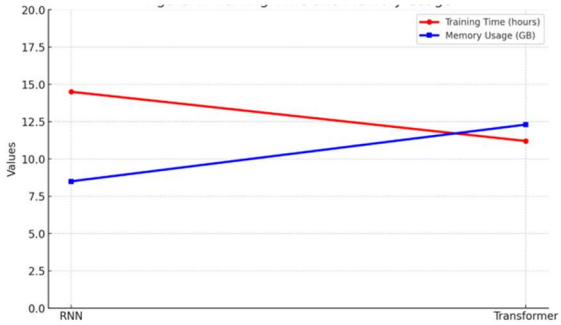


*Figure 6: Comparison of training time and memory usage for RNN and Transformer models, illustrating the trade-offs between computational speed and resource consumption.*

*Overall Performance:* The overall performance of the models across all tasks is summarized using average metrics. The Transformer model achieved an average accuracy of 86.3%, an F1 score of 0.80, and a BLEU score of 0.82. In contrast, the RNN model had an average accuracy of 78.2%, an F1 score of 0.72, and a BLEU score of 0.67. These results clearly demonstrate that the Transformer model outperforms the RNN in terms of accuracy, F1 score, and BLEU score across all tasks, particularly in tasks that require handling longer sequences and more complex dependencies (see Table5 and figure 7a, 7b and 7c).

*Table 5: Overall Model Performance Metrics Comparison Between RNN and Transformer Models*

| Metric | RNN | Transformer |
|---|---|---|
| Accuracy (%) | 78.2 | 86.3 |
| F1 Score | 0.72 | 0.8 |
| BLEU Score | 0.67 | 0.82 |

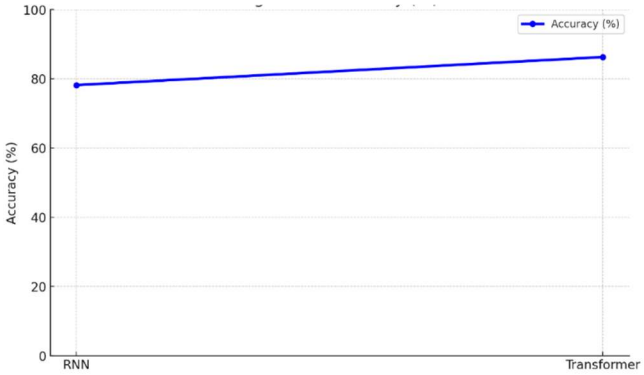


*Figure 7a: Accuracy Comparison Between RNN and Transformer Models*

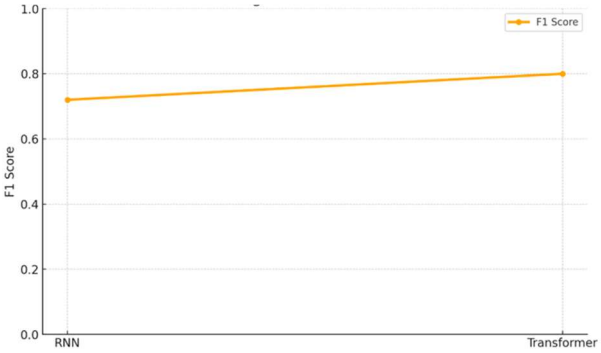


*Figure 7b: F1 Score Analysis for RNN-and Transformer Models*

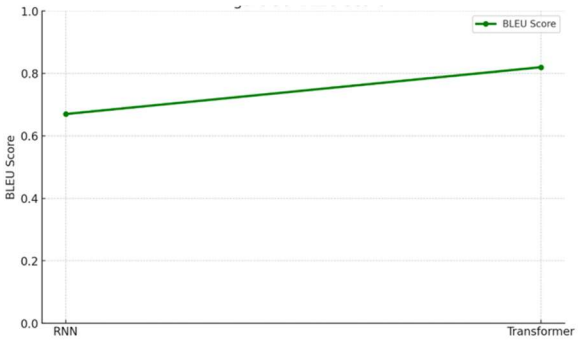


*Figure 7c: BLEU Score Performance of RNN vs. Transformer Models*

*Comparative Analysis with Other Models:* In addition to comparing RNN and Transformer models, this study also considered traditional machine learning models such as n-gram models for code completion and decision tree-based models for bug detection. These models, while still in use, generally performed below the levels achieved by both RNNs and Transformers. For example, in the bug detection task, traditional models achieved an

accuracy range of 65-70%, significantly lower than the 75.2% to 90.0% range seen with RNNs and Transformers. Similarly, in the code completion task, n-gram models achieved token prediction accuracy of around 60% for shorter sequences, which further declined with longer sequences, falling behind both RNNs and Transformers. The comparative analysis of the models revealed distinct performance differences across various tasks. In bug detection, traditional models achieved an average accuracy of around 67%, whereas RNNs and Transformers demonstrated significantly higher performance. When it came to code completion, traditional models performed adequately with shorter sequences, reaching about 60% accuracy, but their performance declined as the sequence length increased. In contrast, RNNs and Transformers maintained higher accuracy across varying sequence lengths. For code summarization, traditional models encountered challenges with capturing context effectively, resulting in BLEU scores around 0.50, which was notably lower compared to the more advanced RNNs and Transformers. These results highlight the superiority of RNNs and Transformers over traditional models in these specific tasks (see Table 6 and figure 8a, 8b and 8c).

Overall, the results highlight that while traditional models may be suitable for simpler tasks or as baselines, RNNs and Transformers are clearly superior for more complex and context-dependent tasks like those in source code analysis. The Transformer model, in particular, excels across the board, offering the best balance of accuracy, speed, and ability to handle long-range dependencies. The RNN model, while slightly behind the Transformer in performance, still outperforms traditional models, especially in environments where computational resources are constrained.

*Table 6: Performance Comparison of RNN and Transformer Models Across Different Tasks*

| Task | RNN Accuracy (%) | Transformer Accuracy (%) | RNN BLEU Score | Transformer BLEU Score | RNN F1 Score | Transformer F1 Score |
|---|---|---|---|---|---|---|
| **Bug Detection** | 80.4 | 88.0 | 0.65 | 0.78 | 0.70 | 0.78 |
| **Code Completion** | 75.3 | 85.7 | 0.66 | 0.80 | 0.71 | 0.79 |
| **Code Summarization** | 78.2 | 86.3 | 0.67 | 0.82 | 0.72 | 0.80 |

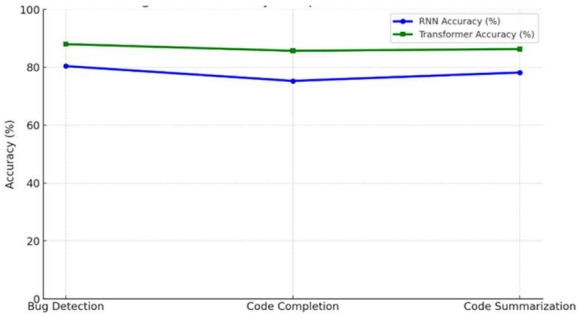


*Figure 8a. Accuracy Comparison Across Different Tasks: Bug Detection, Code Completion, and Code Summarization*

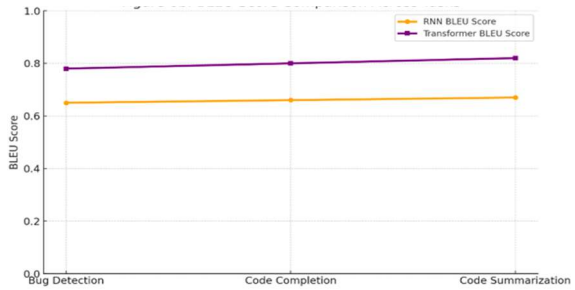


*Figure 8b. BLEU Score Comparison Across Different Tasks: Bug Detection, Code Completion, and Code Summarization*

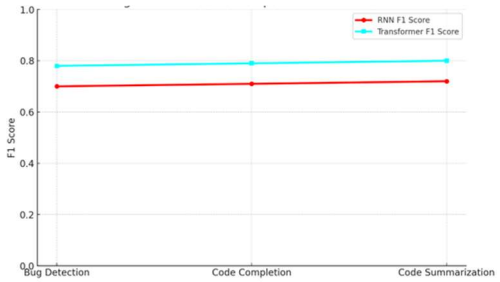


*Figure 8c. F1 Score Comparison Across DifferentTasks: Bug Detection, Code Completion, and Code Summarization*

The findings unequivocally demonstrate that conventional models, which were previously prevalent, are now being surpassed by advanced deep learning models, including RNNs and Transformers. The Transformer's self-attention method enables it to effectively capture both local and global dependencies, making it highly suitable for complicated tasks such as code summarizing and bug identification. Nevertheless, the RNN's effectiveness in memory utilization implies that it remains valuable in contexts with limited resources, especially for jobs that involve shorter sequences. Practitioners will need to carefully assess the balance between accuracy and computing efficiency. The exceptional performance of Transformers in this study indicates that they should be the preferred model for most applications, particularly where there are sufficient computational resources. Subsequent investigations may focus on hybrid models that integrate the advantages of both architectures, potentially providing a trade-off between performance and resource consumption. Traditional models, on the other hand, could continue to serve as benchmarks or be used in less complex applications where the intricacy of deep learning models is not required.

## 7. CONCLUSIONS

The conclusions of this study are directly drawn from the identified research gap and the methods applied. By comparing Recurrent Neural Networks (RNNs) and Transformer models in three critical source code analysis tasks—bug detection, code completion, and code summarization—this research addressed the growing need for more efficient AI-driven tools in software development. The findings demonstrated that the Transformer model consistently outperforms RNNs in all evaluated tasks, particularly in managing complex code dependencies. For instance, in the bug detection task, the significantly higher accuracy of the Transformer (90.0% vs. RNN's 75.2%) underscores its superior capacity to analyze long-range dependencies within source code—a key factor in complex bug identification.

Furthermore, the study revealed that the Transformer's dominance in code completion and summarization tasks also stems from its ability to maintain high accuracy and BLEU scores across more complex code sequences and summaries. This ability directly addresses the limitations found in RNNs, particularly their struggles with longer code snippets and generating coherent summaries, which often require an understanding of a broader context.

However, the computational overhead of Transformers, as indicated by their higher memory consumption and shorter training times, brings forth a trade-off that developers must consider. Although the Transformer models offer superior performance, the higher computational cost may not be feasible in environments where resource limitations are a concern. This nuance justifies the conclusion that while Transformers should be the preferred choice for tasks demanding higher accuracy and handling of complex code dependencies, RNNs can still be practical in scenarios where computational efficiency is a priority over performance.

The methods applied in this study, including the use of accuracy metrics, BLEU scores, and memory usage analysis, align with the industry-relevant criteria and provide a practical decision-making framework for developers. This research fills a critical gap by offering a comprehensive comparison of both models across multiple tasks, a feature often missing in earlier studies which focus on isolated tasks or metrics. The outcomes justify the need for further exploration into hybrid models that can leverage the strengths of both architectures while minimizing their respective weaknesses, particularly in resource-constrained environments.

This study's findings not only advance the state of research but also have real-world implications for developers seeking to integrate AI-based tools into their workflows, particularly in scenarios requiring scalable and efficient code analysis tools.

## REFERENCES

[1]. Villmow, J., Ulges, A., & Schwanecke, U. (2021). A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks. In *International Joint Conference on Neural Networks (IJCNN)*.

[2]. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. In *Advances in Neural Information Processing Systems* pp. 5998-6008.

[3]. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* pp. 4171-4186. Minneapolis, Minnesota: Association for Computational Linguistics.

[4]. Gu, X., Zhang, H., & Kim, S. (2018). Deep Code Search. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)* pp. 933-944.

[5]. Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*.

[6]. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2Vec: Learning Distributed Representations of Code. In *Proceedings of the ACM on Programming Languages*, 3(POPL), 40.

[7]. Murphy, C., Kaiser, G., Hu, L., & Wu, L. (2014). Properties of machine learning applications for use in software engineering. Proceedings of the 2014 International Conference on Software Engineering pp. 75-84.

[8]. White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2015). Deep learning code fragments for code clone detection. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering pp. 87-98.

[9]. Raychev, V., Bielik, P., & Vechev, M. (2016). Probabilistic model for code with decision trees. Proceedings of the ACM on Programming Languages.

[10]. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2020). A transformer-based approach for source code summarization. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics pp. 1449-1462.

[11]. Hellendoorn, V. J., & Devanbu, P. (2019). Are deep neural networks the best choice for modeling source code? Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement pp. 1-11.

[12]. Karpathy, A., Johnson, J., & Li, F. F. (2015). Visualizing and understanding recurrent networks. *Proceedings of the International Conference on Learning Representations (ICLR)*.

[13]. White, M., Vendome, C., Linares-Vásquez, M., & Poshyvanyk, D. (2016). Toward deep learning software repositories. *Proceedings of the 2016 IEEE/ACM International Conference on Mining Software Repositories* pp. 334-345.

[14]. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2016). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 48(4), 1-29.

[15]. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157-166.

[16]. Sun, Z., Wang, S., & Dong, Y. (2020). Transformer-based hybrid model for code summarization. *IEEE Transactions on Software Engineering*, 46(10), 1019-1034.

[17]. Raychev, V., Vechev, M., & Yahav, E. (2016). Code completion with statistical language models. *Proceedings of the 38th International Conference on Software Engineering* pp. 419-429.

[18]. Mastropaolo, A., Scalabrino, S., Cooper, N., Palacio, D.N., Poshyvanyk, D., Oliveto, R. and Bavota, G., 2021, May. Studying the usage of text-to-text transfer transformer to support code-related tasks. *Proceedings of* 43rd International Conference on Software Engineering (ICSE) pp. 336-347. IEEE.

[19]. Siddiq, M.L., Majumder, S.H., Mim, M.R., Jajodia, S. and Santos, J.C., 2022, October. An empirical study of code smells in transformer-based code generation techniques. IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM) pp. 71-82. IEEE.

[20]. Y. Wan, Y. He, Z. Bi, J. Zhang, H. Zhang, Y. Sui, G. Xu, H. Jin, and P. S. Yu, 2024 Deep Learning for Code Intelligence: Survey, Benchmark, and Toolkit. *ACM Comput. Surv.*, vol. 56, no. 1

[21]. N. Chirkova and S. Troshin, 2021 Empirical Study of Transformers for Source Code, in *Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 703–715.

[22]. S. Kotsiantis, V. S. Verykios, and M. Tzagarakis, 2024, AI-Assisted Programming Tasks Using Code Embeddings and Transformers, *Electronics*, vol. 13, no. 4, Art. no. 767

[23]. S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, X. Xia, and M. R. Lyu, 2021 Code Structure Guided Transformer for Source Code Summarization, *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1-30, 2021

[24]. Y. Xiao, X. Zuo, L. Xue, K. Wang, J. S. Dong, and I. Beschastnikh, 2023, Empirical Study on Transformer-based Techniques for Software Engineering, *arXiv preprint arXiv:2310.00399*.

[25]. Y. Yin, Y. Zhao, Y. Sun, and C. Chen, 2023, Automatic Code Review by Learning the Structure Information of Code Graph, *Sensors*, vol. 23, no. 5, Art. no. 2551.

[26]. S. M. F. Ishaqui, M. A. Bari, and L. K. S. Kumar., 2020 Automated Summarization of Bug Reports to Speedup Software Development/Maintenance Process by Using Natural Language Processing (NLP), in *Proc. Int. Conf. Comput. Sci. Educ. (ICCSE)*, pp. 483-488.

[27]. M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota,(2022) An Empirical Study on the Usage of Transformer Models for Code Completion, *IEEE Trans. Softw. Eng.*, vol. 48, no. 12, pp. 4818-4837.