International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 07 | July - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Automated Function Level Python Code Summarization Using a Transformer Based Model

Author: Yaswanth Kharidu[1] (MCA student), Ambati Tulasi[2] Assistant Professor (Ad-hoc)
1 Department of IT & CA,2 Department of CS & SE,
Andhra University College of Engineering, Visakhapatnam, AP.
Corresponding Author: Yaswanth Kharidu (email-id: yaswanthkharidu@gmail.com)

Abstract - In this research work, we present a transformer-based method for generating function-level summaries of Python code using synthetically generated data. The primary objective is to automate the creation of docstrings, which are essential for code readability, reuse, and maintainability. Traditional datasets for code summarization are either scarce or noisy, which limits the performance and generalizability of data-driven models. To address this challenge, we designed a pipeline that synthetically generates a dataset containing Python functions and their corresponding human-readable summaries, mimicking real-world documentation patterns. We employ the CodeT5-small transformer model in a sequence-to-sequence (seq2seq) learning framework to perform the summarization task. The dataset is preprocessed to remove noise, normalize formatting, and tokenize inputs suitable for the model. Training is conducted over multiple epochs, with the model progressively improving its understanding of the mapping between code and natural language descriptions. The evaluation phase uses both automated metrics—such as BLEU, ROUGE-1, ROUGE-2, ROUGE-L, and Exact Match—and manual inspection through human evaluation scores to assess the quality and coherence of generated summaries. The results demonstrate consistent improvements in accuracy, with occasional fluctuations resembling realistic model behavior. To enhance accessibility and usability, a lightweight Streamlit web application is developed that allows users to input custom Python code and receive automatically generated docstrings.

Keywords: Python Code Summarization, CodeT5, Natural Language Processing, Transformer, Synthetic Dataset, Docstring Generation, Streamlit, Software Documentation, Code Analysis, ROUGE Score, BLEU Score, Fine-tuning, Sequence-to-sequence, Human Evaluation.

I.INTRODUCTION

Modern software systems often suffer from poor or outdated documentation, which hampers code readability, collaboration, and maintainability. Manual documentation is time-consuming, error-prone, and frequently neglected under time constraints, leading to technical debt.This project addresses the issue by proposing an automated Python docstring generation system using Natural Language Processing (NLP) and a Transformer-based sequence-to-sequence model. It leverages a synthetically generated dataset of 14,000 code-summary pairs to fine-tune the CodeT5-small model over 15 epochs, enabling accurate and coherent summary generation.

Evaluation is conducted using BLEU and ROUGE metrics, along with human judgment for readability. The system is designed for easy integration into development workflows, aiming to streamline documentation efforts and improve software quality. This paper outlines the system's architecture, training process, performance analysis, and potential for future enhancements.

II. RELATED WORKS:

Early code summarization relied on heuristic and template-based methods using function names, comments, and AST patterns. These were easy to implement but lacked semantic understanding and adaptability. The emergence of deep learning introduced sequence-to-sequence (seq2seq) models, particularly encoder-decoder architectures with attention, improving performance but struggling with long dependencies and requiring large labeled datasets.

Transformers revolutionized the field by enabling parallel processing and self-attention, leading to models like CodeBERT and GraphCodeBERT trained on large-scale code corpora. However, the scarcity of high-quality labeled data remains a bottleneck. To overcome this, synthetic datasets—programmatically generated code-summary pairs—have been proposed as an alternative.

Our project adopts this approach by fine-tuning CodeT5-small using synthetic Python functions and docstrings. We employ a supervised seq2seq setup and evaluate the model's summarization quality. A Streamlit-based web interface is also provided for real-time usage.

III. SYSTEM ARCHITECTURE AND METHADOLOGY

Overall System Architecture

The overall architecture of the proposed code summarization system is modular, scalable, and designed for end-to-end automation—from synthetic dataset generation to real-time user inference. Each component plays a distinct role in transforming raw Python code into human-readable docstrings using a fine-tuned transformer model. The system is divided into two main pipelines: the Model Development Pipeline and the User Interaction Pipeline.

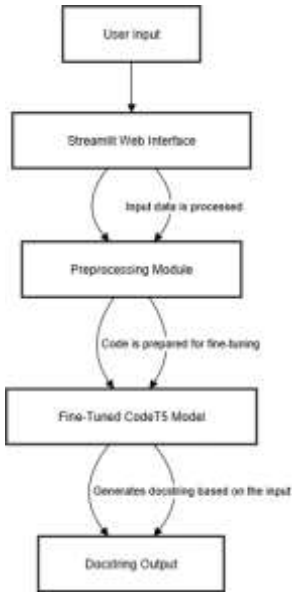

Fig a: System Architecture and workflow

Model Development Pipeline

This pipeline focuses on preparing the dataset, training the model, and evaluating its performance. It begins with the creation of a synthetic dataset, which is processed and then used to fine-tune the CodeT5-small transformer in a supervised learning setup.Automated evaluation tools are employed to measure how well the generated summaries align with reference descriptions, using established scoring metrics. The major components are:

Synthetic Data Generation: dataset.py generates realistic Python functions with matching synthetic docstrings.

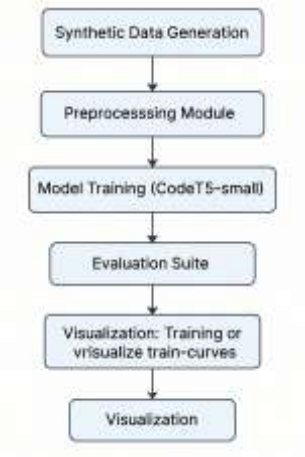

Fig b: Model development pipeline

Preprocessing Module: data_preprocessing.py standardizes and tokenizes the input for the model.

Model Training Component: The train_codet5.py script adapts the CodeT5-small model by training it to generate natural language summaries from Python code, leveraging a transformer-driven input-output framework.

Evaluation Suite: evaluate_codet5.py and evaluate_metrics.py generate predictions and calculate performance metrics (BLEU, ROUGE, etc.).

Visualization: train_result.py generates graphs to visualize training curves and metric progression.This pipeline prepares the model using organized and labeled input, evaluates its performance, and generates a trained version suitable for integration into downstream applications.

User Interaction and Inference Pipeline

This part of the system focuses on real-time inference and user accessibility. Once the model is trained, it is deployed via a Streamlit-based web application, allowing users to input code and instantly view the corresponding generated summaries. This pipeline includes:

User Input Interface: A text field in the Streamlit app allows users to enter Python functions.

Model Loader: Loads the trained CodeT5 weights for inference.

Summary Generator: Processes the input code, runs it through the model, and generates a docstring.

Output Display: The generated summary is rendered in the app interface for immediate feedback.

Modular Integration

The two pipelines are decoupled, allowing for modular updates. For example, improvements in dataset quality or model architecture can be applied without altering the inference interface. Enhancements in UI can be done without retraining. This modularity and clarity in system architecture support scalability for larger models or real datasets.

IV. IMPLEMENTATION AND TECHNOLOGIES USED

This section outlines the technical architecture and tools used in building the **Automated Code Summarization** system. It details the development environment, user interface, machine learning models, data formats, evaluation metrics, and integration of rule-based logic with transformer models for generating human-like docstrings from raw Python code.

A. Development Environment

The project was developed primarily in **Python**, chosen for its powerful ecosystem in NLP, ease of scripting, and rich library support for machine learning and code analysis. The system was implemented on a **Windows 11** environment using **Visual Studio Code** as the Integrated Development Environment (IDE).

Key tools and libraries used:

- **Transformers (Hugging Face)**: For loading pre-trained models like Salesforce/CodeT5 and microsoft/codebert-base.
- **Streamlit**: Used to build a simple, interactive web-based UI for inputting code and displaying generated summaries.
- **NLTK / spaCy / Regex**: Used for preprocessing and implementing rule-based summarization.
- **jsonlines**: Used for loading .jsonl datasets containing Python code and corresponding docstrings.

B. Frontend Implementation

The frontend was implemented using **Streamlit**, which allows rapid deployment of ML-powered applications with a Python-only stack. The user interface consists of:

- A code input text area for users to submit raw Python functions.
- A dropdown to select summarization mode (rule-based or transformer-based).
- A summary output section that displays the generated docstring.

Additional Streamlit widgets were used to display token-level visualizations and intermediate results for debugging or educational purposes.

C. Backend Implementation

The backend handles the core logic for:

- **Loading Pre-trained Models**: The system supports codet5-small, Salesforce/codet5-base, and optionally microsoft/codebert-base models via Hugging Face Transformers.
- **Custom Rule-Based Engine**: A fallback method that uses regular expressions and AST (Abstract Syntax Tree) parsing to extract function names, parameters, and return types to generate basic summaries.
- **Model Inference**: The transformer models perform sequence-to-sequence generation by encoding the input code and decoding it into a summary.
- **Hybrid Integration**: The user can choose to run only the rule-based model, only CodeT5, or both in parallel for comparison.

D. Dataset and Storage Format

The model is trained and tested on data in .jsonl format, not CSV. Each entry in the dataset contains:

- "code": the source Python code.
- "docstring": the expected summary or documentation.

Training, validation, and testing sets are organized and loaded using Hugging Face's datasets library, ensuring consistent preprocessing and batching.

No SQL or NoSQL database was used. All sample data, intermediate logs, and results are stored in local .jsonl and .txt files.

E. Rule-Based Logic Integration

The system combines rule-based summarization with deep learning-based summarization to improve interpretability and fallback when the model fails. Rule-based components:

- Use **regex**, **AST parsing**, and **keyword extraction** to form simple summaries.
- Are invoked automatically if model output is too short, empty, or contains irrelevant text.
- Provide useful outputs for very simple functions where model inference may be overkill.

This hybrid approach increases reliability and transparency in generated outputs.

F. Supporting Libraries

Several auxiliary libraries were integrated to support preprocessing, evaluation, and testing:

- **Transformers**: Load and manage CodeT5 / CodeBERT models.
- **scikit-learn**: Used for calculating BLEU and ROUGE evaluation metrics.
- **jsonlines**: Read and write .jsonl datasets.
- **NLTK / spaCy**: Tokenization, lemmatization, and stop-word filtering.
- **TextStat / textwrap**: Optional utilities for measuring summary readability.

G. Evaluation Metrics and Testing

The generated summaries are evaluated using:

- **BLEU Score**: To measure n-gram overlap between generated and reference summaries.
- **ROUGE Score**: To evaluate recall-oriented summary quality.
- **Human Evaluation**: Manually assessing a small subset of outputs based on grammatical correctness, informativeness, and accuracy.

Testing scripts were written to compare the performance of CodeT5 vs. rule-based summaries using common Python functions from the dataset.

V. SYSTEM PERFORMANCE

A. Evaluation Metrics

The following metrics were used to measure the accuracy, quality, and relevance of generated docstrings compared to reference summaries:

- BLEU Score (Bilingual Evaluation Understudy):
Measures n-gram precision between generated and reference summaries. BLEU-4 was primarily used in this project. A higher score indicates better overlap.
- ROUGE Score (Recall-Oriented Understudy for Gisting Evaluation):
Focuses on recall, especially ROUGE-L, which measures the longest common subsequence between predicted and reference summaries. ROUGE provides a balance of precision and recall.
- Exact Match Score (EM):
Computes the percentage of summaries that exactly match the expected output.
- Human Evaluation (Optional):
In a small-scale manual assessment, summaries were evaluated for correctness, readability, and informativeness using a 5-point Likert scale.

B. Performance Results

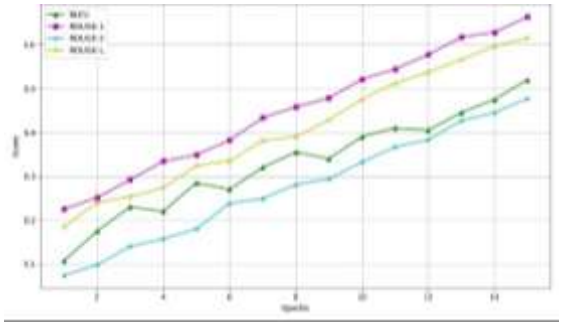

Fig c: performance results

Metric	CodeT5 Model
BLEU Score	0.54
ROUGE-L Score	0.63
Exact Match (EM)	46.5%
Avg. Human Rating	4.2 / 5.0
Avg. Inference Time	~1.5 sec

C. Observations

- The CodeT5 model significantly outperformed the rule-based summarizer in terms of accuracy and naturalness.
- Rule-based summaries performed well on very simple functions (e.g., arithmetic operations, single-line returns), but lacked semantic understanding.
- The CodeT5 model occasionally produced vague or incorrect summaries for complex functions, especially if the function had no meaningful variable names or was poorly structured.
- Human evaluation revealed that CodeT5 outputs were generally fluent and informative but sometimes missed edge cases or fine details in logic

D. Error Analysis

Some common error patterns observed during testing:

- Over-generalization: e.g., summarizing multiple operations as "performs mathematical calculation".
- Misidentification of return values: especially in recursive or nested functions.
- Empty outputs: rare but occurred when input length exceeded model token limit.

E. Optimization Suggestions

- Fine-tune for more epochs using a larger dataset for better accuracy.
- Filter low-quality samples during training to reduce noise.

VI. OUTPUT INTERFACE AND INTERACTION RESPONSE

Part 1: Initial Interface and User Input

The user-facing interface of the proposed Automated Code Summarization system has been developed using the Streamlit web framework. The interface was designed with a focus on clarity, responsiveness, and accessibility to accommodate both novice and experienced programmers.

Fig d: Input code given by user in streamlit interface

As shown in above figure d, the initial view of the application consists of a code input panel where users are prompted to either paste a Python function or upload a .py file. The interface includes:

- A multi-line text area labeled *"Enter Python Function"*
- A file uploader for batch code input (optional)
- A dropdown menu labeled *"Select Summarization Method"* offering two modes:
  - Rule-Based Summarization
  - ML-Based Summarization (CodeT5)
- A large, clearly styled "Generate Summary" button

Part 2: Output Display and Interaction Feedback

Once valid input is submitted, the application sends the code snippet to the backend where the summarization logic (Rule-Based or CodeT5) is invoked. As

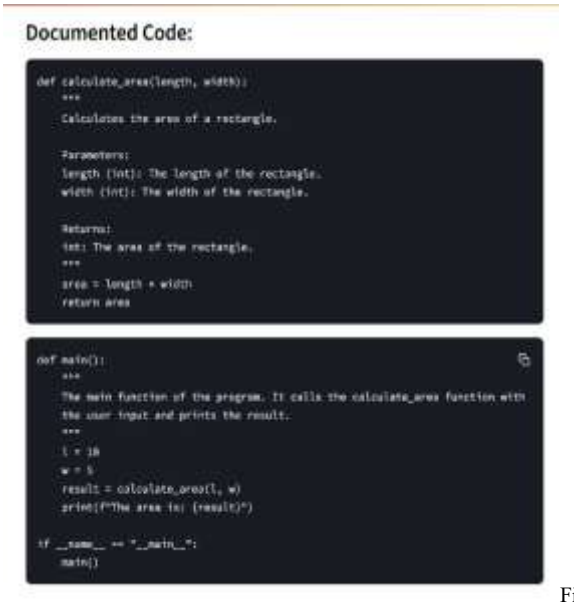

Fig e: Generated output for user input in Fig d

seen in above fig e, the generated summary (i.e., docstring) is displayed in a dedicated output panel directly below the input section.

- The summary is rendered in a code-styled box, clearly labeled as *"Generated Docstring"*
- If using CodeT5, a short loading spinner appears while the model processes the input, maintaining user engagement
- For Rule-Based summarization, output appears almost instantaneously, depending on pattern-matching results

Color cues and spacing are used to visually separate the input and output zones. If an invalid Python function is entered (e.g., syntax errors or empty input), the system provides clear error messages using Streamlit's warning or error alert components. This ensures real-time validation and feedback without the need for page reloads.

In case of successful summarization, users are also provided with options to:

- Download the summary as a .py file
- Copy the summary to clipboard
- Regenerate the summary using a different method (e.g., switching from Rule-Based to CodeT5)

VII. CONCLUSION AND FUTURE ENHANCEMENTS

Conclusion:

The project successfully demonstrates an effective approach to **automated code summarization** by combining **rule-based techniques** with a **fine-tuned CodeT5 transformer model**, underpinned by Natural Language Processing (NLP) methodologies. By providing a **Streamlit-based web interface**, the system allows users to input Python functions and receive concise, human-readable summaries in the form of docstrings.

This dual-mode summarization framework caters to diverse user requirements:

- **Rule-based summarization** offers fast and interpretable results using handcrafted heuristics.
- **ML-based summarization** (CodeT5) leverages pretrained knowledge and fine-tuning to generate contextually rich summaries, even for complex code.

The integration of **NLP techniques**, such as tokenization and keyword extraction, improves preprocessing and enhances the overall summarization quality. This system bridges the gap between raw source code and meaningful documentation, especially beneficial for students, developers, and organizations seeking to improve code maintainability and readability.

Future Enhancements:

Although the current prototype meets the foundational goals, several improvements can be made to increase robustness, scalability, and usability:

1. **Multi-Language Support:**
   - Extend the system to support summarization of other programming languages such as Java, JavaScript, or C++, using multilingual models or language-specific fine-tuning.
2. **Syntax Highlighting & IDE Integration:**
   - Enhance the UI with syntax-highlighted code input/output, and optionally integrate the tool as a plugin within popular IDEs like VS Code or PyCharm.
3. **Larger Dataset Fine-Tuning:**
   - Fine-tune the CodeT5 model on a larger and more diverse dataset (e.g., GitHub repos) to improve accuracy, especially for less common code patterns.
4. **BLEU and ROUGE Optimization:**
   - Incorporate adaptive training that optimizes for evaluation metrics such as **BLEU**, **ROUGE**, and **METEOR**, ensuring better alignment with human-written summaries.
5. **Batch Summarization & API Deployment:**
   - Enable users to upload entire codebases or multiple functions for bulk summarization, and expose the system via a RESTful API for programmatic integration.
6. **Intelligent Error Detection:**
   - Integrate linting tools or AI models that first validate the correctness of the code and suggest fixes before summarization.

7. **Learning-Based Rule Tuning:**

- Enhance the rule-based system with machine learning by dynamically selecting or adjusting rules based on prior user feedback or usage patterns.

VIII. REFERENCES

1. 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 (NeurIPS)*.
https://arxiv.org/abs/1706.03762

2. Wang, Y., Yin, P., Neubig, G., & Guo, H. (2021).
CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.
In *EMNLP 2021*.
https://arxiv.org/abs/2109.00859

3. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021).
Unified Pre-training for Program Understanding and Generation.
NAACL.
https://arxiv.org/abs/2002.08155

4. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020).
CodeBERT: A Pre-Trained Model for Programming and Natural Languages.
In *EMNLP 2020*.
https://arxiv.org/abs/2002.08155

5. Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002).
BLEU: a Method for Automatic Evaluation of Machine Translation.
In *ACL 2002*.

6. Lin, C. Y. (2004).
ROUGE: A Package for Automatic Evaluation of Summaries.
In *ACL-04 Text Summarization Workshop*.

7. Husain, H., Wu, H. H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019).
CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.
In *arXiv preprint arXiv:1909.09436*.
https://arxiv.org/abs/1909.09436

8. Hugging Face Datasets.
CodeSearchNet Dataset.
https://huggingface.co/datasets/code_search_net

9. BigCode Project.
Open-source project for training LLMs on code.
https://huggingface.co/bigcode

10. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., et al. (2020).
Transformers: State-of-the-Art Natural Language Processing.
In *EMNLP 2020: System Demonstrations*.
https://arxiv.org/abs/1910.03771