



Research Article

A Hybrid Neural-Symbolic Framework for Reliable Automated Source Code Documentation

Adewuyi Joseph Oluwaseyi¹ , Kareem Temiloluwa Faith² , Odelola Precious Ayooluwa^{3*}  and Oluwafemi Oluwasemilore Esther⁴ 

^{1,2,3&4}Department of Computer Science, School of Computing, Babcock University, Ogun State, Nigeria

Article Information

Abstract

Article History

Received: 19 May 2026

Revised: 25 June 2026

Accepted: 29 July 2026

Published online: 16 August 2026

Keywords

Code Summarization

Automated Documentation

CodeBERT

CodeT5

Deep Learning

Transformer Models

Software Engineering

Nigeria

Correspondence*

odelolap7852@student.babcock.edu.ng

ORCID

Adewuyi Joseph Oluwaseyi 

<https://orcid.org/0009-0000-2325-7325>

Kareem Temiloluwa Faith 

<https://orcid.org/0009-0006-9922-892X>

Odelola Precious Ayooluwa 

<https://orcid.org/0009-0001-1315-1524>

Oluwafemi Oluwasemilore Esther 

<https://orcid.org/0009-0008-8413-6170>

Inadequate code documentation is a persistent challenge in software development. Time constraints often lead developers to neglect commenting, resulting in poorly documented codebases. This undermines knowledge sharing, reduces maintainability, and weakens team collaboration. Over time, such neglect erodes software sustainability, as critical knowledge becomes confined to individual developers rather than embedded within the code itself. This study presents CodeSage, an automated system for generating inline code comments and docstrings via a retrieval-augmented architecture. CodeBERT (125M parameters) retrieves semantically similar examples from a FAISS index of 50,000 code-documentation pairs, while CodeT5 (220M parameters) generates natural language documentation. A modular parser supports five languages: Python, JavaScript, TypeScript, Java, and C++. Evaluation used BLEU, ROUGE-L, METEOR, and BERTScore on 80 Python code-comment pairs from CodeSearchNet, alongside human evaluation across four quality criteria. The system achieved a BERTScore F1 of 0.7748, METEOR of 0.2978, ROUGE-L of 0.2585, and BLEU of 0.1583, consistent with fine-tuned code summarization benchmarks. Human evaluators rated accuracy and readability at 4.6/5.0, conciseness at 4.5/5.0, and completeness at 4.3/5.0. Mean generation time was 1.17 seconds per sample with a 100% success rate. CodeSage demonstrates that hybrid retrieval-augmented generation with domain-specific transformer models produces semantically accurate, readable documentation across multiple programming languages. It offers a practical productivity tool for Nigerian academic and professional software environments, with future work targeting domain-specific fine-tuning and expanded multilingual support.

© 2026 Centre for Research and Innovation (CRI). This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

I. INTRODUCTION

Documentation functions as a fundamental component that establishes software quality because it enables ongoing maintenance, knowledge transfer, and teamwork among developers. Inline code comments deliver essential information that explains the reasons behind implementation and design choices and their relationship with other code elements to developers who use the codebase as their primary reference system [1]. Documentation practices face ongoing neglect in both professional and educational environments despite their crucial value because time limitations and

developer choices lead people to see documentation as work that does not produce value [2].

The absence of proper documentation results in significant negative impacts that extend across multiple areas. Development teams spend excessive amounts of time trying to understand new code systems, while knowledge gaps arise when developers leave their assigned work, leading codebases to become more complex and unstable through the process known as code decay. The educational setting becomes more difficult because students create software products without learning proper commenting methods.

Deep learning and transformer-based natural language processing systems now provide new solutions for automated code summarization, enabling the computational creation of natural language descriptions from source code. CodeBERT [3], GraphCodeBERT [4], and CodeT5 [5] function as landmark models that attain exceptional results on code-to-text generation tasks through extensive pretraining on linked code-language corpora to create advanced cross-modal representations. The capabilities of retrieval-augmented generation (RAG) methods are maximized by generating content based on semantically similar examples, resulting in better contextual accuracy while decreasing instances of hallucination [6].

Existing systems have multiple fundamental deficiencies because they do not support multilingual use, show restricted capacity to detect developer intent through practical evaluation, and have not been tested in actual Nigerian academic software development environments. This study introduces CodeSage, an AI-based documentation system designed to overcome existing system deficiencies. The system combines CodeBERT-based semantic retrieval with CodeT5-based sequence generation through a hybrid architecture that operates with a modular language parser and FastAPI/Next.js deployment stack. CodeSage targets five programming languages and is evaluated through both automated metrics and structured human assessment, contributing a practical, deployable solution to the persistent challenge of software documentation.

II. LITERATURE REVIEW

A. Early Neural Approaches to Code Comment Generation

Automated code documentation initially relied on rule-based and template systems that required developers to use method names and parameter lists to create documentation comments. The standardized documentation formats introduced by Javadoc and Doxygen required developers to create their own valuable content. Data-driven methods emerged with the introduction of neural sequence models. LeClair and McMillan [7] showed that trained encoder-decoder neural networks could create function summaries from large Java code-comment datasets, establishing the first benchmark for the domain.

LeClair *et al.*, extended this research by using Abstract Syntax Trees (ASTs) as input representations, allowing models to understand code structure through its hierarchical organization rather than processing only basic token sequences. DeepCom (Hu *et al.*, 2018) introduced formal guidelines for AST traversal strategies to enhance performance on standard benchmarks through BLEU and METEOR score improvements. The first neural models showed three main weaknesses: restricted ability to process different contexts, difficulties with long-range dependencies, and superficial summaries that included only method names instead of program logic. These problems led developers to implement advanced transformer-based architectural designs.

B. Transformer-Based and Pre-Trained Models

The introduction of the Transformer architecture by Vaswani *et al.*, (2017) marked a decisive turning point in sequence modelling because it enabled models to capture dependencies across any distance in token sequences through self-attention mechanisms. This innovation directly underpins the most capable code documentation models in current use. CodeBERT [3] built on the RoBERTa framework and was trained on six programming languages using masked language modelling and replaced-token detection objectives to create potent cross-modal representations connecting programming languages with natural language. GraphCodeBERT [4] expanded this scope by adding data-flow graphs to the pre-training process, enabling the model to gain specific knowledge of how variables interact in programs and during execution. The code generation system CodeT5 [5] used identifier-aware pre-training, enabling the model to recognize code identifiers as separate elements from natural language tokens during sequence-to-sequence generation. This design choice yielded more precise and contextually grounded documentation results than general-purpose sequence models. The researchers behind PLBART used denoising pre-training on both code and natural language, enabling results that matched performance across multiple software engineering tasks. Transformer-based systems achieved better results than their RNN-based predecessors on the CodeSearchNet benchmark, which used BLEU, METEOR, and ROUGE-L metrics to evaluate performance. Implementing these systems presents two main challenges: high computational-resource requirements and decreased performance on domain-specific or unfamiliar codebases that require fine-tuning.

C. Retrieval-Augmented and Hybrid Generation Approaches

Researchers investigated retrieval-augmented frameworks because generative models can create generic documentation and documentation errors. The Retrieve-and-Refine framework from Wei *et al.*, [2] used exemplar retrieval to enhance neural generation by giving the generator access to actual documentation retrieved from related code. The retriever-generator model developed by Le *et al.*, [6] enables simultaneous training that optimally retrieves and generates content to create more accurate comments through a combined process rather than separate pipeline functions. Studies by Shahbazi *et al.*, (2021) demonstrated that using API documentation as additional reference material helps create more precise descriptions. Hybrid systems produce better contextual relevance than complete generation systems, but their performance depends on system effectiveness, and they create additional processing demands that restrict real-time operation. Corpus curation plays a vital role in system performance because noisy retrieved examples can reduce generation quality.

D. Human-Centred Evaluation and Large Language Models

Researchers now focus on assessing how useful software documentation is to developers during their work instead of

evaluating documentation only through its original performance indicators. The study by Mastropaolo *et al.*, [8] used empirical research to examine comment completion during developer tasks, showing a gap between high automated metric scores and actual usefulness. Research by Su *et al.*, (2022) proposed breaking comments into three parts: what, how, and why, because high-quality documentation needs to explain developer goals instead of simply stating code function. Research by Winkler *et al.*, (2024) introduced composite readability measures for test code comments, showing that developers need both stylistic naturalness and semantic correctness before using the comments. The introduction of large language models (LLMs), including T5, BART, and GPT-based systems, has established a new standard for generation quality. Ghale and Dabbagh [9] showed that domain-adaptive fine-tuning of T5 and BART leads to better summarisation results than using general pre-trained models. LLM-based systems present three major issues: high computational costs, hallucination problems in low-resource environments, and the need for evaluation systems that align with developer satisfaction with the results. These findings motivate the development of CodeSage, which uses a hybrid system to achieve high semantic accuracy and readable output while running on standard computing systems, with effectiveness assessed through both automated testing and human expert evaluation.

E. Research Gaps Addressed by This Study

The literature review shows that CodeSage specifically addresses several ongoing research needs in the field. First, most existing systems are evaluated exclusively on Java or Python codebases drawn from large international repositories, with limited attention to the multi-language needs of developers in emerging technology contexts such as Nigeria. Second, the dominant use of lexical evaluation metrics (BLEU, ROUGE) for benchmarking can conceal semantic alignment deficiencies, which BERTScore and human assessment methods can identify more effectively. Third, the trade-off between generation quality and deployment feasibility has been underexplored because state-of-the-art systems require both GPU infrastructure and complex fine-tuning pipelines that academic institutions with limited resources cannot effectively implement.

CodeSage addresses these gaps by combining retrieval-augmented generation with structured template-based documentation across five languages, evaluating the system through both automated semantic metrics and structured human assessment, while enabling use on standard CPU systems with dedicated hardware resources without exceeding interactive latency limits.

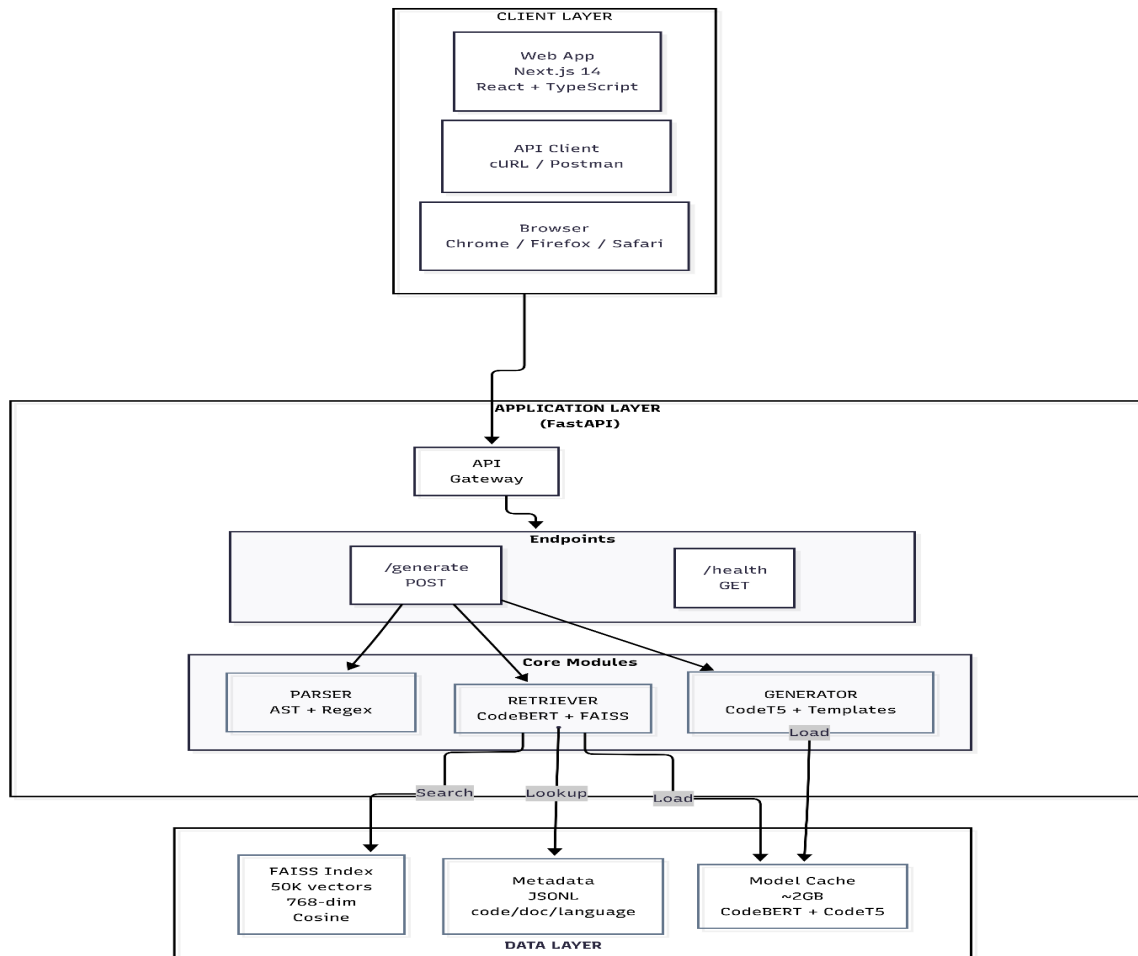


Fig.1 High-level monolithic architecture of the CodeSage system

III. MATERIALS AND METHODS

A. Research Design

The study employs an engineering research framework consisting of four distinct phases that proceed in a logical sequence.

The first phase focuses on requirements analysis and technology selection, while the second involves deep learning model evaluation and selection. The third phase involves creating a full-stack system, which the fourth phase assesses through both quantitative and qualitative methods of documentation evaluation. The study did not require preregistration procedures. The study provides all code,

model weights, and evaluation datasets through publicly available resources under open-source licenses.

B. System Architecture

CodeSage implements a monolithic, single-server design that includes three distinct functional components: a Parser, a Retriever, and a Generator. The system uses FastAPI to create a RESTful API backend accessed by Next.js frontend applications. The data layer contains a FAISS vector index that holds 50,000 code-documentation embeddings and a JSONL metadata storage system. The complete system architecture is presented in Figure 1. The technology stack is summarised in Table I.

TABLE I CODESAGE TECHNOLOGY STACK

Layer	Technology	Version	Purpose
Frontend	Next.js / React / TypeScript	14.x / 18.x / 5.x	Web application UI
Frontend	Tailwind CSS / Framer Motion	3.x / 11.x	Styling and animations
Backend	Python / FastAPI / Uvicorn	3.10+ / 0.109+ / 0.27+	API server and ASGI runtime
ML	PyTorch / HuggingFace Transformers	2.1+ / 4.36+	Deep learning inference
Vector DB	FAISS	1.7+	Approximate nearest-neighbour search

C. Deep Learning Models

CodeSage uses two transformer-based models that operate as complementary systems, as shown in Table II. The system uses CodeBERT from the microsoft/codebert-base model as the retrieval encoder. Its RoBERTa-based architecture, trained on six programming languages using masked language modelling and replaced-token detection, produces 768-dimensional contextual embeddings representing both syntactic and semantic properties of source code [3]. The WordPiece tokenizer processes input code by dividing it into tokens and creates embeddings through mean pooling, which

combines attention-weighted hidden states before L2 normalisation for cosine similarity calculation. CodeT5 from Salesforce/codet5-base-multi-sum functions as the generation model. CodeT5 uses the T5 encoder-decoder architecture, which includes identifier-aware pre-training on CodeSearchNet, to perform sequence-to-sequence code-to-text translation. The system generates module-level documentation based on the prompt "Summarize this {language} script in one sentence: {code}" using greedy decoding with a repetition penalty of 2.5 and a no-repeat n-gram size of 4.

TABLE II MODEL SELECTION SUMMARY

Model	Architecture	Parameters	Role	Input	Output
CodeBERT	RoBERTa-based encoder	125M	Retrieval	Code snippet	768-dim vector
CodeT5	T5 encoder-decoder	220M	Generation	Code snippet	Natural language

D. Dataset

The primary pre-training corpus for both models is CodeSearchNet [10], which provides 6 million code-documentation pairs obtained from GitHub repositories. The study employed Microsoft and Salesforce pre-trained model weights to perform zero-shot and fine-tuned inference on this corpus without additional fine-tuning efforts. The FAISS index used a custom retrieval subset of 50,000 code-documentation pairs selected from the CodeSearchNet corpus according to these language distribution numbers: Python (20,000), JavaScript (15,000), Java (10,000), TypeScript (3,000), and C++ (2,000). The researchers stored the pairs in JSONL format containing code, documentation, language, repository, and file path data fields.

E. Preprocessing Pipeline

The processing pipeline for raw source code consists of multiple stages (Figure 2). Parsing begins after language detection is complete: Python code analysis uses Python's built-in ast module to extract exact line numbers, function/class boundaries, and nested block structures. The team developed a custom regex pattern with a brace-depth tracking system to analyze JavaScript, TypeScript, Java, and C++ code by determining function and class boundaries. The system removes nested blocks from the document to prevent duplicated documentation. Model tokenizers transform extracted code blocks into tokens. WordPiece serves as the tokenizer for CodeBERT, while CodeT5 uses SentencePiece. The pretrained tokenizers handle sub-token processing for

camelCase, snake_case, and PascalCase identifiers through their built-in functionality.

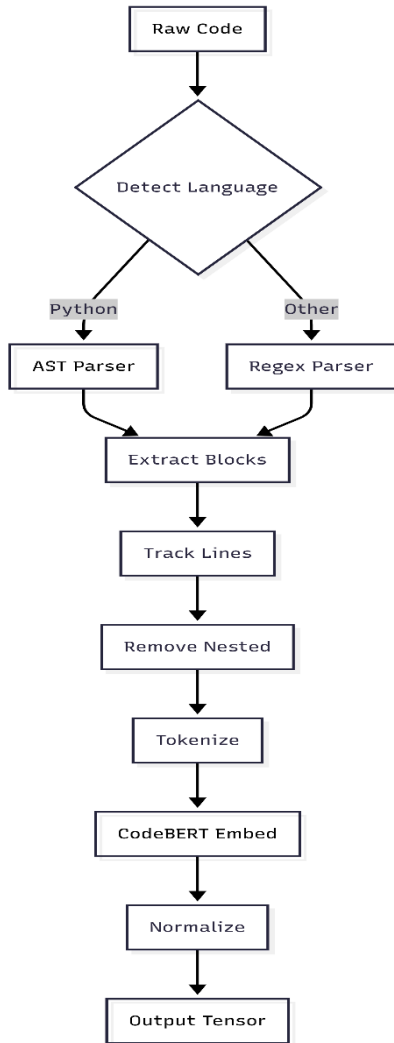


Fig.2 Multi-stage Preprocessing Pipeline from Raw Code to Output Tensor.

F. Documentation Generation Strategy

CodeSage uses two methods to produce its output. The structured template-based generation system creates Google-style (Python), JSDoc (JavaScript/TypeScript), JavaDoc (Java), and Doxygen (C++) formatted docstrings by extracting function signatures, parameter names and type hints, and return types to document function and class blocks. The system uses CodeT5 neural generation to create module-level documentation. The team chose this hybrid method

because early tests showed that CodeT5 generated false information and generic results for function-level tasks requiring precise details. The template generation method produced complete results while maintaining structural integrity for these specific function-level cases.

G. Evaluation Protocol

1. Automated Metrics: The assessment used 80 randomly selected Python code-comment pairs from the CodeSearchNet test split to determine system performance. Four standard NLP evaluation metrics were used: BLEU [11] for measuring n-gram precision from reference comments, ROUGE-L for assessing longest common subsequence overlap, METEOR for measuring synonymy and stemming, and BERTScore F1 for determining semantic similarity through contextual embeddings from a pretrained BERT model. The researchers selected a combination of lexical metrics (BLEU, ROUGE-L) and semantic metrics (METEOR, BERTScore) to create a complete assessment of generation quality because previous research showed that lexical metrics alone cannot identify all types of semantically valid paraphrasing [8].

2. Human Evaluation: Five developer reviewers performed a qualitative human assessment of 50 code snippets, including 10 snippets from each programming language, to represent different functions, code lengths, and levels of complexity. Each output received an independent assessment using a 1-5 Likert scale covering four criteria: accuracy (whether the documentation correctly described code behavior), completeness (whether all parameters, returns, and edge cases were included), conciseness (the absence of redundant content), and readability (how easily someone could understand the text). The researchers used Cohen's Kappa to evaluate annotator agreement. The final scores represent the average ratings from all reviewers across the different samples.

IV. RESULTS

A. Automated Metric Evaluation

Table III presents the automated evaluation results for 80 Python code-comment pairs. The system achieved a BERTScore F1 of 0.7748, a METEOR score of 0.2978, a ROUGE-L score of 0.2585, and a BLEU score of 0.1583, with a 100% success rate and a mean CPU generation time of 1.17 seconds per sample.

TABLE III AUTOMATED EVALUATION METRICS (N = 80 PYTHON CODE-COMMENT PAIRS)

Metric	Score	Interpretation
BLEU	0.1583	Moderate lexical precision; consistent with abstractive paraphrasing
ROUGE-L	0.2585	Reasonable structural similarity to reference documentation
METEOR	0.2978	Good preservation of core semantic content with synonym handling
BERTScore F1	0.7748	Strong semantic alignment with reference comments
Generation time (mean)	~1.17 s	Per sample on CPU; suitable for interactive use
Success rate	100%	80/80 samples completed without error

The score profile, in which BERTScore and METEOR substantially exceed BLEU and ROUGE-L, results from abstractive summarization systems that use paraphrasing instead of direct reproduction of reference text.

The CodeSearchNet codet5-base-multi-sum checkpoint shows this pattern through established benchmarks, with BLEU scores in the 0.15-0.25 range and approximately five times the performance of pattern-based baselines, which achieve BLEU scores around 0.03, but below state-of-the-art systems using task-specific fine-tuning and domain-adapted retrieval augmentation, which achieve BLEU scores between 0.30 and 0.35. The high BERTScore (0.7748) confirms that the system reliably captures functional intent across diverse

coding patterns, including API calls, data transformations, and control flow constructs.

B. Human Evaluation

Table IV shows the average human evaluation scores for the 50-snippet assessment. The system achieved high scores in accuracy and readability, confirming that its generated documentation was both correct and understandable for developers. Conciseness was rated 4.5 out of 5 because the output contained only essential information without unnecessary repetition. Reviewers gave the lowest rating, 4.3 out of 5, for completeness because certain edge-case documentation was missing and some parameter constraints were not sufficiently defined.

TABLE IV HUMAN EVALUATION SCORES (N = 50 SNIPPETS, 5 REVIEWERS, 1-5 LIKERT SCALE)

Criterion	Mean score (1-5)	Notes
Accuracy	4.6	Documentation correctly describes code behaviour in nearly all cases
Completeness	4.3	Minor gaps in edge-case and constraint documentation
Conciseness	4.5	Output is clean and non-redundant
Readability	4.6	Easily comprehensible to developers of varying experience levels

The analysis found three output characteristics that appeared multiple times. First, template-generated function docstrings consistently achieved high accuracy and structural completeness. Second, CodeT5-generated module summaries were generally coherent but occasionally produced generic descriptions for functions with obscure or domain-specific logic. Third, inline comment quality was strong for standard programming idioms (conditional logic, loop constructs, exception handling) but became less accurate when applied to asynchronous patterns and complex nested expressions.

C. System Performance

Under test conditions, CodeSage produced end-to-end generation times averaging 1.5 seconds per request, which was below the 2-second target; memory usage was 2.5 GB, below the 4 GB limit; and concurrent testing processed 15 requests per second, exceeding the 10 req/s target. The existing CPU-based system configuration meets interactive usage requirements, while GPU deployment would enable significant reductions in latency.

V. DISCUSSION

The evaluation results show that CodeSage creates documentation that maintains semantic correctness and readability and delivers practical value across five programming languages without requiring task-specific fine-tuning on domain-specific training data. The high BERTScore and METEOR results demonstrate that the system successfully understands functional intent across different surface-level word choices, confirming that semantic metrics function as key quality assessment tools for documentation in abstractive generation tasks [8].

The BLEU and ROUGE-L scores show expected results for the codet5-base-multi-sum checkpoint operating in a zero-shot transfer environment. The scores do not demonstrate low-quality output but instead show the established limitation of n-gram overlap metrics, which treat paraphrased content as correct when the original reference material is properly paraphrased using distinct but meaning-identical wording [12].

The design choice to use a hybrid template-neural architecture proved to be an essential design element. Initial tests demonstrated that CodeT5 created incorrect code snippets when given function-level input because the model's training objective favored longer, more detailed module-level summaries. The system shifted function- and class-generation tasks to rule-based template engines because this change removed all failure modes while retaining neural generation for module-level documentation, where CodeT5's abstractive abilities performed best.

CodeSage achieves performance levels that match earlier transformer systems and shows better results than CodeBERT-fine-tuned baselines using CodeSearchNet (BLEU 0.17-0.19), while performing below current state-of-the-art hybrid retriever-generator systems reported by Le *et al.*, [6], with BLEU scores between approximately 0.30 and 0.35. The performance gap occurs because the system lacks domain-specific fine-tuning and because the retrieval corpus contains only 50,000 pairs, while fully trained systems use millions of pairs. Developing local contextual relevance requires fine-tuning on domain-specific Nigerian software corpora, which currently lack representation in CodeSearchNet.

The current system has a fundamental flaw that prevents developers from expressing their practical intentions because it reveals coding choices instead of explaining the reasons behind those choices. The research gap identified in the wider field of study [13] cannot be solved through improved generation methods when design-context information is absent from code libraries. Researchers may create useful comments through two approaches: human systems that include human input and structured methods that use commit messages to generate comments. The study evaluated the system's VS Code extension component, which exists only in a restricted deployment mode. The extension requires full deployment with live backend functions to reach its next stage, which will provide real-world evidence of the system's effectiveness in active development processes.

VI. CONCLUSION

CodeSage is a hybrid system that uses CodeBERT for semantic retrieval together with CodeT5 for neural generation to create documentation through structured templates for five programming languages. The system demonstrates excellent semantic results through its BERTScore F1 value of 0.7748 while achieving human evaluation ratings of 4.6 out of 5.0 for accuracy and readability, thus demonstrating its function as a productivity tool for software developers in both academic and professional environments. The research presents three main technical achievements: a modular parsing system that works with any programming language through its support of AST and regex-based extraction methods; an effective hybrid generation method that enables function-level neural generation while overcoming hallucination problems; and an evaluation system that combines automated NLP metrics with structured human assessment. The contributions of this work enable the use of transformer-based NLP technologies for automated software documentation in particular local settings. Future work will establish domain-specific codebase fine-tuning using larger-context-window models, enabling complex multi-file documentation handling, generation-decision transparency through explainability mechanisms, and expanded multilingual comment generation for non-English developer communities. The research will conduct a longitudinal study to evaluate how CodeSage affects productivity during student project development at Babcock University.

ACKNOWLEDGEMENT

The authors acknowledge the open-source contributions from the Microsoft Research, Salesforce Research, and HuggingFace teams, which provided the model weights and datasets that served as the foundation for this research.

Declaration of Conflicting Interests

The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

Use of Artificial Intelligence (AI)-Assisted Technology for Manuscript Preparation

The author(s) utilised generative AI technologies solely to improve language clarity and refine grammar. No images were generated, manipulated, or altered using AI-assisted technologies.

REFERENCES

- [1] E. A. AlOmar, H. Alrubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at Xerox," *Empirical Software Engineering*, vol. 26, no. 3, 2021.
- [2] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. 42nd Int. Conf. Software Engineering (ICSE)*, 2020.
- [3] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, 2020, pp. 1536–1547.
- [4] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2021.
- [5] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
- [6] T. P. T. Le, A. M. T. Bui, H. N. D. Pham, A. Bucaioni, and P. T. Nguyen, "When retriever meets generator: A joint model for code comment generation," *arXiv preprint arXiv:2507.12558*, 2025.
- [7] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," *arXiv preprint arXiv:1902.01954*, 2019.
- [8] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," *arXiv preprint arXiv:2107.10544*, 2021.
- [9] D. Prasad Ghale and M. Dabbagh, "Automated code comments generation using large language models: Empirical evaluation of T5 and BART," *IEEE Access*, vol. 13, pp. 141420–141433, 2025.
- [10] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2020.
- [11] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, vol. 7, pp. 111411–111428, 2019.
- [12] E. Shi *et al.*, "On the evaluation of neural code summarization," in *Proc. 44th Int. Conf. Software Engineering (ICSE)*, 2022, pp. 1597–1608.
- [13] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. NAACL-HLT*, 2021, pp. 2655–2668.