Equivalent: Jurnal Ilmiah Sosial Teknik
Volume 8, Issue 5, 2226-2236
e_ISSN: 2775-0833
https://jurnalequivalent.id/index.php/jequi/index
DOI: doi.org/10.59261/jequi. V8i5.472

Prompt Augmentation with Context at Repository Scale for Code Summarization with the Cache-Augmented Generation Paradigm

Erwin Hartanto*
Computer Science Department, BINUS Graduate Program, Bina Nusantara University, Jakarta, Indonesia 11480

Maria Seraphina Astriani
Computer Science Department, School of Computing and Creative Arts, Bina Nusantara University, Jakarta, Indonesia 11480

***Corresponding author:**
Erwin Hartanto, Computer Science Department, BINUS Graduate Program, Bina Nusantara University, Jakarta, Indonesia 11480.
✉erwin.hartanto@binus.ac.id

Article Info:
Article history:
Received: September 16, 2026
Revised: October 12, 2026
Accepted: October 14, 2026
Available Online: November 11, 2026

Keywords:
Cache-Augmented Generation;
Code Summarization;
Large Language Models;
Prompt Augmentation;
Retrieval-Augmented Generation

This is an open access article under the CC BY-SA license.

Copyright © 2026 by Author. Published by Cerdika International Cyber Polytechnic

Abstract

Background: Code summarization is a fundamental aspect of software development and maintenance. However, the performance of large language models (LLMs) on this task can still be improved through prompt augmentation techniques, including Cache-Augmented Generation (CAG), which remains largely unexplored in the context of repository-level code.
Objective: This study systematically and empirically evaluates the effectiveness of CAG for code summarization when repository-level code structure is provided as context. Its performance is compared with that of Retrieval-Augmented Generation (RAG) and a base model.
Methods: This quantitative experimental study used the Llama 3.2 3B Instruct model on a sample of the CodeSearchNet dataset grouped by Python project repository. The three methods were evaluated and compared using three metrics: Summary-level Semantic Similarity (SIDE), Time to First Token (TTFT), and BERTScore.
Results: CAG achieved the lowest reported TTFT and a marginally higher BERTScore F1 score, whereas RAG achieved the highest SIDE score. Specifically, RAG attained a mean SIDE score of 0.596 and a mean BERTScore F1 score of 0.801. CAG achieved a mean TTFT of 0.188 seconds, considerably faster than RAG (2.680 seconds) and the base model (1.377 seconds).
Conclusion: Preloading comprehensive code context into the LLM's context window is effective for producing semantically aligned code summaries with minimal latency. However, CAG has a notable limitation: its cache must be updated when new information needs to be incorporated.

To cite this article: Erwin Hartanto, Maria Seraphina Astriani. (2026). Prompt Augmentation with Context at Repository Scale for Code Summarization with the Cache-Augmented Generation Paradigm. *Equivalent: Journal of Social Scientific Equivalent: Jurnal Ilmiah Sosial Teknik*, 8 (5), 2226-2236. https://doi.org/10.59261/jequi.v8i5.472

INTRODUCTION

The task of summarizing source code by producing concise, clear natural-language descriptions is one of the cornerstones of software development and maintenance (Su et al., 2024; Lester et al., 2021). Effective code summaries can significantly reduce the cognitive burden on programmers, facilitate code comprehension, help new team members become familiar with a codebase, and support the debugging process (Lomshakov et al., 2024; Tingting Xu et al., 2023).

Large language models (LLMs), trained on large datasets of natural-language text and source code, can generate code summaries with a high degree of fluency (Lyu et al., 2025; Wang

et al., 2025; Zhang et al., 2026). However, their performance can potentially be improved by providing additional relevant information or guiding the summarization process. This is the purpose of prompt augmentation techniques, which modify or enrich the prompts provided to LLMs to produce more accurate, relevant, and well-supported outputs. Given the complexity of source code, including intricate logic, language-specific syntax, and contextual dependencies, effective prompt augmentation is essential for code-related tasks. Techniques such as Retrieval-Augmented Generation (RAG), Chain-of-Thought (CoT) prompting, and in-context learning have demonstrated promising results in various natural language processing (NLP) tasks. More recently, Cache-Augmented Generation (CAG) has emerged as an alternative approach that preloads relevant knowledge into the model's context and prepares a reusable key–value cache, potentially reducing inference latency by minimizing the need for repeated retrieval and prompt processing (Gupta & Ranjan, 2024; Lu et al., 2021).

Various research efforts and approaches have explored this topic, particularly in the field of code summarization. Luo et al. (2024) research introduced an open-source framework that scans a single repository using Python and generates textual documentation based on prompts enriched with contextual information obtained through the Jedi library, a Python development and static-analysis tool. Similarly, Dhulshette et al. (2025) proposed an approach that parses a repository into an Abstract Syntax Tree (AST), extracts information about individual code units—including variables, functions, constructors, and enumerations—and generates summaries for these units. These summaries are subsequently combined to produce function-level summaries, which are then aggregated and summarized at higher levels to obtain package-level summaries (Dhulshette et al., 2025; Parvez et al., 2021). The latter study also employs a hierarchical tree derived from the AST to extract function-related information and descriptions, using the resulting information to enrich the prompt with contextual details.

Enriching LLM prompts with semantic and syntactic information, in conjunction with few-shot prompting, can improve model performance in code summarization. This approach is motivated by the observation that software developers often interpret code within the broader context of an entire repository rather than relying solely on local context at the function, class, or file level (Ahmed et al., 2024; Huang et al., 2024; Makharev et al., 2025).

These findings motivate the present study, which evaluates the use of the Cache-Augmented Generation (CAG) paradigm with repository-level code context for code summarization. Specifically, the study investigates the advantages and limitations of CAG when its cache contains repository-wide metadata and contextual information. It examines whether this approach affects inference speed and the quality of the generated code summaries.

Although both RAG and CAG can provide repository-related information to an LLM, their operational mechanisms and computational costs differ. RAG retrieves relevant passages for each request, potentially introducing retrieval and prompt-construction overhead while allowing contextual information to be refreshed as the repository changes. In contrast, CAG preloads repository context into a reusable key–value cache, potentially reducing repeated processing but requiring cache regeneration when the underlying repository content changes. The research gap addressed in this study concerns the comparative performance of these approaches for repository-context-aware code summarization under a shared experimental setting. The study does not seek to establish their performance across all repositories, datasets, or model families.

This study investigates repository-scale context preloading for code summarization using Cache-Augmented Generation (CAG), comparing its performance with that of a baseline model and Retrieval-Augmented Generation (RAG). The comparison focuses on semantic alignment, reference-based summary quality, and time to first token to examine how each approach balances summary quality and inference latency. By evaluating these three context-delivery strategies under specified model configurations and sampling conditions, the study provides empirical evidence of their relative performance without presupposing the general superiority of any particular approach.

The primary objective of this study is to evaluate the effectiveness of Cache-Augmented Generation (CAG) for repository-level code summarization and compare its performance with Retrieval-Augmented Generation (RAG) and a baseline LLaMA-3.2-3B-Instruct model. Specifically, the study examines the performance of these approaches in terms of semantic

alignment between source code and generated summaries, reference-based summary quality, and inference latency, measured using SIDE, BERTScore, and time to first token (TTFT), respectively. Through this comparative evaluation, the study aims to identify the relative advantages and limitations of repository-scale context preloading in supporting efficient, contextually relevant code summarization.

The benefits of this research are both theoretical and practical. Theoretically, the study contributes empirical evidence to the growing body of literature on prompt augmentation and repository-aware code summarization by examining the trade-offs among contextual information delivery, summary quality, and response latency. Practically, the findings provide insights for software developers and researchers seeking to select appropriate context-augmentation strategies according to their application requirements, particularly when balancing inference efficiency against semantic alignment. The findings may suggest that CAG is suitable for repeated summarization tasks involving relatively stable repository information, whereas RAG may offer advantages when contextual information must be retrieved or updated frequently. These implications are limited to the evaluated model, dataset, and experimental configuration and provide a foundation for further research into the scalability, adaptability, and reliability of repository-level code summarization systems.

METHOD

This quantitative comparative experiment used a subset of the CodeSearchNet dataset (Husain et al., 2019). Eleven Python repositories comprising 1,035 code pieces were grouped by repository_name to support repository-context construction. This subset was selected purposively under computational constraints rather than through probability sampling of all software repositories. The actual inference experiment used 30 sampled code items, selected with a shuffling seed of 42. The manuscript did not document the number of items selected from each repository; therefore, the results should not be interpreted as representative repository-level estimates. The three experimental conditions were the baseline LLaMA-3.2-3B-Instruct model, RAG, and CAG.

Table 1
Sharing of Datasets and Samples Used

Repository Name	Number of Code Pieces
OpenAI/Baselines	86
encode/starlette	15
quantopian/pyfolio	135
Tensorforce/Tensorforce	184
yinkaisheng/Python-UIAutomation-for-Windows	182
bethgelab/foolbox	28
PiotrDabkowski/Js2Py	65
clab/dynet	9
ansible/tower-cli	114
Erotemic/ubelt	117
FenTechSolutions/CausalDiscoveryToolbox	100

Table 1 presents the 11 Python repositories included in the research subset and the number of code pieces obtained from each repository. The number of code pieces ranged from 9 in clab/dynet to 184 in Tensorforce/Tensorforce, yielding a total of 1,035 code pieces. These items were grouped according to their source repositories to support the construction of repository-level context.

The experiments were conducted in Google Colab using an A100 GPU, with the same

LLaMA-3.2-3B-Instruct model used across all three experimental conditions (Grattafiori, 2024). The comparison held the model family, sampled evaluation set, and outcome metrics constant while varying the context-delivery strategy: no additional repository context (baseline), retrieved context (RAG), or preloaded cached context (CAG). The exact prompt token budgets, generation settings, and number of repeated runs were not documented; consequently, complete control over all potential confounding variables could not be verified.

In the CAG condition, all code pieces from a single repository, together with the associated metadata fields repository_name, func_code_tokens, func_path_in_repository, and func_documentation_string, were combined and tokenized. The resulting context was then processed to generate a key–value cache, represented by the past_key_values property, with caching enabled through the use_cache=True configuration. The cache was stored locally in cache_knowledge.pt and generated once during the preloading stage before the subsequent inference session. This approach was intended to reduce the need to reprocess the preloaded context for each subsequent query.

RAG used FAISS for vector-based similarity search and the BAAI/bge-large-en-v1.5 encoder for embedding generation. The method reportedly examined retrieval sizes of 12, 25, and 50 documents; however, the available methodological record did not identify which setting produced the final values reported in Tables 3 and 5. Consequently, these tables are treated as reported experimental outputs without attributing them to a specific retrieval count. A fully reproducible comparison would require identifying the retrieval setting used for the reported results and providing configuration-specific results.

All three methods were evaluated on 30 shuffled code samples using a seed of 42, and the generated summaries and inference timings were recorded. Although Table 1 lists 1,035 available code pieces across 11 repositories, the distribution of the 30 evaluation samples among the individual repositories was not reported. Therefore, the resulting metrics describe only the reported experimental sample. No statistical significance tests or uncertainty intervals were reported.

Performance was assessed using three complementary but non-interchangeable metrics: time to first token (TTFT), which measures response latency; Summary Alignment to Code Semantic (SIDE), which estimates semantic alignment between source code and generated summaries; and BERTScore, which compares generated summaries with human-written reference summaries. Together, these metrics provide a multidimensional assessment of latency and summary quality. However, none independently establishes factual correctness, executable behavior, or statistical significance. Tables 2–4 label two SIDE columns as “SIDE True” and “SIDE Pred.” However, the computational definitions of these columns were not specified in the available methodological record. Therefore, the labels could not be reliably interpreted as measures of ground-truth and predicted alignment without further verification.

1. Time to First Token (TTFT) measures the elapsed time, in seconds, between submitting a prompt and generating the first output token. Lower TTFT values indicate lower initial response latency.
2. Summary Alignment to Code Semantic (SIDE) measures semantic alignment between a source-code fragment and its generated summary by calculating the cosine similarity between their embeddings. The metric uses a fine-tuned MPNet encoder and a hard-negative training technique (Mastropaolo et al., 2023). It does not require a reference summary and focuses on semantic correspondence between the source code and the generated summary rather than textual similarity alone.

BERTScore measures semantic similarity between generated summaries and human-written reference summaries using contextual embeddings from pretrained BERT-based models. It is typically reported in terms of precision, recall, and F1 scores. Higher BERTScore values generally indicate greater semantic similarity to the reference summaries.

RESULTS AND DISCUSSION

Results

Experimental Execution

The experiment was conducted on the Google Colab platform using an A100 GPU. The

Llama 3.2 3B Instruct model was accessed through the Hugging Face Transformers library, with device mapping configured to use CUDA to accelerate inference. The dataset was retrieved using the Hugging Face Datasets library and filtered in Python to select the test samples. The resulting data were then grouped by their original repositories to preserve repository-level context.

The prompt-design stage was conducted first to determine the prompt configuration that yielded the best performance. This stage included the use of Llama-specific special tokens, such as <|start_header_id|> and <|eot_id|>, to ensure that the prompt structure was correctly interpreted by the model. For the CAG method, each data point was augmented with {prompt_metadata}, containing metadata about the file location, function name, and abstract syntax tree (AST) tokens. For RAG, the number of retrieved documents was varied across three configurations: 12, 25, and 50 documents. Document retrieval was performed using similarity search over a FAISS vector index. Each sample was processed through inference, with the context window reset to its initial length for each new query. The generated summaries and per-sample time-to-first-token (TTFT) measurements were stored for the calculation of mean evaluation metrics, including SIDE, TTFT, and BERTScore, using the SIDE evaluation configuration described by (Mastropaolo et al., 2023).

Experimental Results

The evaluation was conducted on eleven selected repositories, which were grouped as shown in Table 1. The detailed results for the baseline model, RAG, and CAG are presented in Tables 2, 3, and 4, respectively, while the aggregate results for all three methods are presented in Table 5.

Table 2
Results of Basic Model Testing Metrics

Repository	Total Code	SIDE True	SIDE Pred	TTFT(s)	BERT Prec.	BERT F1
ansible/tower-cli	114	0.502	0.441	1.415	0.833	0.757
bethgelab/foolbox	28	0.743	0.798	1.302	0.827	0.743
clab/dynet	9	0.338	0.376	1.109	0.836	0.775
FenTechSolutions/CausalDiscoveryToolbox	100	0.439	0.600	1.305	0.839	0.767
encode/starlette	15	0.527	0.519	1.060	0.848	0.821
Erotemic/ubelt	117	0.546	0.437	1.697	0.837	0.786
OpenAI/Baselines	86	0.606	0.738	1.531	0.819	0.789
PiotrDabkowski/Js2Py	65	0.557	0.586	1.359	0.821	0.810
quantopian/pyfolio	135	0.738	0.725	1.944	0.831	0.766
Tensorforce/Tensorforce	184	0.407	0.619	1.293	0.835	0.774
yinkaisheng/Python-UIAutomation-for-Windows	182	0.799	0.760	1.137	0.837	0.781
RED		0.564	0.600	1.378	0.833	0.779

Table 2 presents the baseline model's evaluation results without prompt augmentation across eleven selected repositories. The reported SIDE values range from 0.338 for clab/dynet to 0.799 for yinkaisheng/Python-UIAutomation-for-Windows. The reported mean TTFT is 1.378 seconds, and the mean BERTScore F1 is 0.779. These results provide a baseline for comparison with the RAG and CAG methods.

Table 3
RAG Method Testing Metric Results

Repository	Total Code	SIDE True	SIDE Pred	TTFT(s)	BERT Prec.	BERT F1
ansible/tower-cli	114	0.507	0.468	2.719	0.843	0.788
bethgelab/foolbox	28	0.752	0.811	2.615	0.849	0.762
clab/dynet	9	0.338	0.536	2.924	0.844	0.808
FenTechSolutions/CausalDiscoveryToolbox	100	0.448	0.627	2.851	0.837	0.781
encode/starlette	15	0.527	0.422	2.497	0.845	0.869
Erotemic/ubelt	117	0.558	0.407	2.752	0.846	0.764
OpenAI/Baselines	86	0.605	0.766	2.850	0.834	0.813
PiotrDabkowski/Js2Py	65	0.541	0.645	2.642	0.821	0.825
quantopian/pyfolio	135	0.738	0.742	2.872	0.851	0.763
Tensorforce/Tensorforce	184	0.404	0.632	2.502	0.840	0.801
yinkaisheng/Python-UIAutomation-for-Windows	182	0.800	0.823	2.255	0.845	0.814
RED		0.565	0.625	2.680	0.841	0.799

Table 3 reports a mean TTFT of 2.680 seconds, a reported aggregate SIDE value of 0.625, and a mean BERTScore F1 of 0.799 for the RAG method. However, although the experimental methodology specifies retrieval configurations of 12, 25, and 50 documents, the reported results do not identify which configuration produced these outcomes. This omission limits the reproducibility of the experiment.

Table 4
CAG Method Testing Metric Results

Repository	Total Code	SIDE True	SIDE Pred	TTFT(s)	BERT Prec.	BERT F1
ansible/tower-cli	114	0.502	0.457	0.251	0.858	0.782
bethgelab/foolbox	28	0.743	0.800	0.062	0.852	0.760
clab/dynet	9	0.338	0.520	0.047	0.865	0.810
FenTechSolutions/CausalDiscoveryToolbox	100	0.546	0.407	0.268	0.836	0.784
encode/starlette	15	0.527	0.414	0.043	0.864	0.865
Erotemic/ubelt	117	0.546	0.407	0.268	0.855	0.770
OpenAI/Baselines	86	0.606	0.739	0.149	0.845	0.821
PiotrDabkowski/Js2Py	65	0.557	0.633	0.116	0.840	0.822
quantopian/pyfolio	135	0.738	0.737	0.339	0.852	0.785
Tensorforce/Tensorforce	184	0.407	0.629	0.302	0.842	0.809
yinkaisheng/Python-UIAutomation-for-Windows	182	0.799	0.821	0.224	0.854	0.806
RED		0.573	0.597	0.188	0.851	0.801

Table 4 shows that CAG achieved a reported mean TTFT of 0.188 seconds and a mean

BERTScore F1 of 0.801. However, its mean SIDE value in Table 5 is 0.596, which is lower than that of RAG (0.625) and slightly lower than that of the baseline model (0.599). Therefore, the results support a latency advantage and a marginal advantage in BERTScore F1, rather than superior performance across all quality metrics.

Table 5

Aggregate Results of the Third Test Method

Method	Mean SIDE	Mean TTFT(s)	BERT Prec.	BERT F1	BERT Recall
Basic Model	0.599	1.377	0.832	0.778	0.803
Retrieval-Augmented Generation	0.625	2.680	0.841	0.798	0.819
Cache-Augmented Generation	0.596	0.188	0.851	0.801	0.823

Table 5 shows that RAG achieved the highest mean SIDE value (0.625), whereas CAG achieved the lowest mean TTFT (0.188 seconds) and the highest BERTScore F1 (0.801), marginally exceeding RAG (0.798). The baseline model recorded a mean SIDE value of 0.599 and a mean TTFT of 1.377 seconds. Because no inferential statistical tests were reported, these differences should be interpreted as descriptive rather than statistically significant.

Performance Evaluation

Tables 2–4 present the detailed evaluation results for each method, while Table 5 summarizes the aggregate results. The mean values for the three evaluation dimensions are illustrated in Figures 1–3.



Figure 1. Mean distribution of the third SIDE method

Figure 1 compares the mean SIDE values of the baseline model (0.599), RAG (0.625), and CAG (0.596) in a bar chart. RAG achieved the highest mean SIDE value, although the differences among the three methods were relatively small.

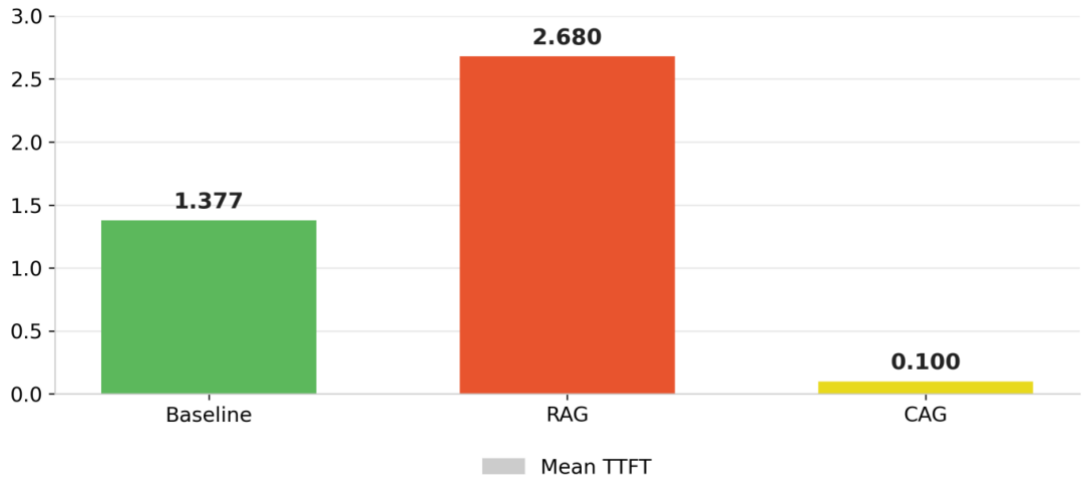


Figure 2. Mean distribution of TTFT of the three methods

Figure 2 compares the mean TTFT values of RAG (2.680 seconds), the baseline model (1.377 seconds), and CAG (0.188 seconds). This descriptive difference is consistent with the potential latency benefits of reusing cached context in CAG. However, the reported measurements do not establish the separate contributions of preprocessing, cache construction, and retrieval costs under different deployment conditions.

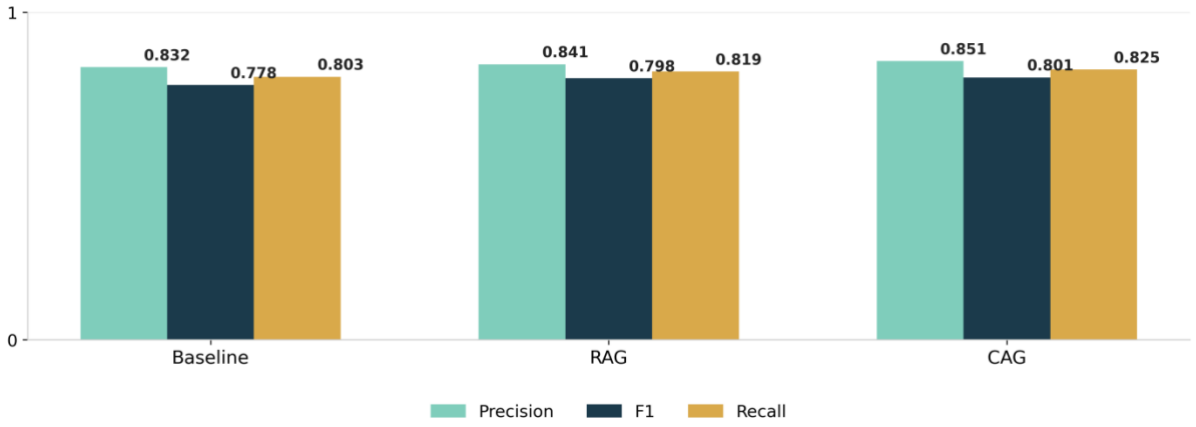


Figure 3. Precision, F1, and Recall Distributions of the three BERTScore methods

Figure 3 compares the three BERTScore metrics—precision, F1, and recall—for the baseline model, RAG, and CAG. All three methods exhibit a similar pattern, with precision values higher than recall and F1 values. CAG achieved the highest reported precision (0.851) among the three methods.

SIDE Results Analysis

Table 5 and Figure 1 show that RAG achieved the highest mean SIDE value (0.625), compared with the baseline model (0.599) and CAG (0.596). SIDE evaluates semantic correspondence through embedding-based representations rather than directly verifying every aspect of code behavior. Its interpretation therefore benefits from complementary reference-based metrics, such as BERTScore, and, in future evaluations, direct human assessment of the factual and functional faithfulness of generated summaries.

TTFT Results Analysis

Table 5 and Figure 2 indicate that CAG achieved the lowest reported mean TTFT (0.188 seconds), compared with the baseline model (1.377 seconds) and RAG (2.680 seconds). These figures indicate a substantial descriptive difference in latency within the experimental setup. However, they do not establish an independently verified effect that generalizes beyond the tested

configuration.

BERTScore Results Analysis

Based on Table 5, CAG achieved the highest BERTScore F1 (0.801), followed by RAG (0.798) and the baseline model (0.778), although the differences were small. This result suggests that CAG's generated summaries achieved slightly greater similarity to the human-written reference summaries under the BERTScore F1 metric. Nevertheless, this marginal advantage does not establish that CAG produces more faithful or functionally accurate summaries in every case. The results suggest that CAG's latency advantage coincided with a small improvement in BERTScore F1, although further evaluation is needed to establish the robustness of this finding.

Discussion

The experiments and subsequent evaluations revealed a trade-off between code-summary quality and inference speed among prompt augmentation methods that incorporate broader and more detailed code context (Agrawal & Kumar, 2025; Yang et al., 2025; Zhu et al., 2024).

CAG was distinguished most clearly by its lower reported latency, although its mean SIDE value did not exceed those of either comparator. Its BERTScore F1 advantage over RAG was small (0.801 versus 0.798). These differences highlight a trade-off involving cache reuse, context freshness, and distinct dimensions of summary quality rather than demonstrating uniform semantic superiority.

Preloading repository context can reduce the need to repeatedly process the same contextual information across a sequence of queries. However, changes to source code or documentation may invalidate cached key-value representations, necessitating explicit cache refreshing or reconstruction. Future implementations could investigate change detection, repository-version tagging, selective cache invalidation, and the measurement of cache-refresh overhead. These are proposed research directions rather than techniques evaluated in this experiment.

RAG achieved the highest reported mean SIDE value, suggesting an advantage in code-summary semantic correspondence under the evaluation metric used in this study. However, it also incurred higher TTFT in the tested configuration. CAG produced faster responses and a marginally higher BERTScore F1. In the absence of repeated trials and statistical significance testing, neither method can be declared statistically superior overall.

The study has several limitations, including the use of a single model and hardware configuration, a stated 30-item evaluation subset, unspecified sample counts per repository and the final RAG retrieval configuration, missing definitions for the “SIDE True” and “SIDE Pred” columns, and the absence of statistical significance testing. Consequently, generalization to entire repositories and the attribution of observed improvements solely to caching have not been established.

Based on the reported results, the study addresses the following research questions.

1) How does Cache-Augmented Generation (CAG) perform in code summarization when the LLM is provided with contextual information representing the overall code structure? The experimental results suggest that CAG is promising for code summarization, particularly in terms of inference efficiency. It achieved the lowest reported mean TTFT (0.188 seconds) and the highest BERTScore F1 (0.801). However, its mean SIDE value was lower than those of both the baseline model and RAG. Accordingly, its performance should be characterized as favorable in terms of latency and marginally favorable in terms of BERTScore F1, rather than uniformly superior in summary quality.
2) How does CAG compare with other techniques, such as Retrieval-Augmented Generation (RAG), in terms of summary quality and efficiency? Based on Tables 2–5 and Figures 1–3, CAG achieved the lowest observed mean TTFT and the highest reported BERTScore F1, whereas RAG achieved the highest reported mean SIDE value. The evidence therefore supports a conditional trade-off between latency and summary quality rather than the categorical conclusion that CAG outperforms RAG across all evaluation metrics.

CONCLUSION

This study evaluated the effectiveness of Cache-Augmented Generation (CAG) for repository-level code summarization by comparing its performance with Retrieval-Augmented Generation (RAG) and a baseline Llama 3.2 3B Instruct model. The experimental findings showed that CAG achieved the lowest mean time-to-first-token (TTFT) of 0.188 seconds, compared with 1.377 seconds for the baseline model and 2.680 seconds for RAG. CAG also recorded the highest BERTScore F1 (0.801), slightly exceeding RAG (0.798) and the baseline model (0.778). However, RAG achieved the highest mean SIDE value (0.625), compared with 0.599 for the baseline model and 0.596 for CAG. These findings indicate that CAG offers a substantial descriptive advantage in inference latency and a marginal improvement in reference-based summary similarity, whereas RAG demonstrates stronger semantic correspondence between code and summaries under the evaluation metric employed. Therefore, the study concludes that preloading repository-scale context through CAG is a promising strategy for improving summarization efficiency, although it does not consistently outperform alternative approaches across all dimensions of summary quality.

The findings contribute to an understanding of the trade-offs among inference efficiency, semantic alignment, and contextual information management in LLM-based code summarization. From a practical perspective, CAG may be beneficial for repeated summarization tasks involving relatively stable repository contexts, whereas RAG may be more suitable when frequently updated contextual information must be retrieved. Nevertheless, the conclusions are limited using a single language model, a stated evaluation subset of 30 code samples drawn from selected Python repositories in CodeSearchNet, incomplete reporting of repository-specific sampling and the final RAG retrieval configuration, and the absence of repeated trials and statistical significance testing. Consequently, the observed differences should be interpreted descriptively rather than as evidence of general superiority. Future research should incorporate larger and more diverse datasets, additional language models and programming languages, standardized experimental configurations, repeated evaluations with statistical testing, and direct assessments of code-summary faithfulness. Further investigation of cache construction, invalidation, and refresh costs is also necessary to determine the practical scalability and adaptability of CAG in continuously evolving software repositories.

ACKNOWLEDGEMENT

The authors would like to express their gratitude to the Universitas Bina Nusantara, for the academic support provided throughout the conduct of this research. The authors also extend their appreciation to all parties who provided valuable input, suggestions, and support during the research process and manuscript preparation. The various contributions and feedback received have helped improve the quality, accuracy, and scientific presentation of this article, thereby enhancing its suitability for publication.

AUTHOR CONTRIBUTION STATEMENT

Erwin Hartanto: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Resources, Data Curation, Writing – Original Draft, Visualization, Project administration. Maria Seraphina Astriani: Writing – Review & Editing, Supervision, Project administration.

DATA AVAILABILITY

Data supporting this study are publicly available from huggingface at https://huggingface.co/datasets/code-search-net/code_search_net.

REFERENCES

Agrawal, R., & Kumar, H. (2025). Enhancing Cache-Augmented Generation (CAG) with Adaptive Contextual Compression for Scalable Knowledge Integration. *ArXiv*.

Ahmed, T., Pai, K. S., Devanbu, P., & Barr, E. T. (2024). Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). *Proceedings - International Conference on Software Engineering*, 2720–2732.

https://doi.org/10.1145/3597503.3639183

Dhulshette, N., Shah, S., & Kulkarni, V. (2025). *Hierarchical Repository-Level Code Summarization for Business Applications Using Local LLMs*. http://arxiv.org/abs/2501.07857

Grattafiori, A. (2024). *The llama 3 herd of models."arXiv preprint arXiv:2407.21783*.

Gupta, S., Ranjan, R., & Singh, S. N. (2024). A comprehensive survey of retrieval-augmented generation (rag): Evolution, current landscape and future directions. *arXiv preprint arXiv:2410.12837*.

Huang, H.-H., Chan, B. J., Chen, C.-T., & Cheng, J.-H. (2024). *Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks*. https://doi.org/10.1145/3701716.3715490

Lester, B., Al-Rfou, R., & Constant, N. (2021). The Power of Scale for Parameter-Efficient Prompt Tuning. http://arxiv.org/abs/2104.08691

Lomshakov, V., Podivilov, A., Savin, S., Baryshnikov, O., Lisevych, A., & Nikolenko, S. (2024, November). ProConSuL: Project context for code summarization with LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track* (pp. 866-880).

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., … Liu, S. (2021). *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. http://arxiv.org/abs/2102.04664

Luo, Q., Ye, Y., Liang, S., Zhang, Z., Qin, Y., Lu, Y., Wu, Y., Cong, X., Lin, Y., Zhang, Y., Che, X., Liu, Z., & Sun, M. (2024). RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. *EMNLP 2024 - 2024 Conference on Empirical Methods in Natural Language Processing, Proceedings of System Demonstrations*. https://doi.org/10.18653/v1/2024.emnlp-demo.46

Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., & Thongtanunam, P. (2025). Automatic programming: Large language models and beyond. ACM Transactions on Software Engineering and Methodology, 34(5), 1-33.

Makharev, V., & Ivanov, V. (2025). *Code Summarization Beyond Function Level*. http://arxiv.org/abs/2502.16704

Mastropaolo, A., Ciniselli, M., Di Penta, M., & Bavota, G. (2023). *Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization*. http://arxiv.org/abs/2312.15475

Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). *Retrieval Augmented Code Generation and Summarization*. http://arxiv.org/abs/2108.11601

Su, C.-Y., Bansal, A., Huang, Y., Li, T. J.-J., & McMillan, C. (2024). *Context-aware Code Summary Generation*. http://arxiv.org/abs/2408.09006

Tingting Xu, Yun Miao, Chunrong Fang, Hanwei Qian, Xia Feng, Zhenpeng Chen, Chong Wang, Jian Zhang, Weisong Sun, Zhenyu Chen, & Yang Liu. (2023). *A Prompt Learning Framework for Source Code Summarization*. *2022-March*, 36–47. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Wang, Y., Xie, C., Zhang, W., & Qiu, X. (2025, July). FRCS-LLM: A Framework for Refining Code Summarization in Large Language Models via Pre-trained Models. In 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 1244-1253). IEEE.

Yang, Z., Chen, S., Gao, C., Li, Z., Hu, X., Liu, K., & Xia, X. (2025). An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities. *ACM Transactions on Software Engineering and Methodology*, *34*(7). https://doi.org/10.1145/3717061

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yu, S., Sun, W., ... & Chen, Z. (2026). A survey on large language models for software engineering. Science China Information Sciences, 69(4), 141102.

Zhu, Q., Luo, X., Liu, F., Gao, C., & Che, W. (2024). A Survey on Natural Language Processing for Programming. *2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC-COLING 2024 - Main Conference Proceedings*. https://doi.org/10.63317/2tdeieb24zmy