BRAD: Enhancing Code summarization with Bytecode Retrieval-Augmented Deliberation model

Anonymous ACL submission

Abstract

Comments and summaries play a critical role in program comprehension. However, high-quality comments are often missing in large codebases. Existing code summarization methods improve linguistic fluency but still fall short in capturing program execution semantics. Furthermore, source-level retrieval methods struggle to identify code examples that are syntactically different but functionally equivalent. To address these limitations, we propose BRAD (Bytecode Retrieval-Augmented Deliberation), a novel retrieval-augmented deliberation model for code summarization that integrates bytecode-level semantics with multi-pass deliberation. Specifically, BRAD (1) encodes bytecode control-flow graphs (CFG) with a Graph Attention Network (GAT) to preserve control flow semantics, and (2) retrieves exemplar drafts in the bytecode text space to find functionally similar references. These are then fused with textual encodings in a deliberative refinement process to generate the final summary. We evaluate BRAD on a public Java dataset, and the results showed that compared with the strong baselines, BRAD has increased by 18.7% in BLEU-1, 16.2% in BLEU-2, 8.5% in BLEU-3, 2.8% in BLEU-4, 13.4% in ROUGE-L, and 10.0% in CIDEr.

1 Introduction

Code comments play a critical role in facilitating software development (Nielebock et al., 2019; Hu et al., 2022) and maintenance (Wen et al., 2019; Misra et al., 2020; Tan, 2015). Writing code comments has been recognized as a good practice in the software development community or industry projects (De Souza et al., 2005; Zhai et al., 2020), and high-quality code comments (e.g., method or function summaries) greatly reduce the cognitive cost of program comprehension (Steidl et al., 2013; Huang et al., 2020b). However, writing high-quality code comments is a tedious and time-consuming task (De Souza et al., 2005; Kajko-Mattsson, 2005). Therefore, various automatic source code summarization methods have been proposed to generate code comments (also referred to as summaries). Specifically, given a code snippet (a method or function) by the developer, code summarization aims to generate summaries describing the functionality of the code snippet.

Existing code summarization methods can be classified into *template-based* (Mosa et al., 2017; Sridhara et al., 2010), *retrieval-based* (Wong et al., 2015; Zhu et al., 2022; Parvez et al., 2021; Wei et al., 2020), *learning-based* (Alon et al., 2018; Hu et al., 2018; Li et al., 2020; Ahmad et al., 2020; Han et al., 2021; Li et al., 2022; Gao et al., 2023; Niu et al., 2022) and *pretrained language models* (Feng et al., 2020; Guo et al., 2022; Wang et al., 2022, 2021; Nam et al., 2024; Ahmed and Devanbu, 2022; Ahmed et al., 2024; Song et al., 2024). Despite achieving promising results, these methods often rely heavily on surface-level or syntactic representations such as token sequences or Abstract Syntax Trees (ASTs) (Nie et al., 2021; Chen et al., 2021). These representations capture syntactic patterns but neglect the execution semantics or control flow structures that are critical for describing the program intent (Zhang et al., 2022; Lin et al., 2021). Moreover, the retrieval-augmented generation (RAG) frameworks (Nishikawa et al., 2024; Lu and Liu, 2024; Yu et al., 2022) enhance the comment generation performance by incorporating similar <code, comment> exemplars retrieved from large corpora. However, most of these methods perform retrieval with source code, where similarity is measured by lexical or syntactic overlap rather than by functional similarity.

Recently, some studies (Huang et al., 2020a; Shi et al., 2023) demonstrate that bytecode (the intermediate representation produced by compilers) can offer valuable program execution and control-flow information for enhancing code comment genera-

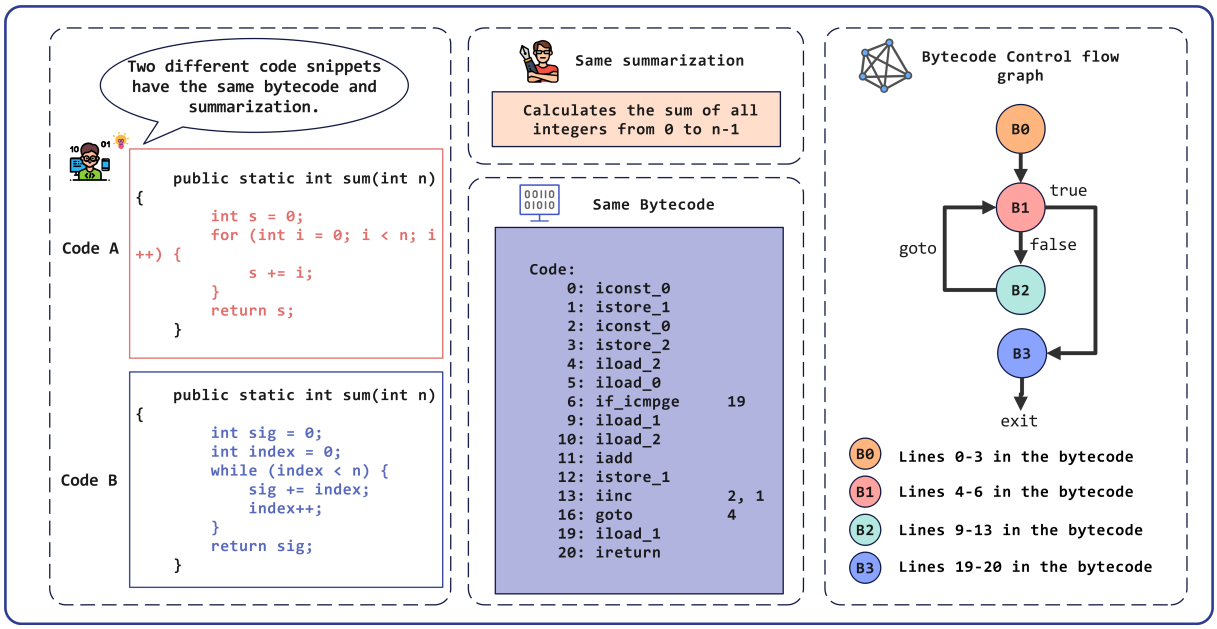

Figure 1: An example of different implementations which have the same bytecode. This example shows the possibility of identifying the semantic similarities through bytecode.

tion. Figure 1 indicates an example for the importance of bytecode information. Code A and Code B are two pieces of Java code snippets with the same functionality, which calculate the cumulative sum of all integers from 0 to $n-1$. Their implementations are different, where Code A uses a for loop, while Code B uses a while loop. However, their bytecodes and code comments are exactly the same, indicating the possibility of identifying similar functionalities through bytecode. This observation motivates us to explore how to combine bytecode-level retrieval information with control flow structures to enhance the efficiency of code summarization tasks.

To address the above issues, we propose BRAD, which is a novel Bytecode Retrieval-Augmented Deliberation model with control flow graph encoding for code summarization. BRAD first employs bytecode-level retrieval to obtain exemplar code comment draft that are more likely to be functionally similar to the query code. Second, BRAD encodes the control flow graph of source code with a Graph Attention Network to augment code structure information. Finally, the exemplar code comment draft from byte-level retrieval, control flow graph information, and query code are sent to a deliberation network (A multi-pass decoding model can optimize the generated results (Xia et al., 2017; Liao et al., 2025; Mu et al., 2022)) to iteratively generate the summary with rich functional semantics.

To the best of our knowledge, this work represents an early attempt to incorporate bytecode-level retrieval into a deliberation-based framework for code summarization. To validate the effectiveness of BRAD, we conducted extensive experiments on a public Java dataset (Huang et al., 2025). The results demonstrate that BRAD outperforms the baseline methods, achieving relative improvements of up to 18.7% in BLEU-1, 16.2% in BLEU-2, 8.5% in BLEU-3, 2.8% in BLEU-4, 13.4% in ROUGE-L, and 10.0% in CIDEr. Furthermore, we perform a qualitative assessment with LLMs (Dong et al., 2025; Li et al., 2024) across four human-oriented dimensions (Wu et al., 2025) (correctness, readability, informativeness, and usefulness) and human evaluation in three aspects (naturalness, informativeness, and usefulness). The results also demonstrate that BRAD can generate more informative and useful comments. We summarize the main contributions of this paper as follows:

- We propose BRAD, which encodes the bytecode CFG through graph attention networks to guide the model in capturing control flow information that reflects the execution semantics, thus improving the relevance and accuracy of code comments.
- We introduce a bytecode-level retrieval augmentation for code summarization that retrieves exemplar drafts in bytecode space

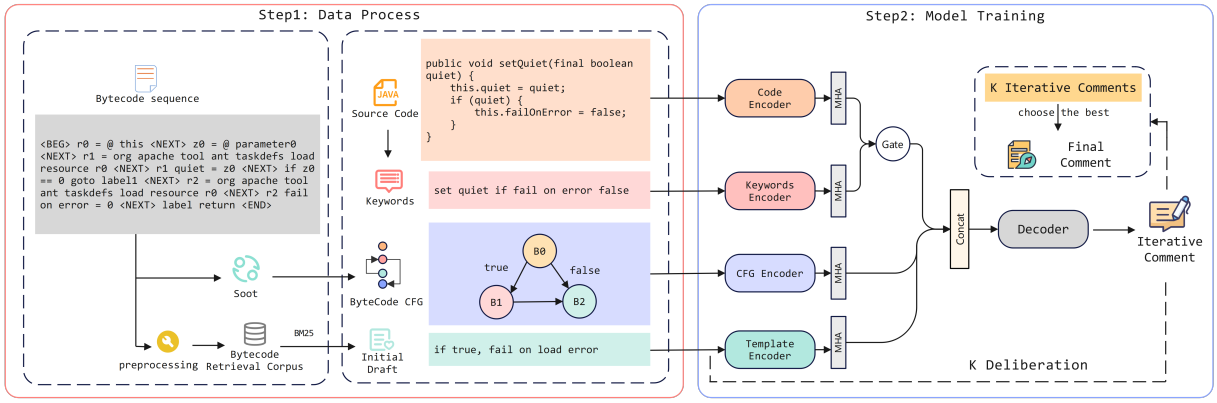

Figure 2: The overall framework of the BRAD.

rather than the raw source-text space. This method addresses the shortcomings of source-level lexical matching by retrieving functionally nearer exemplars to bootstrap the generator.

- We compare BRAD with strong baselines and observe consistent improvements across automatic metrics. We further validate the quality of generated comments through human evaluation in terms of naturalness, informativeness, and usefulness. The replication package is available at [1].

2 Related Work

Code comment generation Early neural approaches formulated summarization as a sequence-to-sequence learning problem, treating code as token sequences and applying encoder–decoder architectures adapted from machine translation (Hu et al., 2018; Li et al., 2020). Subsequent work demonstrated that explicitly modeling program structure can substantially improve semantic understanding. Structure-aware models incorporate abstract syntax trees, control-flow graphs, data-flow graphs, or other program representations using path-based encodings (Alon et al., 2018) or graph neural networks (Son et al., 2022; Yang et al., 2023; Bansal et al., 2023), showing consistent gains over sequence-only baselines by better capturing long-range dependencies and execution-related relations. Recent studies (Huang et al., 2023; Chen et al., 2025; Huang et al., 2025) have explored incorporating bytecode information as a complementary semantic signal. However, most existing approaches rely on CFG serialization, which destroys the topological information of the graph.

[1] https://anonymous.4open.science/r/BRAD-2CFB/

In parallel, retrieval-augmented generation has emerged as an effective strategy for code summarization. By retrieving similar code-comment pairs as exemplars or drafts, retrieval-based and hybrid frameworks (Parvez et al., 2021; Yu et al., 2022; Lu and Liu, 2024) reduce hallucination and improve fluency and informativeness, especially when combined with neural refinement or multi-pass deliberation mechanisms (Mu et al., 2022; Hou et al., 2023). However, current methods only perform retrieval at the source code level, ignoring semantically equivalent code with different implementations. This work positions BRAD in this gap by combining GAT-based encoding of bytecode CFGs with bytecode-level retrieval to better support semantic reasoning for source code summarization.

3 METHODOLOGY

3.1 Overview of BRAD Framework

The BRAD framework is a modular approach for code summarization that integrates bytecode-level structure, exemplar retrieval, and deliberative generation. As shown in Figure 2, we derive bytecode-level control-flow graphs (CFGs) to capture the execution-oriented structure. The CFG is encoded with a Graph Attention Network (GAT) (Velickovic et al., 2017) to obtain a structure-aware representation of bytecode execution. In parallel, bytecode-level retrieval provides an exemplar draft, which is combined with structural and source-code representations to guide generation. Through multi-pass deliberation, the model progressively refines the draft into a faithful natural language summary.

Bytecode-level retrieval and CFG encoding are complementary components that jointly enable the deliberation network to produce more faithful and detailed summaries. Bytecode retrieval supplies

external knowledge in the form of high-quality exemplar drafts that capture what the method does at a macroscopic level (**What**). These drafts provide useful priors about functionality but can be vague about implementation details. In contrast, the GAT encoded control flow graph supplies internal knowledge that describes how the method implements its behavior (**How**), such as the branching and looping for implementing functions. The deliberation network acts as a cross-checking mechanism between these two knowledge sources: it uses the internal knowledge provided by GAT to review and optimize the initial drafts provided by the search, thereby generating more accurate and detailed summaries.

3.2 CFG Encoder via GAT

Control-flow structure plays a central role in characterizing program execution semantics. However, serializing a CFG into a sequence inevitably obscures branching and looping topology, limiting the model's ability to reason about execution behavior. To preserve structural semantics explicitly, BRAD encodes bytecode-level CFGs using a Graph Attention Network (GAT) (Velickovic et al., 2017). The architecture of the CFG encoder is shown in Figure3.

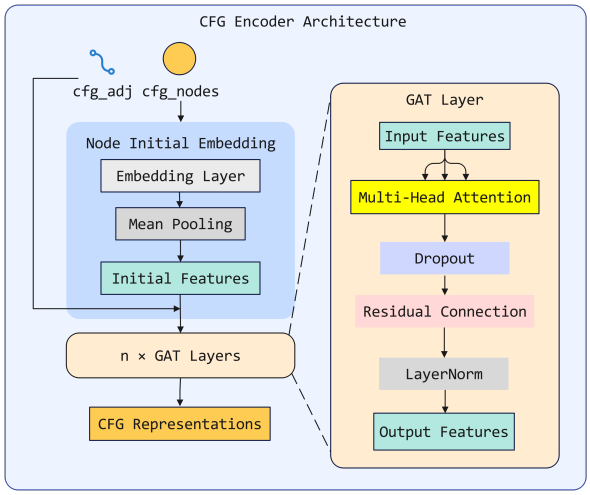

Figure 3: CFG Encoder Architecture.

For each method, we construct a bytecode-level CFG $G = (V, E)$, where nodes represent bytecode basic blocks and directed edges denote control-flow transfers.

Each node $v \in V$ is initialized using an opcode-based embedding:

$$\mathbf{h}_v^{(0)} = \mathrm{Embed}(\mathrm{opcode}(v)), \qquad (1)$$

which captures the local operational semantics of the bytecode block.

We apply multiple GAT layers to propagate information over the CFG. At layer l, node representations are updated as:

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{uv}^{(l)}\, \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right), \qquad (2)$$

where attention weights α_{uv} model the relative importance of different control-flow dependencies.

After the final GAT layer, node representations are pooled to produce a fixed-size graph embedding, which summarizes the execution structure of the method and is used in downstream deliberation.

3.3 Bytecode-Level Retrieval

3.3.1 Motivation and insight

A central challenge for retrieval-augmented comment generation is that lexical matching on source code does not reliably reflect functional equivalence: developers frequently implement the same behavior using different syntactic constructs and identifier names like Figure1, so surface token overlap at the source level is often a poor proxy for whether two fragments perform the same computation. Our guiding insight is that compiled bytecode abstracts many of these surface differences and yields a more normative representation of program behavior. Therefore, we assume that lexical overlap in the bytecode space is a more robust proxy metric for functional similarity. **That is to say, using bytecode for retrieval can find similar samples that are more relevant in terms of functional semantics, and their summaries are also more relevant to the true summaries of the queried samples.**

Based on this assumption, we adopt the efficient sparse retrieval tool BM25 to verify whether it is possible to retrieve more functionally relevant examples merely by matching the normalized bytecode sequence than by matching the original source code. Specifically, we construct a retrieval pipeline that (1) canonicalizes and tokenizes method-level bytecode, (2) indexes the bytecode corpus with a standard sparse retrieval model BM25 (Robertson et al., 2009), (3) query the index using the bytecode of the target method during training and inference, and (4) encodes the retrieved natural-language summary into a dense semantic representation using a Transformer-based text encoder. This representation serves as a draft embedding derived from

the retrieval, which provides high-level semantic guidance for the subsequent deliberation generation process.

3.3.2 BM25 Indexing and Scoring

We employ BM25 (Robertson et al., 2009), a widely used sparse retrieval function, to measure the similarity between a query bytecode sequence q (the canonicalized token sequence of the target method) and each candidate document d in the corpus. The BM25 score is defined as follows:

$$\text{score}(q,d) = \sum_{t \in q} \text{IDF}(t) \frac{f(t,d)(k_1+1)}{f(t,d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)} \tag{3}$$

Where $f(t,d)$ denotes the frequency of token t in document d, $|d|$ is the document length (in tokens), and avgdl represents the average document length across the corpus. The parameters k_1 and b are tunable hyperparameters (typically $k_1 \in [0.5, 2.0]$, $b \in [0,1]$). The inverse document frequency is computed as $\text{IDF}(t) = \log \frac{N - n_t + 0.5}{n_t + 0.5}$, where N is the total number of documents and n_t is the number of documents containing token t.

In practice, we build a BM25 index over the canonicalized bytecode corpus and apply the same preprocessing pipeline to the query bytecode during retrieval. Top-k neighbors are selected based on BM25 scores, and their corresponding summaries are used as initial drafts for downstream deliberation.

3.4 Deliberation Model

After obtaining source-code representations, bytecode CFG embeddings, and an initial summary draft retrieved from similar methods, BRAD applies a deliberation model to refine the summary iteratively. Instead of generating the final output in a single pass, the deliberation model treats the retrieved draft as an intermediate hypothesis and improves it through multiple refinement rounds. In the first pass, the generator produces an initial summary by combining the source code representation and the retrieval-derived draft embedding r via concatenation or gating. In subsequent passes, the model conditions on the previously generated summary while retaining the same contextual representations, enabling iterative revision. This iterative process enables the model to recover missing information, resolve semantic inconsistencies, and correct inaccurate descriptions.

4 EXPERIMENTS

4.1 Dataset

We use the dataset originally collected by Huang et al (Huang et al., 2025). The dataset is derived from a paired collection of JAR and source JAR files downloaded from a Maven repository (Saini et al., 2014), which contains 559 JAR packages. After alignment, filtering, and CFG extraction, the corpus contains 101,204 method-level examples. Each example is represented as a triple: (source code, reference annotation, serialized bytecode CFG).

4.2 Baseline

To assess the effectiveness of our proposed enhancements compared to existing methods, we compare them against a set of representative and strong baseline models drawn from different methodological families. These include conventional sequence-to-sequence models (eg, DeepCom (Hu et al., 2018), Hybrid-DeepCom (Hu et al., 2020), AST-AttendGRU (LeClair et al., 2019)), Transformer-based encoder–decoder models (eg, Transformer (Ahmad et al., 2020), CodeT5 (Wang et al., 2021)), bytecode-enhanced methods(eg, BC-Gen (Huang et al., 2023)), multi-stage generation systems (eg, DECOME (Mu et al., 2022)), and large language model(eg, GPT-4 (Achiam et al., 2023)).

4.3 Evaluation Metrics

To rigorously evaluate and compare various models on the code comment generation task, we adopt a suite of widely used automatic metrics in code summarization and related generation domains: BLEU1-4 (Papineni et al., 2002), ROUGE-L (Lin, 2004), METEOR (Banerjee and Lavie, 2005), and CIDEr (Vedantam et al., 2015). These metrics quantify different facets of the overlap and alignment between the generated comment and the reference (ground-truth) annotation.

4.4 Research Questions

In our experiments, we mainly focus on the following research questions: **RQ1**: How does BRAD perform compared to the code summarization baseline methods? **RQ2**: How do the two components of BRAD contribute to the overall model? **RQ3**: Does bytecode-level retrieval produce more useful drafts than source-level retrieval, leading to better final summaries? **RQ4**: How do human evaluation

Table 1: Test-set performance comparison (RQ1) — BLEU-1/2/3/4, ROUGE-L, METEOR, CIDEr.

Category	Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	METEOR	CIDEr
RNN/GRU-based	DeepCom	31.22	23.85	19.55	16.39	40.72	30.63	1.814
	Hybrid-DeepCom	34.43	26.76	22.26	18.96	44.63	34.42	2.372
	AST-AttendGRU	34.84	27.50	23.38	20.23	41.52	35.59	2.330
Transformer-based	Transformer	41.07	34.08	29.82	26.45	46.84	**42.45**	2.784
	BCGen	35.13	28.26	24.39	21.44	42.80	38.29	1.928
	DECOME	39.28	28.98	21.85	17.34	43.94	22.05	2.106
	CodeT5	35.46	27.50	23.09	19.84	42.87	38.16	2.011
Large Language Models	GPT-4	40.42	33.08	28.13	26.30	48.74	38.70	2.526
Ours	**BRAD**	**48.73**	**39.60**	**32.34**	**27.20**	**53.13**	28.10	**3.063**

Table 2: Ablation experiment results (RQ2).

Variant	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	METEOR	CIDEr
BRAD w/o GAT and BYretriever	39.28	28.98	21.85	17.34	43.94	22.05	2.106
BRAD w/o BYretriever	39.90	29.67	22.47	18.04	44.61	22.54	2.170
BRAD w/o GAT	45,63	36.96	29.98	24.24	50.03	26.32	2.795
BRAD	**48.73**	**39.60**	**32.34**	**27.20**	**53.13**	**28.10**	**3.063**

and case analysis reveal the quality of the summaries generated by BRAD?

5 RESULTS AND ANALYSIS

5.1 RQ1: How does BRAD perform compared to the code summarization baseline methods?

Table 1 reports the test-set performance of BRAD and a set of representative baselines on standard automatic metrics. BRAD attains the best performance across BLEU, ROUGE-L, and CIDEr metrics (Table 1).

Relative to the strong baseline Transformer, BRAD yields large absolute and relative improvements: BLEU-1 +7.66 (+18.7%), BLEU-2 +5.52 (+16.2%), BLEU-3 +2.52 (+8.5%), BLEU-4 +0.75 (+2.8%), ROUGE-L +6.29 (+13.4%), CIDEr +0.28 (+10.0%). Furthermore, BRAD also improves over the most closely related baseline DECOME (BLEU-4 +9.86 (+56.9%), ROUGE-L +9.19 (+20.9%), METEOR +6.05 (+27.4%), CIDEr +0.9 (+45.4%)), demonstrating that the gains are not solely attributable to model capacity but to the *modeling and retrieval choices* (i.e., graph-based bytecode encoding and bytecode-level retrieval) introduced by BRAD.

BRAD substantially outperforms all baseline models across multiple evaluation metrics, achieving consistent and statistically significant improvements on BLEU, ROUGE-L, and CIDEr. These results demonstrate that integrating a graph-based bytecode CFG encoder and bytecode-level retrieval yields richer semantic representations and more consistent summaries. **Therefore, BRAD effectively enhances the performance of code summarization tasks compared with existing baselines.**

5.2 RQ2: How do the two components of BRAD contribute to the overall model?

To comprehensively evaluate the contributions of different components in our proposed method, we conducted ablation studies by constructing three model variants: (1) BRAD w/o GAT and BYretriever, which removes both the Graph Attention Network and the bytecode-level retrieval module; (2) BRAD w/o BYretriever, which only eliminates the bytecode-level retrieval component while retaining GAT; (3) BRAD w/o GAT, which removes the Graph Attention Network but preserves bytecode-level retrieval. All variants were trained under identical experimental settings and hyperparameters as the complete BRAD model.

The experimental results presented in Table 2 clearly demonstrate the significance of each component. The complete BRAD model achieves the best performance across all evaluation metrics, substantially outperforming all ablated variants. Specifically, the removal of both GAT and BYretriever leads to the most significant performance degradation, with absolute reductions of 9.45, 10.62, 10.49, and 9.86 points in BLEU-1 through BLEU-4 scores, respectively. This substantial drop underscores the

Table 3: Comparison of BRAD using bytecode-level vs source-level retrieval (RQ3).

Variant	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	METEOR	CIDEr
BRAD-scRetrieval	40.00	30.00	22.60	18.10	44.60	22.50	2.182
BRAD-bcRetrieval	48.73	39.60	32.34	27.20	53.13	28.10	3.063

Table 4: LLM evaluation scores and significance tests: bytecode vs source retrieval (RQ3).

	Coherence	Consistency	Fluency	Relevance
BRAD-scRetrieval	2.49	2.44	3.43	2.43
BRAD-bcRetrieval	2.53	2.49	3.46	2.49
Paired t-test $p-value$	2.0×10^{-9}	5.6×10^{-7}	1.0×10^{-6}	6.4×10^{-10}
Wilcoxon $p-value$	2.5×10^{-8}	5.0×10^{-6}	1.4×10^{-6}	1.7×10^{-9}

complementary importance of both architectural components. Notably, the individual ablation of either component also reveals distinct contributions. The removal of each component will lead to a decline in model performance to varying degrees.

These findings collectively validate that both the Graph Attention Network and the bytecode-level retrieval module play vital and complementary roles in our framework. The GAT effectively models the structural information of bytecode, while the bytecode-level retrieval enhances the semantic retrieval capability, with their integration yielding synergistic improvements in code summarization performance.

5.3 RQ3: Does bytecode-level retrieval produce more useful drafts than source-level retrieval, leading to better final summaries?

To isolate the contribution of retrieval modality, we compare two BRAD variants that are identical except for the retrieval space used to obtain draft exemplars: (1) BRAD-bcRetrieval (bytecode-level retrieval over serialized CFG tokens) and (2) BRAD-scRetrieval (source-code-level retrieval). Table 3 presents their automatic-metric comparison on the test set. Across all metrics, bytecode-level retrieval consistently yields substantial improvements over source-level retrieval.

We complemented automatic evaluation with an LLM-based qualitative assessment using LLaMA-3-8B scoring on four human-oriented dimensions (Wu et al., 2025): Coherence (clearly reflects the main functionality and logic of the code without drifting off-topic), Consistency (faithful to the original code and does not introduce incorrect or extra information), Fluency (well-written, clear, and easy to read), and Relevance (highlights the code's key functional or business significance within the system) in Table 4. Although the absolute score differences are modest (0.03–0.06 on a 0–4 scale), paired significance testing over the full test set using both the paired t-test and the Wilcoxon signed-rank test yields statistically significant results for all dimensions ($p < 0.05$). The paired t-test assesses mean-level differences under approximate normality assumptions, while the Wilcoxon test provides a non-parametric confirmation that is robust to skewed score distributions.

These gains indicate that bytecode-based retrieval provides more functionally aligned exemplars for deliberation. Overall, we conclude that the improvements introduced by bytecode-level retrieval augmentation are not only consistent but also statistically reliable. This confirms that retrieving exemplars based on bytecode representations more effectively captures functional similarity and provides valuable semantic guidance for code summarization.

5.4 RQ4: How do human evaluation and case analysis reveal the quality of the summaries generated by BRAD?

5.4.1 Human Evaluation

Although the evaluation metrics can measure the lexical gap between the generated comments and the references, they can hardly reflect the semantic gap. Therefore, we conducted a human evaluation with six experienced developers (1–4 years of industrial experience) who were not authors. Following prior work (Huang et al., 2025; Mu et al., 2022), we randomly selected 50 code snippets from the test set, creating a questionnaire where each snippet was paired with the reference comment and

comments generated by different methods. Participants scored 300 <code snippet, comment> pairs, with comments shuffled to prevent bias. Each comment was rated on three aspects: (1) *Naturalness* for grammatical fluency, (2) *Informativeness* for information richness, and (3) *Usefulness* for helping understand code behavior, on a 0–4 scale. The final score for each comment is the average across the three scores.

Table 5 reports the results of the human evaluation. Overall, BRAD outperforms all non–large-language-model baselines and ranks just behind GPT-4 on all three criteria. In particular, BRAD consistently surpasses traditional neural and retrieval-based models, indicating that its generated comments are more fluent, information-rich, and helpful for understanding code behavior. While GPT-4 attains the highest scores, this outcome is expected given its training on massive, diverse corpora and the substantial computational and data resources involved. In contrast, BRAD is a task-specific model designed to enhance code summarization through structured bytecode reasoning and retrieval augmentation, without relying on large-scale general-purpose pre-training. These human evaluation results are consistent with trends observed in automatic metrics, further confirming that BRAD improves both the readability and semantic usefulness of generated comments.

Table 5: Human evaluation (RQ4) — Naturalness, Informativeness.

Model	Naturalness	Informativeness	Usefulness
Transformer	3.48	3.22	3.28
DeepCom	3.18	2.52	2.84
CodeT5	3.15	2.73	2.86
DECOME	2.98	2.46	2.63
GPT4	**3.73**	**3.32**	**3.48**
BRAD	3.65	3.29	3.45

5.4.2 Case Study

Figure 4 presents a representative case study illustrating how bytecode-level retrieval and structural encoding jointly improve the summarization of methods with conditional control logic. The *setQuiet* method involves condition-dependent behavior and state changes, which are difficult to capture using source-level similarity alone.

As shown, the ground-truth comment explicitly describes a conditional operation that suppresses error reporting and sets `failonerror` to false. GPT-4.0 produces a partially correct summary but fails to preserve the conditional dependency, while DECOME introduces irrelevant concepts due to semantically misaligned retrieval. BCGen generates an oversimplified description that ignores the conditional relationship.

Comparing the retrieved drafts, source-level retrieval yields comments focusing on surface operations without conditional semantics, whereas bytecode-level retrieval produces drafts aligned with the underlying control flow. In addition, the bytecode CFG encoding explicitly models the conditional structure, enabling the deliberation process to preserve "If true" and accurately describe the associated state change.

This case study demonstrates that bytecode-level retrieval provides semantically grounded drafts, while bytecode structural encoding supplies execution-aware constraints, and their combination enables more faithful summarization.

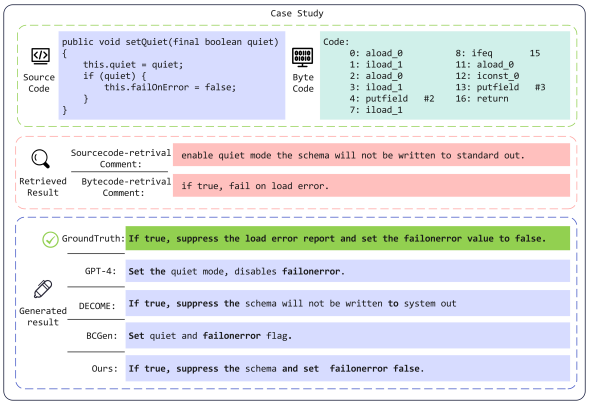

Figure 4: Examples of case analysis.

6 CONCLUSION

In this work, we propose **BRAD** (Bytecode Retrieval-Augmented Deliberation mode), a novel framework for automatic code comment generation that explores the integration of bytecode-level retrieval with deliberation-based code summarization. BRAD unifies graph-based bytecode representation and retrieval-augmented multi-pass deliberation, enabling the model to exploit control-flow semantics and retrieve functionally equivalent exemplars beyond surface lexical similarity. The assessment results show that BRAD has achieved significant improvements over strong baselines in multiple automatic metrics. The human evaluation also confirms the comments generated by BRAD tend to be more readable, informative, and useful.

7 Limitations

While BRAD demonstrates strong performance on the evaluated benchmarks, several limitations remain. Our experiments are limited to Java programs and adopt a sparse bytecode-level retrieval strategy, leaving the potential of alternative retrieval paradigms, such as dense representations, unexplored. In addition, the framework does not consider multilingual or cross-language code summarization scenarios, and its generalization to other programming languages with different compilation pipelines is unclear. Moreover, incorporating bytecode-level retrieval and structural encoding introduces additional computational overhead compared to purely source-level approaches, which may affect scalability in large-scale or latency-sensitive settings. Future work will focus on improving retrieval efficiency and extending the framework to broader language scenarios.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pages 1–5.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Aakash Bansal, Zachary Eberhart, Zachary Karas, Yu Huang, and Collin McMillan. 2023. Function call graph context encoding for neural source code summarization. *IEEE Transactions on Software Engineering*, 49(9):4268–4281.

Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology*, 30(2):1–29.

Xiangping Chen, Junqi Chen, Zhilu Lian, Yuan Huang, Xiaocong Zhou, Yunzhi Wu, and Zibin Zheng. 2025. An alternative to code comment generation? generating comment from bytecode. *Information and Software Technology*, 179:107623.

Sergio Cozzetti B De Souza, Nicolas Anquetil, and Káthia M De Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–32.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. 2021. Transformer in transformer. *Advances in neural information processing systems*, 34:15908–15919.

Shifu Hou, Lingwei Chen, Mingxuan Ju, and Yanfang Ye. 2023. Leveraging comment retrieval for code summarization. In *European Conference on Information Retrieval*, pages 439–447. Springer.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pages 200–210.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217.

Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *Proceedings of the 44th international conference on software engineering*, pages 1693–1705.

Yuan Huang, Jinbo Huang, Xiangping Chen, Kunning He, and Xiaocong Zhou. 2023. Bcgen: a comment generation method for bytecode. *Automated Software Engineering*, 30(1):5.

Yuan Huang, Jinbo Huang, Xiangping Chen, and Zibin Zheng. 2025. Towards improving the performance of comment generation models by using bytecode information. *IEEE Transactions on Software Engineering*.

Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020a. Towards automatically generating block comments for code snippets. *Information and Software Technology*, 127:106373.

Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. 2020b. Does your code need comment? *Software: Practice and Experience*, 50(3):227–245.

Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 795–806. IEEE.

Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1571–1575.

Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*.

Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Doyle Paul. 2022. Setransformer: A transformer-based code semantic parser for code comment generation. *IEEE Transactions on Reliability*, 72(1):258–273.

Zhifang Liao, Xiaoyu Liu, Peng Lan, Song Yu, and Pei Liu. 2025. Cmdesum: A cross-modal deliberation network for code summarization. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension*, pages 250–261. IEEE Computer Society.

Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, pages 184–195. IEEE.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Hanzhen Lu and Zhongxin Liu. 2024. Improving retrieval-augmented code comment generation by retrieving for generation. In *2024 IEEE International Conference on Software Maintenance and Evolution*, pages 350–362. IEEE.

Vishal Misra, Jakku Sai Krupa Reddy, and Sridhar Chimalakonda. 2020. Is there a correlation between code comments and issues? an exploratory study. In *Proceedings of the 35th Annual ACM symposium on applied computing*, pages 110–117.

Mohamed Atef Mosa, Alaa Hamouda, and Mahmoud Marei. 2017. Ant colony heuristic for user-contributed comments summarization. *Knowledge-Based Systems*, 118:105–114.

Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic comment generation via multi-pass deliberation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2021. Impact of evaluation methodologies on code summarization. *arXiv preprint arXiv:2108.09619*.

Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering*, 24(3):1418–1457.

Kazu Nishikawa, Genta Koreki, and Hideyuki Kanuka. 2024. Enhancing source code comment generation via retrieval-augmented generation with design document term dictionary. In *2024 31st Asia-Pacific Software Engineering Conference*, pages 467–471. IEEE.

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th international conference on software engineering*, pages 2006–2018.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Vaibhav Saini, Hitesh Sajnani, Joel Ossher, and Cristina V Lopes. 2014. A dataset for maven artifacts and bug patterns found in them. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419.

Chaochen Shi, Borui Cai, Yao Zhao, Longxiang Gao, Keshav Sood, and Yong Xiang. 2023. Coss: Leveraging statement semantics for code summarization. *IEEE Transactions on Software Engineering*, 49(6):3472–3486.

Jikyoeng Son, Joonghyuk Hahn, HyeonTae Seo, and Yo-Sub Han. 2022. Boosting code summarization by embedding code structures. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 5966–5977.

Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, and 1 others. 2024. Code needs comments: Enhancing code llms with comment augmentation. *arXiv preprint arXiv:2402.13013*.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, pages 43–52.

Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st international conference on program comprehension*, pages 83–92. Ieee.

Lin Tan. 2015. Code comment analysis for improving software quality. In *The art and science of analyzing software data*, pages 493–517. Elsevier.

Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, and 1 others. 2017. Graph attention networks. *stat*, 1050(20):10–48550.

Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th international conference on software engineering*, pages 287–298.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 349–360.

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension*, pages 53–64. IEEE.

Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International conference on software analysis, evolution, and reengineering*, pages 380–389. IEEE.

Yang Wu, Yao Wan, Zhaoyang Chu, Wenting Zhao, Ye Liu, Hongyu Zhang, Xuanhua Shi, Hai Jin, and Philip S Yu. 2025. Can large language models serve as evaluators for code summarization? *IEEE Transactions on Software Engineering*.

Yingce Xia, Fei Tian, Lijun Wu, Jianxin Lin, Tao Qin, Nenghai Yu, and Tie-Yan Liu. 2017. Deliberation networks: Sequence generation beyond one-pass decoding. *Advances in neural information processing systems*, 30.

Kaiyuan Yang, Junfeng Wang, and Zihua Song. 2023. Learning a holistic and comprehensive code representation for code summarization. *Journal of Systems and Software*, 203:111746.

Chi Yu, Guang Yang, Xiang Chen, Ke Liu, and Yanlin Zhou. 2022. Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In *2022 IEEE International Conference on Software Maintenance and Evolution*, pages 82–93. IEEE.

Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. Cpc: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 1359–1371.

Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. 2022. A survey of automatic source code summarization. *Symmetry*, 14(3):471.

Xiaoning Zhu, Chaofeng Sha, and Junyu Niu. 2022. A simple retrieval-based method for code comment generation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 1089–1100. IEEE.

A Implementation Details

All experiments were executed on a machine equipped with an NVIDIA GeForce RTX 4090 (driver 550.107.02, CUDA 12.4); we used PyTorch with automatic mixed precision (AMP) to maximize throughput and memory efficiency. Training largely follows DECOME's setup: mini-batch size = 32, number of deliberation passes = 3, dropout = 0.1, and we train up to 100 epochs with early stopping monitored on validation BLEU-4 / ROUGE-L. The generator is optimized with AdamW (weight decay = 0.01, learning rate for the generator 3e-5), while the GAT encoder (jointly fine-tuned) uses a higher LR (1e-3); gradients are clipped to norm 1.0. The GAT encoder is configured as 2 layers × 4 heads with node hidden size 256 and layer dropout 0.1. Other engineering details are provided in the project repository.

B Bytecode Preprocessing and Normalization

Raw bytecode often contains low-level details (e.g., constant values, literal addresses, or local variable indices) that are irrelevant to program functionality and may fragment the retrieval space. To improve retrieval quality, we apply a lightweight canonicalization process before indexing, transforming each method's bytecode into a normalized token sequence that emphasizes its computational structure and control flow. The preprocessing procedure consists of the following steps:

1. **Opcode extraction.** Each bytecode instruction is converted into its opcode name token (e.g., `iload`, `invokevirtual`, `if_icmplt`), optionally retaining short operand type signatures while omitting literal values.

2. **Operand normalization.** Identifiers such as variable indexes or constant literals are replaced with placeholders, e.g., `iload_1` → `iload VAR`, `ldc #5` → `ldc CONST`.

3. **Basic-block separators.** A special token (e.g., `BBSEP`) is inserted between basic blocks to retain coarse-grained control-flow boundaries in the serialized sequence.

4. **Edge-type markers (optional).** When available, control-flow edge types are preserved through special tokens (e.g., `JUMP`, `FALLTHROUGH`, `EXC`), enhancing the representational capacity for control-flow signals even within a linearized format.

5. **Sequence truncation.** Extremely long bytecode sequences are truncated or windowed to maintain tractable document lengths for BM25 retrieval.

This canonicalization removes spurious variability (e.g., register numbering or literal constants) and ensures that indexing focuses on structural patterns and control signals that better reflect program functionality.