Code vs Serialized AST Inputs for LLM-Based Code Summarization: An Empirical Study

Shijia Dong
University of Glasgow
Glasgow, United Kingdom
2810995d@student.gla.ac.uk

Haoruo Zhao
University of Glasgow
Glasgow, United Kingdom
Haoruo.Zhao@glasgow.ac.uk

Paul Harvey
University of Glasgow
Glasgow, United Kingdom
Paul.Harvey@glasgow.ac.uk

Abstract

Summarizing source code into natural language descriptions (code summarization) helps developers better understand program functionality and reduce the burden of software maintenance. Abstract Syntax Trees (ASTs), as opposed to source code, have been shown to improve summarization quality in traditional encoder–decoder-based code summarization models. However, most large language model (LLM)-based code summarization methods rely on raw code or only incorporate partial AST signals, meaning that the potential of complete AST representation has not been fully explored for LLMs.

This paper presents AST(NIT), an AST augmentation and serialization method that preserves lexical details and encodes structural information into LLM-compatible sequences. Experiments with the LLaMA-3.1-8B model on the CodeXGLUE Python dataset show that the proposed serialized ASTs reduce the length of LLM inputs, require shorter training times, and achieve summarization quality comparable to existing approaches.

CCS Concepts

• **Software and its engineering** → *Documentation*; **General programming languages**; • **Computing methodologies** → **Natural language processing**.

Keywords

Abstract Syntax Trees, Source Code Summarization, Large Language Models

1 Introduction

Code summarization aims to automatically generate concise natural language descriptions of source code, providing significant value for program understanding, software maintenance, and collaborative development [46].

In existing encoder–decoder-based code summarization, explicitly incorporating structural and semantic information into input representations for the code summarization process provides richer context and improves performance [6]. Among various input representations, Abstract Syntax Trees (ASTs) [7] are hierarchical, tree-based abstractions of source code that have been widely adopted and have demonstrated strong performance [18, 19, 22, 37]. While programming languages follow strict grammatical rules, models that process flat code tokens often struggle to capture the structural relationships inherent in code [24]. ASTs address this limitation by explicitly representing code's hierarchical structure, thereby providing richer context and enabling neural models to produce more accurate summaries.

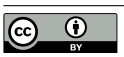
More recently, large language models (LLM)-based code summarization methods have demonstrated remarkable performance [13, 23]. The successful adoption of ASTs in earlier encoder–decoder-based models suggests that leveraging complete ASTs may also benefit LLMs. However, in recent LLM-based code summarization, most works incorporate only partial structural signals from ASTs in the model input, such as data or control flow edges and tagged identifiers [2, 27]. Such designs fail to preserve the complete structural context of the original tree, resulting in the loss of hierarchical relationships and control structures, potentially limiting the model's ability to generate precise code summaries from the AST representation. One possible way to fully exploit the complete AST is to serialize the entire tree into a linear sequence, as in the Structure-Based Traversal (SBT) method proposed by Hu et al. [18], which has been shown to improve summarization quality in encoder–decoder settings. However, LLMs process inputs purely as sequences without explicit structural encoders, so it remains unclear whether the benefits of AST serialization extend to this setting. This leads to the following research question:

RQ: Under LLM fine-tuning, can serialized ASTs achieve comparable or superior method-level summarization quality to code sequences?

In this paper, *serialized ASTs* refer to linearized representations of ASTs obtained through a tree-node traversal; *code sequences* denote tokenized source code text without explicit structural information; *Method-level summarization* focuses on generating single-sentence natural language descriptions for individual functions or methods.

To address this question, we propose **AST(NIT)**, an AST augmentation and serialization method that preserves lexical details and encodes tree structure into LLM-compatible sequences. We systematically evaluate four input representations—AST(NIT), AST(Preorder), AST(SBT), and Code on the code summarization task using the CodeXGLUE (Python) subset [29], under identical fine-tuning and decoding settings. Experimental results show that, for method-level code summarization, serialized ASTs can achieve summarization quality comparable to code sequences when used as LLM inputs. Compared with the SBT method, AST(NIT) achieves comparable summary quality while reducing average input length by 28.6% and total training time by 11.3%, resulting in measurable efficiency improvements. In short, the contributions of this work are as follows:

- AST(NIT), an AST augmentation and serialization method that preserves lexical details and encodes tree structure into LLM-compatible sequences;



- Empirical evaluation of complete serialized AST sequences as the sole input to LLMs for method-level code summarization;
- Systematic comparison of Code, AST(Preorder), AST(SBT), and AST(NIT) under an identical LLM fine-tuning and decoding setup. The results indicate that serialized ASTs can achieve summarization quality comparable to code sequences, and that AST(NIT) provides measurable efficiency improvements over AST(SBT).

2 Background & Motivation

2.1 Code Summarization

Code summarization is the process of generating natural language descriptions for code [36]. An accurate summary helps developers quickly understand program intent, facilitating collaboration and software maintenance [46]. The research in the code summarization field has evolved from early template-based and information retrieval methods [11, 15, 30] to neural encoder-decoder-based models [12, 18, 20, 35]. In recent years, LLMs have made rapid progress in code summarization [13, 34].

In addition to the continued evolution of model architectures, another prominent trend in this field is the incorporation of structural and semantic information [6] such as ASTs [3, 18], data flow [14], and control flow graphs [42] to enrich the input representation and provide models with richer context, thereby improving summary quality.

Typically, the granularity of existing code summarization approaches is at the operation, method, or class level [47]. In this work, we focus on the method level, aiming to generate concise and accurate summaries for individual functions or methods.

2.2 LLaMA-3.1

LLMs are advanced neural models based on the transformer architecture [39] and have achieved remarkable results in code understanding and generation tasks [33, 41].

Among these models, we adopt *LLaMA 3.1*, an open-source LLM released by Meta in 2024, due to its strong performance on code tasks and robust long-context modeling capabilities [10]. These features make it well suited for experiments that require processing large inputs, such as serialized AST sequences. *LLaMA 3.1* is available in several parameter sizes (8B, 70B, 405B); we select the *LLaMA-3.1-8B*[1] as it provides a good balance between performance and computational feasibility for fine-tuning on a single GPU (see Section 4 for details).

2.3 Abstract Syntax Tree (AST)

An AST [7] is a hierarchical, tree-based representation of the abstract syntactic structure of source code, explicitly encoding both syntactic and structural information. AST representations, when used as model inputs, have demonstrated notable performance improvements across various code-related tasks, such as code summarization [22], code search [17], code clone detection [44]. In a typical AST, terminal (leaf) nodes represent variables and types, while non-terminal (internal) nodes denote syntactic constructs such as loops, expressions, or declarations [37].

[1]https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

2.4 Motivation

Inspired by the strong performance of ASTs in encoder–decoder-based models for code summarization, recent LLM-based approaches have increasingly explored the potential of ASTs. However, most existing LLM studies utilize only partial structural signals from ASTs, such as data flow or tagged identifiers, incorporated as prompt augmentations by appending these structural hints to the text prompt [2, 27]. Compared with using the full AST as input, this partial-structure design overlooks the complete structural context of the original tree, causing the hierarchical relationships and control structures within the AST to be lost. This limits the model's ability to accurately capture code semantics (functional meaning and logical intent) from the AST representation and to generate precise code summaries.

Given that SBT methods [18, 22] have shown that AST serialization can effectively support code summarization within encoder–decoder frameworks, a potential solution for leveraging complete AST in LLM-based code summarization is to use a serialized AST as an independent input to the model. However, there are two main challenges that need to be overcome:

Listing 1: Check negative balance

```
1 def check_negative_balance(operations):
2     balance = 0
3     for op in operations:
4         balance += op
5         if balance < 0:
6             return True
7     return False
```

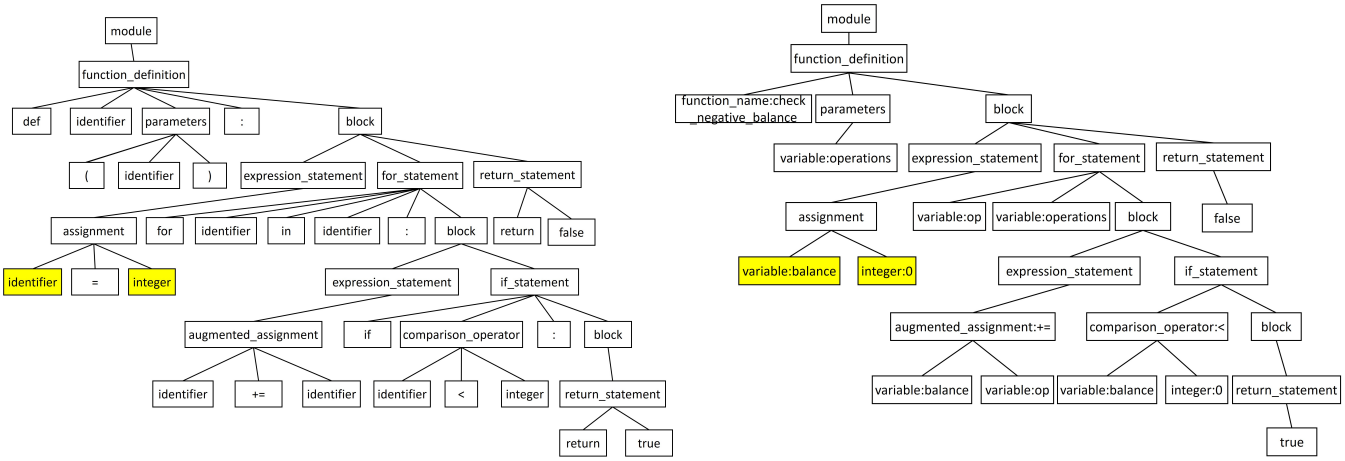
(1) **Loss of lexical detail.** Prior work has shown that identifiers carry critical lexical details from the original code, strongly influence model performance on code summarization tasks [1]. However, in standard ASTs, these identifiers are often abstracted away, as terminal nodes typically record only the type without the concrete values [37]. In Listing 1, for the statement *"balance = 0"* (line 2), the nodes corresponding to *"balance"* and *"0"* are represented only as *"identifier"* and *"integer"* (Fig. 1a).

(2) **Structural mismatch.** Without a structure-aware encoder, the key challenge is to serialize the AST while preserving its original hierarchical relationships (i.e., the parent–child and nested node structure that reflects how statements and expressions are organized in the code), ensuring LLMs can effectively capture and utilize this structural information to generate accurate summaries.

3 Methods

To address the challenges outlined in Section 2, we propose AST(NIT), an AST augmentation and serialization method that preserves lexical details and encodes tree structure into LLM-compatible sequences.

Our workflow (Fig. 2) consists of two stages: (i) **model fine-tuning**, which includes parsing, AST augmentation, AST serialization, and fine-tuning, and (ii) **code summary inference**. The



(a) Raw AST from Tree-sitter. **(b) AST after lexical injection and structural normalization**

Figure 1: Side-by-side comparison of (a) the raw AST and (b) the augmented AST for Listing 1.

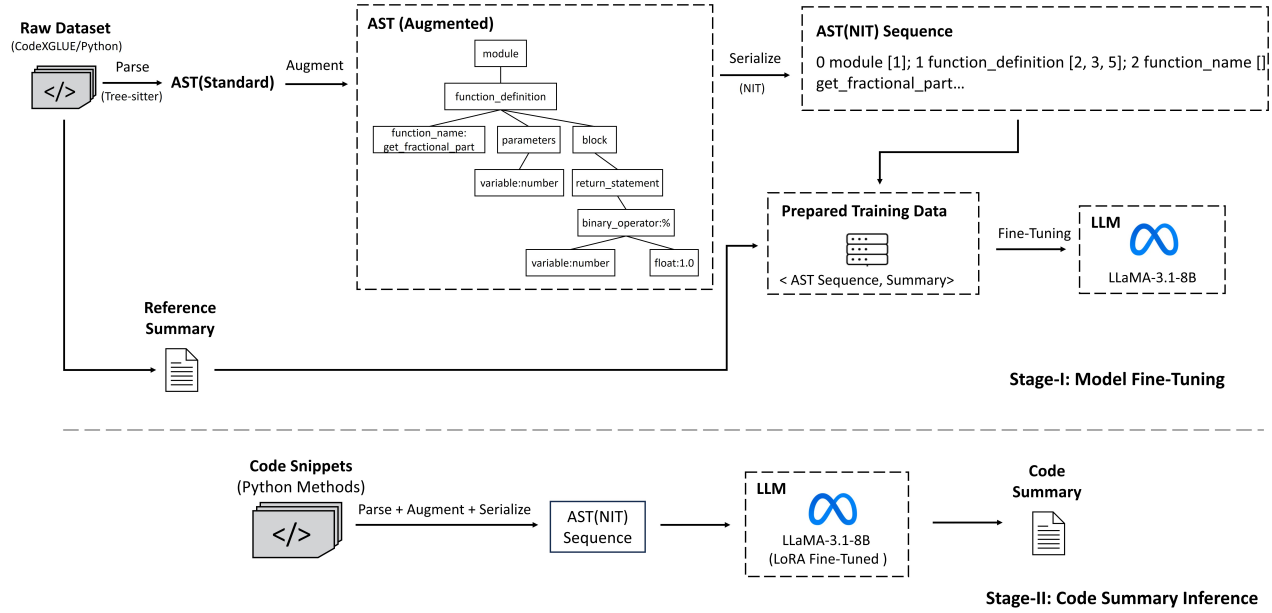


Figure 2: The overall workflow of the proposed method AST(NIT) for code summarization with serialized AST inputs.

first three steps: parsing, AST augmentation, and AST serialization together constitute AST(NIT). We describe the implementation of AST(NIT) below, while the fine-tuning configuration is detailed in Section 4. The code summary inference stage follows the trained model's decoding setup, also described in Section 4.

3.1 AST Augmentation

In this work, we consider Python code as prior work [37] reports that AST encoding is more effective for Python, as its corresponding ASTs are relatively shorter and thus allow structural information to contribute more significantly than in Java. We use the open-source Python project *Tree-sitter*[2] to parse code snippets into ASTs. However, as discussed in Section 2, standard ASTs generated by such parsers do not retain all lexical details from the source code. Moreover, standard ASTs are typically large (see Figure 1a) and contain numerous unnamed nodes [8], such as commas and parentheses. For some complex methods, serializing the corresponding ASTs can produce long sequences that may approach or exceed the context window of an LLM. In addition, longer inputs increase computational cost and can diffuse the model's attention over less

[2]https://tree-sitter.github.io/tree-sitter

informative tokens, ultimately degrading the quality of generated code summaries. To address these issues, we modify the parsed ASTs as follows:

(1) **Lexical Injection.** We explicitly inject lexical information into terminal nodes by appending identifier names and embedding numeric or string literals. In addition, we refine certain node types to better reflect their semantic roles: for example, an *"identifier"* within a function definition is relabeled as *"function_name"*, while identifiers in assignment expressions are relabeled as *"variable"*. After modification, the nodes corresponding to *"balance"* and *"0"* become *"variable:balance"* and *"integer:0"*, as highlighted in Figures 1a and 1b.

(2) **Structural Normalization.** We remove most unnamed nodes unless they carry clear semantic value (such as colons in slice expressions). For operator nodes, we embed the actual operator as its value (e.g., *"binary_operator:+"*) rather than creating a separate child node. This effectively reduces the size of the tree and eliminates irrelevant noise. For the example shown in Figures 1a and 1b, the number of AST nodes decreases from **41** in the raw tree to **27** after structural normalization.

This modification augments the AST with lexical details for code summarization, while reducing overall tree size and preserving its original structure and depth, making it suitable as input to LLMs. For comparison, we also evaluate the raw AST without any lexical injection or structural normalization in Section 4 (see AST(Preorder)).

3.2 AST Serialization

To make the augmented AST compatible with sequence-based LLMs, we serialize it into a flat token sequence using a **N**ode-**I**ndex **T**raversal (**NIT**). We refer to the final linear representation as **AST(NIT)**, emphasizing that it includes both augmentation and serialization.

Traversal procedure: Starting at the root, we perform a Depth-First Search (DFS) based preorder traversal [38] and assign each visited node a globally unique integer identifier (ID) in order of visitation. Each node is recorded as fixed-field tuple and appended into the token sequence, with individual nodes separated by semicolons (;). The tuple comprises the following fields:

- **ID**: the unique index of the node;
- **Type**: the node type (e.g., *"function_definition"*, *"call"*);
- **Value** (optional): concrete lexical content, if present (e.g., identifier names, literals, embed operators);
- **Children** (optional): the list of child IDs in visitation order, if any.

We use a DFS-based preorder traversal because it follows the nested structure of code, exploring each execution path to completion before backtracking. For each node, we record its attributes in a fixed field order, making the structural information explicit in the serialized sequence. As a result, the entire AST can be encoded as a flat sequence, which can then be used as a direct input to LLMs.

4 Experimental Evaluation

To systematically evaluate serialized AST representations for code summarization, we first introduce the experimental setup and evaluation metrics, and then describe the four input representations in this section: Code, AST(Preorder), AST(SBT), and AST(NIT). Quantitative results and qualitative analysis are presented in Section 5.

4.1 Experimental Setup

Dataset. We use the Python subset of the code summarization task in CodeXGLUE [29], a widely used benchmark dataset for program understanding. We first randomly sample approximately 50,000 methods for training, 5,000 for validation, and 5,000 for testing. We then apply the filtering procedure to the existing records in the dataset adopted in prior work [18, 37]: (i) remove methods whose reference summary contain fewer than four words; (ii) exclude constructors, property accessors, and test cases, as their summaries are typically trivial for the model to generate and may artificially inflate performance estimates [18]; (iii) discard duplicate samples; (iv) if a summary contains multiple sentences, retain only the first sentence; and (v) remove samples whose AST cannot be successfully parsed. After cleaning, the final splits contain 30,227 training, 2,771 validation, and 3,097 test instances (Table 1).

Table 1: Dataset statistics (token counts measured with the LLaMA-3.1 tokenizer).

Split	Count	Code length (tokens)			Summary length (tokens)		
		Min	Mean	Max	Min	Mean	Max
Train	30227	16	85.97	474	4	9.47	38
Valid	2771	17	84.15	395	4	9.43	31
Test	3097	16	85.20	433	4	9.33	31

Model. To ensure a fair comparison across input representations, we fine-tune the *LLaMA-3.1-8B* model using the above datasets, rather than relying solely on its pre-trained parameters, thereby reducing bias from pre-training and prompt sensitivity. All experiments use the *Meta-Llama-3.1-8B-Instruct*[3] checkpoint with 4-bit quantization via the *BitsAndBytes*[4] library, and the same configuration is applied for all input representations.

Training Details. We apply Parameter-Efficient Fine-Tuning [9] with *LoRA* [16], implemented using the *unsloth*[5] library which substantially reduces memory and time cost. *LoRA* is applied with rank $r = 16$, $\alpha = 16$, and dropout = 0.05 to the projection layers, combined with gradient checkpointing for memory efficiency. Across all input representations, we use the **same** hyper-parameters: learning rate = 5×10^{-5}, warm-up ratio = 0.05, weight decay = 0.01, and max gradient norm = 1.0. We train for three epochs with a context window of 5,000 tokens for all conditions, using the AdamW (8-bit) optimiser [28] and mixed-precision. Since the average input length varies across the four input representations—Code, AST(Preorder), AST(SBT), and AST(NIT), we adopt token-based batching targeting $\approx$ 50,000 tokens per update to reduce gradient variance. All checkpoints are retained during training, and the checkpoint with the highest validation BLEU-4 [31] is selected for test evaluation. Inference uses deterministic decoding (beam = 4, length penalty = 0.6,

[3] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
[4] https://github.com/bitsandbytes-foundation/bitsandbytes
[5] https://unsloth.ai

Table 2: Example input representations for the same function, from Listing 1 (truncated, * denotes our method).

Representation	Example
AST(NIT)*	`0 module [1]; 1 function_definition [2,3,5]; 2 function_name [] check_negative_balance; 3 parameters [4]; 4 identifier [] operations; 5 block [6,10,25]; 6 expression_statement [7]; 7 assignment [8,9]; 8 identifier [] balance; 9 integer [] 0; ...`
AST(Preorder)	`module function_definition identifier parameters identifier block expression_statement assignment identifier ...`
AST(SBT)	`(module (function_definition (function_name_check_negative_balance) function_name_check_negative_balance (parameters (identifier_operations) identifier_operations)parameters (block (expression_statement (assignment (identifier_balance)identifier_balance (integer_0)integer_0)assignment)expression_statement ...`
Code	`def check_negative_balance(operations): balance = 0 ...`

max_new_tokens = 64). All input types share the same zero-shot style template.

Hardware. All models are trained on a single NVIDIA A6000 GPU (48GB).

4.2 Evaluation Metrics

We evaluate the generated summaries on the test set using four widely adopted metrics for code summarization: (1) **BLEU-4** [31] measures the precision of overlapping n-grams between a generated summary and its reference. Following prior work [18], we set $n = 4$ to balance local phrase accuracy with overall fluency. It captures how many contiguous sequences of up to four tokens appear in both texts. (2) **METEOR** [5] computes similarity through unigram alignment between the generated and reference summaries. Unlike BLEU-4, METEOR allows flexible matching based on exact forms, stemming, and synonyms [24], making it more sensitive to linguistic variation and synonym usage. (3) **ROUGE-L** [25] evaluates sentence-level similarity by identifying the longest common subsequence between a generated and a reference summary. By focusing on the longest in-sequence matches, ROUGE-L rewards summaries that preserve the overall order of key words or phrases, rather than isolated n-gram matches. (4) **BERTScore** [45] moves beyond lexical overlap and measures semantic similarity by leveraging contextualised embeddings from a variant of BERT [36]. Each token in the generated summary is aligned to its most similar reference token using embedding-based cosine similarity, and overall precision, recall, and F1 scores are then computed. This allows BERTScore to capture semantic equivalence even when surface forms differ.

4.3 Baselines

We compare the performance of the following four input representations on the code summarization task. Models trained with different input representations share **identical** hyperparameters and training/inference settings. Table 2 shows encodings of Listing 1 in each representation.

(1) **AST(NIT) (our method).** An AST sequence obtained by first parsing the source code with *Tree-sitter*, then applying Lexical Injection and Structural Normalization to augment the tree, and finally serializing the augmented AST using our proposed Node-Index Traversal (NIT) (see Section 3)
(2) **AST(Preorder).** An AST sequence obtained by parsing the source code with *Tree-sitter* and serializing the unmodified tree via preorder traversal without lexical injection or structural normalization. AST(Preorder) is a purely structural representation: it retains only AST node types and excludes identifier names and literal values.
(3) **AST(SBT).** An AST sequence obtained by parsing with *Tree-sitter* and serialized via Structure-Based Traversal (SBT) [18]. SBT encodes tree structure using bracket-style markers and injects concrete lexical values at terminal nodes. Although originally demonstrated on Java, we apply the same serialization strategy to Python to ensure a fair comparison. AST(SBT) thus preserves identifier-level lexical information and does not require the raw source code as a separate input.
(4) **Code.** The source code token sequence. As a lexical representation that does not explicitly encode structural information, Code constitutes the most direct input format for code-related tasks. We use it as a baseline to compare the effectiveness of explicit AST-based input representations.

5 Results

To address the research question stated in Section 1, we evaluate the four input representations from two perspectives: (i) summary quality, assessed by BLEU-4, METEOR, ROUGE-L, and BERTScore under identical fine-tuning and decoding settings; and (ii) efficiency, measured by average input length, total trained tokens, training time, and peak memory usage. Qualitative analysis and discussion are also presented in this section.

5.1 Code vs AST(NIT) vs AST(Preorder)

Table 3 shows that, across all evaluation metrics, Code and AST(NIT) achieve near-identical results. This indicates that, under LLM fine-tuning, using either code sequences or a structured representation like AST(NIT) as input yields comparable code summarization performance. However, AST(Preorder) substantially underperforms (e.g., BLEU-4: 11.75). This is mainly because the absence of lexical details in AST(Preorder): user-defined identifiers and literals are

Table 3: Comparison of code summarization performance on BLEU, METEOR, ROUGE-L, and BERTScore metrics for different input representations (* denotes our method).

Input	BLEU-4	METEOR	ROUGE-L	BERTScore		
				Precision	Recall	F1
AST(NIT)*	23.07	0.39	0.48	0.93	0.91	0.92
AST(Preorder)	11.75	0.19	0.25	0.89	0.88	0.89
AST(SBT)	23.22	0.40	0.48	0.93	0.91	0.92
Code	23.48	0.39	0.49	0.93	0.91	0.92

collapsed into generic node types, leading to semantic loss. By contrast, Code retains lexical cues natively, and AST(NIT) reintroduces them via lexical injection. These observations indicate that lexical information is essential for code summarization, align with prior findings [1], and indicate the effectiveness of the AST-augmentation design used in AST(NIT).

5.2 AST(SBT) vs AST(NIT)

Table 3 shows that our proposed method, AST(NIT), achieves summarization quality comparable to the AST(SBT) across all evaluation metrics. From the efficiency perspective (Table 4), AST(NIT) reduces the average input length by approximately 28.6% compared to AST(SBT), and shortens total training time by 11.3% , with similar peak memory usage. This efficiency gains come from AST(NIT)'s compact fixed-field node tuple and child-ID list design, which avoids the repeated type markers and bracket nesting of AST(SBT). In summary, our method achieves summarization performance comparable to AST(SBT) while using shorter input sequences and has lower training cost.

Table 4: Training statistics across different input representations (* denotes our method).

Input Representation	Avg Length (tokens)	Total Trained Tokens (M)	Training Time (h)	Peak Memory Usage (GB)
AST(NIT)*	470.92	14.23M	11.81	11.50
AST(Preorder)	133.94	4.05M	3.57	8.80
AST(SBT)	659.17	19.90M	13.32	11.50
Code	117.82	3.56M	3.04	7.61

5.3 Qualitative Analysis

We present two representative examples to qualitatively assess the semantic accuracy of summaries generated from different input representations.

In Listing 2, summaries from Code, AST(NIT), and AST(SBT) all capture the intended file deletion operation, with minor lexical differences such as *"lock file"* or *"temporary file"*, as shown in Table 5. In contrast, AST(Preorder) incorrectly describes the action as *"removing a directory"*, missing the file-specific context.

Listing 2: cleanup

```
def cleanup(self):
    if os.path.exists(self.path):
        os.remove(self.path)
```

Table 5: Generated Summaries for Listing 2 Across Input Representations

Reference	Clean up files in the specified path.
AST(NIT)*	Removes the temporary file.
AST(SBT)	Removes the temporary file.
AST(Preorder)	Removes the directory.
Code	Remove the lock file.

In Listing 3, both Code and lexical-injected AST inputs (AST(NIT), AST(SBT)) accurately describe the creation of a virtual environment, as shown in Table 6. However, AST(Preorder) fails to capture this intent and instead generates a summary about creating a file instance.

Listing 3: create

```
def create(env_dir, system_site_packages=False,
    clear=False, symlinks=False, with_pip=False,
     prompt=None):
    builder = ExtendedEnvBuilder(
        system_site_packages=
            system_site_packages,
        clear=clear,
        symlinks=symlinks,
        with_pip=with_pip,
        prompt=prompt
    )
    builder.create(env_dir)
    return builder.context
```

Table 6: Generated Summaries for Listing 3 Across Input Representations

Reference	Create a virtual environment in a directory.
AST(NIT)*	Create a virtual environment in the directory.
AST(SBT)	Create a virtual env in the given directory.
AST(Preorder)	Create an instance of file.
Code	Create a virtual environment in the given directory.

In these two examples, when lexical information is preserved, either natively in Code or via lexical injection in AST(NIT) and AST(SBT), the generated summaries are generally more specific and better capture the intended functionality. In contrast, purely structural inputs such as AST(Preorder) often result in information loss and less precise outputs, consistent with Table 3. Meanwhile, AST(NIT) produces code summaries comparable to those from Code in both accuracy and detail.

5.4 Discussion

Together the results from Table 3 with qualitative analysis, we find that while serialized ASTs (e.g., SBT) once offered clear benefits in encoder–decoder model, our empirical evaluation suggests that, under LLM fine-tuning, serialized ASTs achieve summary quality comparable to code sequences.

This shift can be explained by two possible situations. First, emergent abilities from large-scale pretraining enable LLMs to internalize structural and semantic information directly from code without requiring explicit AST signals. Second, because modern LLMs are pretrained on massive corpus that likely include human-curated datasets, or close variants, used in current benchmarks, the evaluation setting differs fundamentally from that of traditional encoder–decoder models. In our fine-tuning scenarios, it is no longer possible to guarantee a strict separation between training and test sets, meaning that the model may already have encountered the benchmark data or similar examples during pretraining. This overlap makes it harder to obtain clear insights into the comparative evaluation of AST versus code inputs. Our findings do not prevent the possibility that AST representations remain valuable in specialized tasks or data-scarce settings, where explicit structural signals may still provide complementary benefits.

6 Related Work

6.1 AST Serialization Techniques

To enable sequence-based neural models to process ASTs, researchers commonly employ traversal methods to serialize tree structures into sequential representations.

Classic traversal strategies, such as preorder and postorder, generate serialized sequences by recording node types and visitation order, but typically do not support lossless reconstruction of the original AST [43]. To address the limitations of classic traversals, SBT [18, 19, 22] introduces bracket-style markers to enable unambiguous reconstruction of the original tree. Similarly, multi-sequence approaches, such as providing both root and leaf node sequences or combining preorder traversal sequences with parent node sequences [26, 32], also support reconstructing the original AST. However, these methods typically result in much longer and more redundant sequences, which increases computational cost. Besides, some studies employ path-based methods, such as Code2Seq and PathMiner [4, 21], which serialize ASTs as collections of root-to-leaf paths or subtrees. These methods also preserve the structural integrity of the AST. However, the number of path combinations is huge, which leads to input expansion while only sampling selection is difficult to cover the global structure.

While tree and graph-based neural models [24, 40] capture explicit hierarchy, sequence-based AST serializations still have the advantage in LLM pipelines due to their compatibility with transformer architectures and direct support for efficient, end-to-end processing.

6.2 LLMs for Code Summarization

Recent LLM-based code summarization studies can be grouped by their input:

(1) Some models use only the code sequence as input. For example, StarCoder [23], InCoder [13], and CodeLlama [33] generate summaries directly from tokenized source code, without adding structural information.
(2) To provide LLMs with richer context, thereby improving summary quality. Some other models augment the input with additional context or structure. These methods add information such as function signatures, comments, documentation, or file paths, and may include structural signals such as call or dependency graphs, data-flow information. Examples include PROCONSUL [27], which uses call graphs, and Ahmed et al. [2], which supplement few-shot prompts with repository path and data-flow details.

Research in the second category is more related to our work, as ASTs can be seen as structure-augmented input. This motivates us to examine whether serialized AST representations may achieve comparable or superior summarization quality over code input with LLMs in a fine-tuning context.

7 Conclusion and Future Work

This paper investigates whether serialized ASTs can achieve comparable or superior method-level summarization quality to code sequences. To this end, we propose AST(NIT), an AST augmentation and serialization technique that preserves lexical details and serializes the tree as a compact sequence via node-index traversal. Using the *LLaMA-3.1-8B*, we systematically compare AST(NIT) with two established AST serializations (SBT and preorder) as well as with code sequences, evaluating both summary quality and efficiency. Our results show that serialized ASTs can achieve summary quality comparable to code sequences; moreover, compared to SBT, AST(NIT) significantly reduces average input length and training time.

Future work will examine whether these findings generalize across other programming languages, datasets, and LLM variants. We also plan to explore the applicability of AST(NIT) to other software engineering tasks where structural information may be more directly relevant, such as semantic code clustering and software modularization. In addition, human evaluation may be considered to complement automatic metrics.

References

[1] 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*. 1–5.
[2] 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*. 1–13.
[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).
[5] 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*. 65–72.
[6] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 253–264.
[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings.*

International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, 368–377.

[8] Tom Beckmann, Jan Reppien, Jens Lincke, and Robert Hirschfeld. 2024. Supporting Construction of Domain-Specific Representations in Textual Source Code. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. 17–28.

[9] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence* 5, 3 (2023), 220–235.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[11] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 13–22.

[12] Chunrong Fang, Weisong Sun, Yuchen Chen, Xiao Chen, Zhao Wei, Quanjun Zhang, Yudu You, Bin Luo, Yang Liu, and Zhenyu Chen. 2024. Esale: Enhancing code-summary alignment learning for source code summarization. *IEEE Transactions on Software Engineering* 50, 8 (2024), 2077–2095.

[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[15] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 223–226.

[16] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[17] Fan Hu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Xirong Li. 2022. Tackling long code search with splitting, encoding, and aggregating. *arXiv preprint arXiv:2208.11271* (2022).

[18] 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*. 200–210.

[19] 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 (2020), 2179–2217.

[20] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2073–2083.

[21] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. Pathminer: a library for mining of path-based representations of code. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 13–17.

[22] 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 (ICSE)*. IEEE, 795–806.

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[24] 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 (ICPC)*. IEEE, 184–195.

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

[26] Junhao Lin and Lu Lu. 2021. Semantic feature learning via dual sequences for defect prediction. *IEEE Access* 9 (2021), 13112–13124.

[27] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. 2024. Proconsul: Project context for code summarization with llms. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 866–880.

[28] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[30] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International conference on program comprehension (ICPC)*. IEEE, 23–32.

[31] 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*. 311–318.

[32] Shaoming Qiu, E Bicong, Xinchen Huang, and Liangyu Liu. 2024. Software Defect Prediction Based on Double Traversal AST. In *2024 8th Asian Conference on Artificial Intelligence Technology (ACAIT)*. IEEE, 1665–1674.

[33] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[34] Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for source code summarization. *Automated Software Engineering* 31, 1 (2024), 22.

[35] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, et al. 2024. An extractive-and-abstractive framework for source code summarization. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–39.

[36] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959* (2024).

[37] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. 2022. Ast-trans: Code summarization with efficient tree-structured attention. In *Proceedings of the 44th International Conference on Software Engineering*. 150–162.

[38] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[40] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 397–407.

[41] 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* (2021).

[42] Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. 2023. Cp-bcs: Binary code summarization guided by control flow graph and pseudo code. *arXiv preprint arXiv:2310.16853* (2023).

[43] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.

[44] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[45] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[46] Xuejun Zhang, Xia Hou, Xiuming Qiao, and Wenfeng Song. 2024. A review of automatic source code summarization. *Empirical Software Engineering* 29, 6 (2024), 162.

[47] Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352* (2019).