The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget

Dangfeng Pan
Monash University
Australia
dpan0026@student.monash.edu

Zhensu Sun
Singapore Management University
Singapore
zssun@smu.edu.sg

Cenyuan Zhang
Monash University
Australia
cenyuan.zhang@monash.edu

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Xiaoning Du
Monash University
Australia
xiaoning.du@monash.edu

ABSTRACT

Source code is usually formatted with elements like indentation and newlines to improve readability for human developers. However, these visual aids do not seem to be beneficial for large language models (LLMs) in the same way since the code is processed as a linear sequence of tokens. Furthermore, these additional tokens can lead to increased computational costs and longer response times for LLMs. If such formatting elements are non-essential to LLMs, we can reduce such costs by removing them from the code. To figure out the role played by formatting elements, we conduct a comprehensive empirical study to evaluate the impact of code formatting on LLM performance and efficiency. Through large-scale experiments on Fill-in-the-Middle Code Completion tasks across four programming languages (Java, Python, C++, C#) and ten LLMs—including both commercial and open-source models—we systematically analyze token count and performance when formatting elements are removed. Key findings indicate that LLMs can maintain performance across formatted code and unformatted code, achieving an average input token reduction of 24.5% with negligible output token reductions. This makes code format removal a practical optimization strategy for improving LLM efficiency. Further exploration reveals that both prompting and fine-tuning LLMs can lead to significant reductions (up to 36.1%) in output code length without compromising correctness. To facilitate practical applications, we develop a bidirectional code transformation tool for format processing, which can be seamlessly integrated into existing LLM inference workflows, ensuring both human readability and LLM efficiency.

1 INTRODUCTION

Large Language Models (LLMs) have revolutionized software development through their remarkable capabilities in code understanding and code generation. When tasked with code generation or completion, LLMs can interpret a developer's intent from incomplete code snippets or natural language descriptions, producing code suggestions that align closely with the developer's expectations. Recently, advanced models such as GPT-4o and Gemini-1.5 have shown performance levels comparable to those of human programmers across a range of programming tasks and languages [22]. LLMs focus on the next-token-prediction task during pretraining, where they learn from vast and diverse textual corpora and become generalizable to various tasks and domains [32]. On the other hand, this paradigm imposes restrictions on the representation of content, particularly for information dimensions that cannot be fully or effectively captured in a linear fashion [6].

Code differs from natural language due to its structured nature. Since code can be quite complex, adhering to appropriate coding styles is crucial for ensuring readability. Specific conventions are usually developed for programming in different languages, defining coding rules that have been shown to be most beneficial for understanding and maintaining the code. Recommended code styles visually enhance the identification of the code structure without affecting the code semantics, thereby aiding in the comprehension of its meaning [36]. Formatting elements such as indentation, whitespace, and newlines are commonly used to achieve this clarity. However, these formatting practices can increase the overall length of the code, leading to more tokens being processed by language model tokenizers. The visual advantages of well-formatted code can be lost in a linear representation of tokens, rendering the effort to use formatting somewhat ineffective. Additionally, the extra tokens consume a significant portion of the token budget in LLM-based code generation.

To have a quantitative understanding of the contribution of formatting elements in token count, we carried out a preliminary study. We randomly sampled 100,000 source files for each programming language (Java, C#, and C++) from the Stack v2 [8] code datasets, and measured their token counts using GPT-4o's tokenizer. As a comparison, we also measure the token count when the three types of formatting elements - indentation, whitespace, and newline - are removed. Notably, we only remove the ones that do not affect the code semantics, i.e., the AST before and after the removal remains the same. We demonstrate the effect of format removal for a Java class definition in Figure 1. We can observe that the token count is reduced from 60 to 47, indicating the formats incur 13 tokens' redundancy in the representation. With the same measurement, we observe a surprisingly non-marginal token reduction for all three languages, with 14.7% for Java, 13.2% for C#, and 13.2% for C++. Since LLMs operate on a token-by-token basis, the number of tokens to be processed or generated directly impacts their inference efficiency, where most commercial LLM APIs charge based on token count for both input and output [2, 17]. Given this non-trivial overhead brought by formatting elements, it is critical to have a clear understanding of the role they play in code representations when being processed by LLMs.

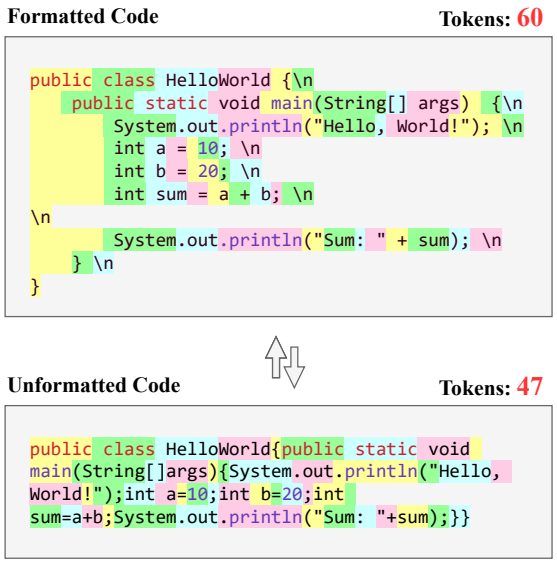

Figure 1: A comparison between formatted and unformatted Java code snippet, tokenized by GPT-4o's tokenizer. Continuous characters with the same background color represent the same token. The unformatted code is produced by removing indentation, whitespaces, and newlines while remaining syntactic correctness.

To the best of our knowledge, there is still a limited understanding of this topic. Different studies have reported differing findings: some indicate that LLMs pay less attention to formatting elements [41], while others suggest that seemingly insignificant tokens may help LLMs make more informed decisions [20]. In a recent paper [36], the authors introduce a simplified AI-oriented Python grammar that removes both formatting elements and unnecessary grammar tokens. However, it does not analyze the impact of formatting elements and requires further pre-training to adapt to the new grammar, leaving the effects of formatting elements on already-trained LLMs unclear. To address this question more effectively, a large-scale comprehensive experimental evaluation is necessary. Instead of relying on attention measurements, a more convincing approach would be to directly evaluate how model performance changes when formatting elements are either included or removed.

To fill this knowledge gap, we conduct a comprehensive empirical study to understand the role code formats play with respect to LLM's code generation capability. We employ a specific code completion task, the Fill-in-the-Middle (FIM), which shows the model a piece of incomplete code with missing middle sections and instructs it to generate the completion. It is a common task for nearly all coding assistants in IDEs and can assess both the code understanding and generation capabilities of the LLM. Our study will focus on four widely-used programming languages: Java, Python, C++, and C#, all of which are commonly mastered by most LLMs. We identify the formatting elements that can be omitted for each language by examining their lexer configurations and grammar rules. We select elements categorized as non-essential or skippable, filter out those related to comments or other non-code components, and retain only the relevant elements for evaluation. Finally, we target three formatting elements, i.e., indentation, whitespace, and newline. It is important to note that our goal is to remove formatting elements that do not contribute any semantic meaning, ensuring that the code functionality conveyed by both the formatted and unformatted code remains unchanged. The code before and after the removal is respectively named as **Formatted Code** and **Unformatted Code**. The study includes ten models: five commercial API-based models and five open-weight models. We utilize McEval[4], a multilingual dataset for FIM tasks, which encompasses all four programming languages being studied and provides test cases for evaluation. Next, we introduce the three RQs and summarize essential findings.

RQ1: Can LLM maintain their performance when handling unformatted code, and how does code formatting impact their efficiency? In this RQ, we explore how well LLMs can maintain their performance when formatting elements are removed from the prompts of FIM tasks. Additionally, we are interested in whether LLMs can adhere to the input code's formatting style when generating answers. If they can, it suggests that LLMs tend to maintain a consistent coding style during operations, which could be utilized to create more efficient LLMs.
Findings. On average, across all settings, the performance of evaluated models remains largely unaffected by the removal of formatting. For example, DeepSeek-V3 shows minimal variation in Pass@1 scores across all programming languages, with an average of 79.1% for formatted code and 80.0% for unformatted code. Regarding inference efficiency, removing formatting elements significantly reduces the number of input code tokens by 24.6%, while the decrease in output code tokens is much smaller at only 2.9%. This indicates that the model tends to generate code in a familiar formatting style, regardless of that in the input. In summary, removing formatting from the input has a minimal impact on the performance of LLMs and improves inference efficiency. However, the efficiency gain could be enhanced if the models become more adaptable to different styles.

RQ2: What is the impact of each formatting element on model performance and efficiency? Although removing all formatting elements has minimal performance impact, further ablation studies are necessary to determine if significant variance exists between the impact of different formatting elements. In this RQ, through an ablation study, we assess the influence of removing one type of formatting element at a time on token consumption and model performance.
Findings. Unlike the removal of all formatting elements, removing individual formatting elements can introduce negative impacts for some LLMs. Specifically, Claude and GPT-4 exhibit strong robustness, with performance variations (less than 1.6%) remaining minimal, consistent with the findings when all formatting elements are removed. In contrast, Gemini shows significant performance drops when any single formatting element is removed, highlighting its sensitivity to partially formatted styles. Additionally, while removing single formatting elements can effectively reduce input

tokens, the reduction rate of output tokens remains low, similar to the case of completely unformatted code. On average, output tokens decrease by only 0.4% for Claude, 3.5% for Gemini, and 1.4% for GPT-4 when individual formatting elements are removed. These limited reductions in output tokens, coupled with the performance degradation observed in Gemini, underscore the importance of adapting LLMs to different formatting styles.

RQ3: How to enable LLMs to minimize token usage when generating outputs? In RQ1 and RQ2, we demonstrated that LLMs benefit from unformatted code input. However, these models still prefer generating formatted code as output, regardless of the input format, leading to unnecessary token usage for formatting. In RQ3, we explore adapting models for producing token-efficient code in the output, focusing on two cost-effective approaches: training-free prompting and fine-tuning (on very few samples).
Findings. Prompting LLMs with clear instructions to generate unformatted output code can effectively reduce token usage while maintaining performance. For instance, with a well-crafted prompt, GPT-4o achieves a significant reduction in output tokens (an average of 27.2%) while maintaining performance in Java, C++, and C#. However, prompting can fail if the instructions are ambiguous or misinterpreted by the LLM. For example, Gemini often removes elements in a way that violates syntax rules, leading to a sharp decline in Pass@1 (e.g., from 67.2% to 11.1% in C++). Similarly, fine-tuning with unformatted samples can also reduce output tokens while preserving or even improving Pass@1. By fine-tuning with just 50 unformatted Java code samples, Gemini and GPT-4o achieve substantial reductions in output tokens (35.9% and 24.8%, respectively) with statistically insignificant performance impact. Both methods are feasible, and the choice between prompt engineering and fine-tuning depends on the LLM's use case and the user's role.

These empirical results strongly suggest that code format can be and should be removed when LLMs work with source code. To this end, we propose a code transformation tool that enables bidirectional conversion between formatted and unformatted code, preserving program semantics while reducing token overhead. As shown in Figure 2, this tool can transforms human-formatted code into an unformatted, token-efficient representation for LLM processing, and then reformats the LLM-generated output into a human-readable format. It enables developers to work with familiar, well-formatted code while LLMs work with token-efficient, unformatted code. The tool currently supports four programming languages and has been well tested through rigorous AST equivalence verification across the entire McEval dataset. It demonstrates an average transformation speed of 76ms per code sample, ensuring both semantic preservation and efficient real-time processing.

The tool code and live demo are available at https://sites.google.com/view/the-hidden-cost-of-readability. Our work makes several key contributions:

- We reveal that using unformatted code can significantly reduce token usage without compromising the performance of LLMs, offering a practical optimization strategy for efficient LLM serving.
- We demonstrate that LLMs can be trained or instructed to produce token-efficient code, further enhancing their efficiency in code generation tasks.
- We propose and implement a code transformation tool that facilitates bidirectional conversion between formatted and unformatted code, allowing developers to work with human-readable code while enabling LLMs to process code in a token-efficient manner.

2 EXPERIMENTAL SETTING

In this section, we introduce the experimental setting for this study, including the task, benchmark, evaluation metrics, LLMs, format processing methods, and implementation details. Our study is guided by three research questions:

- **RQ1:** Can LLMs maintain their performance when handling unformatted code, and how does code formatting impact their efficiency?
- **RQ2:** What is the impact of each formatting element on mode performance and efficiency?
- **RQ3:** How to enable LLMs to minimize token usage when generating outputs?

2.1 Task and Benchmark

We choose the **Fill-in-the-Middle** (FIM) code completion task [13], the most prevalent and practical code completion paradigm used by commercial AI-powered programming assistants, to investigate the impact of code format [16]. The task requires the LLM to complete the code snippets with missing middle sections, which can assess the model's code understanding and generation capability at the same time. Given the significant costs associated with using commercial LLM APIs, our experiment includes four programming languages: Java, Python, C++, and C#, which feature diverse formatting conventions and are among the most popular languages in production environments. Catering to these needs, we select **McEval**[4] as the benchmark for our study. It is created by professional developers through a rigorous annotation and verification process, covering 40 programming languages. By having experienced developers manually create and validate each sample, the dataset is independently generated rather than being automatically translated from Python, which enhances diversity and authenticity while minimizing the risk of data leakage from training sets. In this benchmark, each sample consists of a code snippet with a missing middle section for the FIM task, a detailed problem description that explains the coding challenge, and a set of corresponding test cases to evaluate the correctness of the generated code. We use its subset for the four languages, which offers 314 samples for C++, 318 for C#, 355 for Java, and 330 for Python.

2.2 Large Language Models

To ensure the generalization of our findings, we evaluate a diverse range of state-of-the-art LLMs, including five commercial API-based models and five open-weight models. Among the commercial models, we include three models from OpenAI: **GPT-3.5-turbo**, a widely used efficient model that balances performance and cost-effectiveness in production applications; **GPT-4o-mini**, a mid-sized model that strikes a balance between computational efficiency and robust capabilities; and **GPT-4o**, an advanced multimodal model representing the state-of-the-art in commercial LLM performance [14, 30]. Additionally, we evaluate Google's **Gemini**,

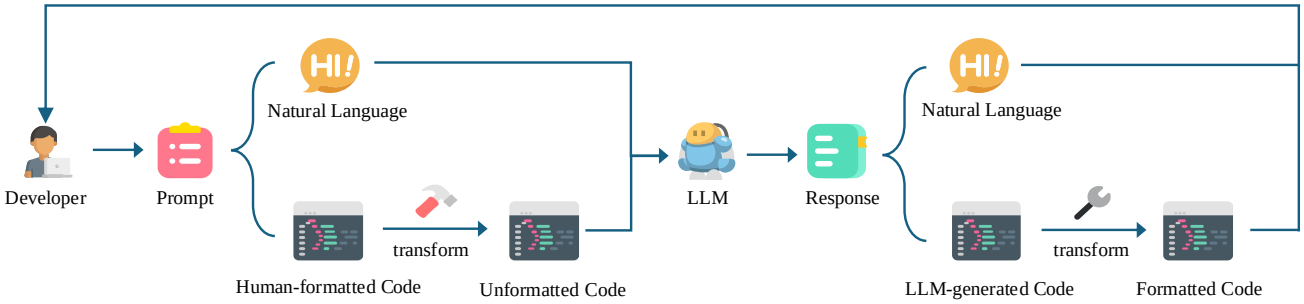

Figure 2: A demonstration of how unformatted code can fit in the existing Human-AI workflow.

specifically using the gemini-1.5-flash version, which is a fast and versatile multimodal model for scaling across diverse tasks. [11], and Anthropic's **Claude 3.7 Sonnet**, which emphasizes safety and alignment [1].

On the open-weight front, the five models are **Phi** (with 3.82B parameters), a lightweight instruction-tuned model pre-trained on synthetic data and filtered web content for reasoning and instruction-following tasks [12]; **Qwen** (1.54B parameters), Alibaba's specialized instruction-tuned model optimized for code generation and reasoning [9]; **Magicoder**, an instruction-tuned system (6.7B parameters) fine-tuned from Deepseek-coder-6.7B using the OSS-Instruct method to enhance code generation quality [40]; and two models from DeepSeek: **DeepSeek-V3**, a powerful Mixture-of-Experts model with 671B total parameters (37B activated per token) pre-trained on 14.8 trillion tokens and fine-tuned via supervised and reinforcement learning [10], and **Deepseek-coder-1.3B**, trained from scratch on 2 trillion tokens with a composition of 87% code and 13% natural language [21].

2.3 Evaluation Metrics

We evaluate our approach using the following metrics:

- **Pass@1:** To compute Pass@1, one code sample is generated for each problem in the benchmark, and a problem is considered solved if the samples pass the unit tests. We report the fraction of problems being successfully solved.
- **Token Counts:** We assess efficiency by counting the tokens in the input code and the generated output. Notably, since tokens are processed using each model's tokenizer, the same text may yield different token counts across different LLMs. For input token reduction, we extract the code portion from the prompt and calculate the difference in token counts between the formatted and unformatted versions. For output token reduction, directly comparing outputs generated from formatted and unformatted inputs is unfair, as differences in input may lead to semantically different outputs. Instead, we estimate output token reduction by reformatting the LLM-generated code for unformatted input into the most concisely formatted version and comparing the token counts before and after reformatting. This approach quantifies the potential token savings in the output. Notably, token counts can be directly translated to the financial cost when using commercial LLM APIs. For example, GPT-4o charges $2.5 per 1M tokens for input and $10.00 per 1M tokens for output [29].
- **Statistical Validation:** We employ McNemar's test [26], a non-parametric method particularly suited for comparing the performance of two models on the same test instances. This test helps us determine whether differences in Pass@1 between the experimental and control groups are statistically significant (p-value < 0.05). Additionally, for analyzing the statistical significance of differences in input and output token counts between these experimental and control groups, we utilize the Mann-Whitney U Test [24]. This non-parametric test is appropriate for comparing distributions of token counts as it does not assume normality in the data, allowing us to robustly assess whether the observed token reductions are statistically significant. To address the multiple testing problem across different RQs and measurements, we apply False Discovery Rate (FDR) correction using the Benjamini-Hochberg procedure [3], which controls the expected proportion of false positives and strengthens the validity of our statistical claims.

2.4 Format Processing

We clarify the formatting elements considered in our study and clearly define formatted code and unformatted code.

- **Formatting elements.** To identify omittable formatting elements for evaluation, we analyze the lexer configurations and grammar rules of each programming language, selecting elements categorized as non-essential or skippable. After excluding elements related to comments or other non-code components, finally three formatting elements are selected for evaluation: indentation, whitespace, and newlines. In Java, C#, and C++, indentation, newlines, and additional whitespace are formatting tokens that do not affect the semantic meaning of the code, except in the case of preprocessor directives in C# and C++. For Python, only whitespace is removed as the other two elements are required by Python syntax.
- **Formatted Code (Control Group):** To establish uniform control groups, we standardize the usage of the formatting elements mentioned above in all code samples according to the Google Style guidelines [18] for C-family languages (C++, C#, Java) and PEP 8 [38] for Python. This ensures that our formatting accurately reflects real-world practices. This standardization results in a slight increase of 1.15% in the token count of the evaluation datasets. While this may affect the quantitative results of our

experiments, it allows for more informed and transparent conclusions. We believe that the qualitative findings remain valid despite these slight changes in formatting rules.

- **Unformatted Code (Experimental Group):** For all code samples, we minimize whitespaces, newlines, and indentation without violating the syntax rules of each programming language. Regarding whitespaces, we eliminate non-essential spaces while keeping those required by the language's syntax (e.g., spaces between keywords and identifiers). For newlines, we remove blank lines and combine multiple statements onto single lines wherever syntactically permissible. We preserve necessary newlines, such as those separating preprocessor directives in C++. As for indentation, we remove all leading spaces or tabs at the beginning of lines, as long as the syntax allows.

2.5 Implementation Details

In our experiments, we evaluate a diverse range of popular LLMs, including commercial API-based models and open-weight models. The commercial API-based models include GPT-3.5-turbo, GPT-4o-mini, GPT-4o from OpenAI, Gemini from Google, Claude from Anthropic, and DeepSeek-V3 from Novita (a third-party API provider). For API invocation, we implement standardized REST requests to each provider's endpoints using their official SDKs with default parameters. Additionally, we evaluate open-weight models locally, including Phi, Qwen, Deepseek-coder, and Magicoder. The local models are implemented using the Huggingface Transformers library with PyTorch and executed on hardware equipped with 40 vCPUs, 480GB RAM, and an NVIDIA GeForce RTX 3090 GPU (24GB VRAM). For these local models, the maximum token limit is set to 2048 due to GPU memory constraints. During inference, we use default hyper-parameters for all models, setting the temperature to 0 to ensure deterministic outputs. For fine-tuning experiments, we employed two distinct approaches. For parameter-efficient fine-tuning (PEFT), we randomly sampled 5000 examples for each programming language from the code-instruct-700k [34] dataset to train smaller models. For API-based fine-tuning, we selected a slimmer dataset of 50 examples from the McEval benchmark for each language to fine-tune commercial models, balancing cost considerations with effective adaptation.

3 STUDY RESULTS

In this section, we report our experimental results and answer the three research questions.

3.1 RQ1: Impact of Unformatted Code on LLM Performance and Efficiency

In this RQ, we systematically evaluate the impact of removing formatting elements from the prompts of FIM tasks. In addition to LLM performance, we also examine the LLMs' adherence to the formatting style of the prompts. The ability to maintain a consistent coding style during generation could further enhance LLMs' token efficiency.

To systematically evaluate the impact of code formatting on LLM performance and token efficiency, we evaluate all ten models using the McEval benchmark of the four programming languages. We prompt the models with two versions of the incomplete code snippet, namely, the formatted version and the unformatted version. We calculate Pass@1 using test cases and measure token counts for both the incomplete code and the generated completions, following the methodology outlined in Section 2.3. Additionally, we assess the statistical significance between the Pass@1 scores and token counts of the experimental group (using unformatted code) and the control group (using formatted code).

The results are presented in Table 1. We bold case where the performance on unformatted input code is either equal to or only marginally lower (within a 2% margin) than the performance on formatted code.

3.1.1 Impacts on LLM performance. Most models exhibit stable performance across formatted and unformatted code inputs, with only minor fluctuations observed in certain languages. Specifically, across all models and languages, we do not observe a Pass@1 drop higher than 4.2% after removing the code format in the input, and all those drops exhibit insignificant p-values. State-of-the-art (SOTA) models, such as Deepseek-V3, Gemini, and GPT-4o, demonstrate negligible differences in Pass@1 scores between formatted and unformatted code, indicating that removing formatting has no significant impact on their performance. Smaller or less performant models exhibit more pronounced fluctuations. For example, Deepseek-Coder shows performance degradation in C++ (3.2% drop in Pass@1) and C# (2.9% drop), while Phi struggles with C++ (4.2% drop). This may be caused by the inherent limitations of smaller models in generalizing across diverse input structures. In contrast, SOTA models, with their larger parameter counts and more advanced training methodologies, are better equipped to handle unformatted inputs. Interestingly, we also observe cases where performance improves after removing the format, some of which are even statistically significant. For instance, in Python, GPT-4o's performance increases from 66.4% to 71.5% when transitioning from formatted to unformatted code. We infer that the removal of formatting elements simplifies the input in certain cases, reducing distractions from non-essential tokens and allowing models to focus on the core logic. Investigating the reasons behind this phenomenon could be an interesting direction for future work.

> **Takeaway#1:** The code format in the input does not negatively impact the performance of LLMs, as they demonstrate comparable Pass@1 scores across both formatted and unformatted inputs.

Among the evaluated languages, Java stands out as the most stable in terms of model performance when transitioning from formatted to unformatted code inputs. This stability is evident across all models, with no significant performance drops observed even for smaller or less performant models. This exceptional stability can likely be attributed to the prevalence of unformatted Java code in their pre-training datasets. Java is one of the most widely used programming languages, and its code is frequently shared in diverse formats in open-source repositories, forums, and documentation. As a result, models are likely exposed to a significant amount of unformatted or minimally formatted Java code during training, enabling them to better generalize such inputs. In contrast, other

Table 1: Comparison of model performance (Pass@1) and token efficiency between formatted code (F) and unformatted code (U). Models include DeepSeek-V3 (DS-V3), Claude, Gemini, MagicCoder (MC), GPT-4o, GPT-4o-mini (GPT-4o-m), DeepSeek-Coder (DS-C), Qwen, GPT-3.5, and Phi.

Language	Metric	DS-V3	Claude	Gemini	MC	GPT-4o	GPT-4o-m	DS-C	Qwen	GPT-3.5	Phi
C++	Pass@1 (F)	**76.1%**	**72.9%**	**68.5%**	**65.9%**	**63.7%**	**62.4%**	49.0%	**39.2%***	**31.5%***	16.9%
	Pass@1 (U)	**76.8%**	**72.9%**	**67.2%**	**64.3%**	**61.8%**	**62.4%**	46.8%	**46.8%***	**38.2%***	12.7%
	Input Reduction	30.8%**	28.9%**	34.0%**	24.8%**	33.7%**	33.7%**	24.8%**	34.4%**	35.1%**	31.0%**
	Output Reduction	0.0%	-0.7%	9.3%	2.9%	0.4%	0.7%	2.6%	1.9%	2.8%	1.3%
C#	Pass@1 (F)	**87.7%**	90.3%	**77.4%**	**73.6%**	**76.7%**	**78.3%**	51.6%	**45.6%**	**47.8%**	**15.4%**
	Pass@1 (U)	**86.8%**	87.7%	**76.4%**	**73.6%**	**77.7%**	**77.7%**	48.7%	**49.1%**	**50.6%**	**15.7%**
	Input Reduction	22.3%**	22.7%**	29.7%**	22.7%**	26.2%**	26.2%**	22.7%**	26.8%**	27.1%**	26.2%**
	Output Reduction	-0.5%	-2.3%	2.8%	0.7%	-0.5%	-0.4%	0.7%	1.2%	0.5%	0.4%
Java	Pass@1 (F)	**70.7%**	**68.5%**	**67.9%**	**63.7%**	**63.1%**	**65.1%**	**56.1%**	**47.9%**	**36.9%***	**15.8%**
	Pass@1 (U)	**71.3%**	**69.0%**	**67.9%**	**63.7%**	**65.6%**	**66.5%**	**58.9%**	**51.3%**	**43.4%***	**17.7%**
	Input Reduction	29.9%**	35.7%**	42.0%**	33.1%**	33.9%**	33.9%**	33.1%**	35.0%**	35.1%**	37.2%**
	Output Reduction	3.7%	5.0%	11.5%	6.8%	4.6%	4.1%	7.2%	7.3%	4.6%	6.2%
Python	Pass@1 (F)	**81.8%**	**84.5%**	**71.2%**	**44.2%**	**66.4%***	**52.1%**	**50.9%**	66.7%	73.6%	**20.9%**
	Pass@1 (U)	**85.2%**	**87.0%**	**71.8%**	**44.8%**	**71.5%***	**51.8%**	**49.7%**	63.9%	70.3%	**20.3%**
	Input Reduction	7.4%	4.7%	5.6%	2.2%	9.4%	9.4%	2.2%	9.5%	9.5%	5.2%
	Output Reduction	0.0%	-1.1%	4.1%	1.9%	0.4%	0.5%	1.4%	4.3%	2.3%	0.1%

Bold: Unformatted code performance is either better than or within 2% of formatted code. * : p-value < 0.05 ** : p-value < 0.01

languages exhibit greater sensitivity to format changes. For example, both Qwen and GPT-3.5 show noticeable performance drops in Python (2.8% and 3.3% decrease in Pass@1, respectively), highlighting the influence of language-specific syntax and training data distribution on model performance.

Takeaway#2: Models exhibit language-specific sensitivity to code formatting changes, where Java demonstrates exceptional stability across formatted and unformatted inputs.

3.1.2 Impacts on Token Reduction. Using unformatted code as input can reduce a considerable number of tokens across all evaluated models, where the extent of this reduction varies due to differences in tokenizers. On average, the input code tokens are reduced by 25.8% across all settings. For Deepseek-V3, the input code token reduction is particularly notable, with Java code tokens reduced by 42.0%, highlighting the potential of format removal in accelerating the models' code understanding speed. Similarly, commercial LLMs like Claude, Gemini, and GPT-4o also demonstrate substantial input token reduction, with averages of 23.0%, 27.8%, and 25.8%, respectively. For these commercial models, these reductions directly translate to cost savings in their API services. Take Claude 3.7 Sonnet for example. It charges $3 per 1M input tokens, where a 23.0% token reduction can save $0.69 per 1M tokens [2]. This cost efficiency makes format removal an attractive option for users of these models. Moreover, the efficiency gains vary substantially depending on the programming language. Java shows the highest efficiency gains, averaging 34.9% token reduction, due to its verbose syntax and reliance on formatting for readability. The efficiency gains are also substantial for C++ and C#, respectively 31.12% and 25.26%. In contrast, Python achieves only a 6.51% average code token reduction, as formatting elements such as newlines and indentations are integral to its grammar and cannot be removed without compromising code functionality. This language-specific variability must be carefully considered when deciding whether to use unformatted code for specific LLM-driven applications.

Takeaway#3: Removing code formatting can substantially reduce input tokens for languages that do not deeply integrate formatting elements into their syntax (e.g., Java, C++, C#) and only slightly reduce tokens for languages like Python, where formatting is essential to syntax and functionality.

While input token reduction is substantial, output token reduction remains modest, depending on the language or models. Specifically, output code token reduction averages 2.5% across all models and languages, with the highest reduction being 11.5%, achieved by Gemini in Java. This suggests that models have internalized formatting conventions and maintain them in outputs regardless of input formatting. It underscores the need for further optimization in output generation to fully capitalize on the efficiency gains achieved through input token reduction. We also observe some instances showing negative reduction. Upon examining the generated code, we find that this is because the code generated by the models sometimes misaligns with the style guideline we use. As a result, when we convert the generated code into the style guideline format, additional tokens are introduced, leading to negative reductions in some cases. LLM may produce a shorter response if the generated code is of low quality.

Takeaway#4: LLMs tend to maintain formatting conventions in their outputs, regardless of input formatting.

3.2 RQ2: Contribution of Individual Formatting Elements

While the removal of overall formatting had minimal performance impact, the mixed results—ranging from performance drops to improvements—prompted us to figure out the impact of each individual formatting element. To this end, we conduct an ablation study on the three top-performing commercial LLMs (Claude, Gemini, and GPT-4o) from our prior experiments. The ablation study involves removing one type of formatting element at a time—indentation, whitespaces, or newlines—and observing the resulting performance of the LLMs. This study focuses on C++, Java, and C#, excluding Python, as its syntax only allows for whitespace removal, leaving no room for ablation. Specifically, for each sample in the benchmark, we generate three ablated versions: Whitespaces Removed, Indentations Removed, and Newlines Removed. Similar to the experiments in RQ1, we feed these versions to the LLMs and compute the Pass@1 and token reductions for both inputs and outputs. Additionally, we assess the statistical significance between the Pass@1 scores and token counts of the experimental group (one formatting element removed) and the control group (all formatting elements removed). The results are presented in Table 2.

3.2.1 Impacts on LLM performance. Across the three commercial LLMs evaluated, we observe that the impact of removing individual formatting elements differs across models. Claude and GPT-4o demonstrate remarkable stability in performance when individual formatting elements are removed, where all p-values are all higher than 0.05, indicating no significant differences. Specifically, across all languages, Claude's performance varies by less than 1% when whitespaces or indentations are removed and even improves by an average of 1.3% when only newlines are removed. Similarly, GPT-4o exhibits minimal performance fluctuations, with an average variation of 0.8% across all removal strategies, highlighting its robustness to formatting changes. However, Gemini shows greater sensitivity to formatting removal on a single type of formatting element, with some settings, such as whitespaces removed C# and newlines removed Java, showing statistically significant Pass@1 drops compared to removing all elements. Compared to fully unformatted code, its average Pass@1 across all three languages declines 4.0% with whitespace removal, 3.1% with indentation removal, and 2.9% with newline removal. This indicates that Gemini is more sensitive to the removal of individual formatting elements, particularly in C# and Java. These findings indicate that our prior observations—that code formatting does not negatively impact LLM performance—do not extend to the removal of individual formatting elements. We hypothesize that partially unformatted code is less common in training datasets, making it more challenging for some LLMs to process effectively. Interestingly, certain selective removal strategies yield better performance than both fully formatted and unformatted code for Claude and GPT-4o. For instance, when only newlines are removed in C++, Claude's Pass@1 increases 1.6% compared to the version with all elements removed. These results indicate that these LLMs may have specific preferences for certain formatting elements, which could be an interesting area for future research.

Takeaway#5: In terms of individual formatting elements, removing such an element may introduce a negative impact, as Gemini exhibits a significant performance degradation.

3.2.2 Impacts on Token Reduction. Our analysis reveals that different formatting elements contribute unequally to token consumption, with significant variations across both languages and models. For input code tokens, newlines contribute most significantly to token count for Claude and Gemini, accounting for an average of 14.6% for Claude and 17.5% for Gemini across languages. For GPT-4o, however, whitespaces have a higher impact with an average of 10.7% compared to newlines at 7.5%. Indentations represent the second largest contributor with an average of 7.9% for Claude, 8.9% for Gemini, and 9.6% for GPT-4o across all languages. Such differences are caused by the different tokenizers of each LLM. The input efficiency gains also vary by programming language. For instance, newline removal alone reduces Java tokens by 18.7% for Claude and 22.0% for Gemini. C++ follows with an average reduction of 9.2% across selective strategies, while C# shows 7.9%. Similar to Takeaway #3, output token reduction remains minimal across all models and formatting strategies. On average, output tokens decrease by only 0.4% for Claude, 3.8% for Gemini, and 1.3% for GPT-4o when individual formatting elements are removed.

Takeaway#6: Removing individual elements can also reduce input tokens by a considerable amount, but they still suffer from the same issue in output token efficiency as completely unformatted code.

3.3 RQ3: Methods for Token-Efficient Generation

As demonstrated in RQ1 and RQ2, even when the input code is unformatted, LLMs tend to generate formatted code in their outputs, resulting in insufficient code reduction. To explore whether and how this issue can be mitigated, we experiment with two techniques: prompt engineering and fine-tuning. The experiments were performed on two SOTA commercial LLMs, GPT-4 and Gemini, because they are the only commercial models, to the best of our knowledge, offering APIs for fine-tuning. In the following, we detail the experimental setup for each method and present their respective results.

3.3.1 Prompting with instructions to generate unformatted code. Since LLMs can interpret user instructions in prompts, a natural approach is to explicitly request unformatted code outputs. To explore this direction, we design two distinct instructions: one concise (**P1:** "Output code without formatting, maintaining syntax.") and one detailed and explicit (**P2:** "Please directly output the following code, deleting all spacings, newlines, and indentations, provided that it does not violate any syntax rules:"). Each instruction is appended to the existing FIM task prompts. Using this revised setup, we evaluate the LLM-generated code completions against unformatted benchmark code in three languages, C++, C#, and Java, measuring pass@1 performance and quantifying reductions in input and output token counts. We also compute the statistical significance between the

Table 2: The Pass@1 and token reduction of three LLMs with various formatting configurations.

Model	Format	Pass@1			Token Reduction					
		C++	C#	Java	C++		C#		Java	
					Input	Output	Input	Output	Input	Output
Claude	Whitespaces Removed	72.3%	87.7%	69.0%	5.2%	0.4%	3.2%	-2.4%	3.3%	4.3%
	Indentations Removed	72.2%	88.7%	68.2%	6.7%	-1.4%	6.4%	-1.6%	10.5%	4.4%
	Newlines Removed	74.5%	89.0%	69.0%	13.4%*	-2.6%	11.7%*	-1.4%	18.7%**	4.3%
	All Removed	72.9%	87.7%	69.0%	28.9%	-0.7%	22.7%	-2.3%	35.7%	5.07%
Gemini	Whitespaces Removed	64.6%	71.3%*	63.7%*	6.3%	5.4%	3.4%	2.2%	3.4%	6.0%
	Indentations Removed	66.6%	71.1%*	65.1%	7.3%	2.5%	7.5%	0.5%	11.8%	6.8%
	Newlines Removed	66.9%	73.9%	62.8%**	15.4%**	1.9%	15.0%*	0.7%	22.0%**	7.8%
	All Removed	67.2%	76.4%	67.9%	34.0%	9.3%	29.7%	2.8%	42.0%	11.5%
GPT-4o	Whitespaces Removed	63.7%	79.5%	63.3%	14.2%	1.1%	8.4%	-0.2%	9.4%	4.0%
	Indentations Removed	63.7%	78.6%	63.3%	8.2%	0.1%	8.1%	-0.4%	12.5%	3.7%
	Newlines Removed	62.4%	79.2%	65.4%	7.9%	0.1%	6.1%	-0.8%	8.4%	3.8%
	All Removed	61.8%	77.7%	65.5%	33.7%	0.4%	26.2%	-0.5%	33.9%	4.6%

* : p-value < 0.05 ** : p-value < 0.01

Table 3: Comparison of Pass@1 and token reduction across three settings: unformatted input code with the original instruction (Origin) and two experimental prompts (P1 and P2).

Model	Language	Pass@1			Input Reduction	Output Reduction		
		Origin	Prompt(P1)	Prompt(P2)		Origin	Prompt(P1)	Prompt(P2)
Gemini	C++	67.2%	47.4%**	11.1%**	34.0%**	9.3%	25.4%**	29.2%**
	C#	76.4%	48.7%**	4.4%**	29.7%**	2.8%	21.0%**	25.1%**
	Java	67.9%	34.6%**	14.6%**	42.0%**	11.5%	37.2%**	38.9%**
	Python	71.8%	68.1%	10.9%**	5.6%	4.1%	5.5%	9.9%
GPT-4o	C++	61.8%	69.7%**	65.9%	33.7%**	0.4%	1.2%	32.6%**
	C#	77.7%	85.8%**	82.7%*	26.2%**	0.5%	0.1%	25.6%**
	Java	65.6%	68.5%*	66.5%	33.9%**	4.6%	5.4%	36.1%**
	Python	71.5%	80.9%**	59.7%**	9.4%	0.4%	0.4%	14.4%**

* : p-value < 0.05 ** : p-value < 0.01

pass@1 scores of the experimental group (using the revised prompt) and the control group (using the original prompt).

As shown in Table 3, prompting can be effective in addressing the insufficient output code token reduction. GPT-4o, instructed with P2, is observed to successfully achieve substantial output token reduction while maintaining its performance in Java, C++, and C#. Specifically, it reduces output code tokens by an average of 27.2% when processing unformatted code inputs with P2, which is comparable to its input reduction percentage. Meanwhile, its performance on these three languages is maintained or even significantly improved. It demonstrates the feasibility of using prompts for more output token reduction. However, in other settings, the LLMs cannot work well. For example, the concise prompt P1 is misunderstood by GPT-4o, resulting in output reductions similar to the original prompt, with the highest reduction being only 5.4% across all languages. Gemini experiences severe performance degradation under both P1 and P2, despite achieving satisfying output token reductions. Upon closer inspection, we find that Gemini tends to remove elements in a way that violates syntax rules. For example, when given a C# method declaration like `static bool HasCloseElements(List<double> numbers, double threshold)`, Gemini generates `staticboolHasCloseElements(List<double>numbers,doublethreshold)` where it incorrectly removes the spaces between keywords and types. This creates syntax errors as neither `staticbool` nor `doublethreshold` are valid constructs in C#. These failure cases highlight the importance of clear instructions for LLMs.

Takeaway#7: Prompting LLMs with well-crafted prompts to request unformatted output code can effectively reduce output code tokens while maintaining model's performance.

Notably, appending these instructions as new prompts introduced an overhead in input token count. Specifically, P1 and P2 added 8 and 28 tokens, respectively, to the input prompt, as measured by GPT-4o's tokenizer. This creates a break-even point where

Table 4: Comparison between models fine-tuned on formatted code (F) and unformatted code (U).

Model	Reduction		Pass@1	
	Input	Output	Finetuned(F)	Finetuned(U)
Gemini	44.2%**	35.9%**	64.1%	63.7%
GPT-4o	36.5%**	24.8%**	64.8%	67.4%

* : p-value < 0.05 ** : p-value < 0.01

the method becomes beneficial only when the expected output token reduction exceeds this initial cost. However, the length of the prompt not only affects overhead but also determines the space available for clearly describing instructions, presenting an efficiency-performance trade-off. For example, the short prompt P1, despite having less overhead, was misunderstood by GPT-4o due to its insufficient information.

Takeaway#8: Prompt design should balance the trade-off between the overhead from prompt length and the clarity of instructions.

3.3.2 Finetuning with unformatted samples. Fine-tuning is a widely used technique for adjusting model behavior. In this experiment, we fine-tune the two LLMs, GPT-4o and Gemini, using unformatted code samples. We also include a control group for measuring the impact of fine-tuning on model performance, i.e., the same model fine-tuned with formatted code samples in the same fine-tuning environment. Due to budget constraints, the fine-tuning is limited to Java. Specifically, from the McEval benchmark, we randomly select 50 Java samples as the training dataset and retain the remaining 305 samples as the test set. The training dataset is processed into two versions: formatted and unformatted. For each model, we train two variants, each using one version of the dataset. The training employs a parameter-efficient fine-tuning method, QLoRA [7]. We evaluate each fine-tuned model on the retained test set and compare their performance in Pass@1 and token reductions. We also compute the statistical significance between the pass@1 scores and token counts of the experimental group (fine-tuned with unformatted code samples) and the control group (fine-tuned with formatted code samples).

The results are reported in Table 4. Similar to prompt engineering, fine-tuning can also significantly increase the output token reduction. Specifically, Gemini achieves a substantial 35.9% reduction in output tokens with only a minimal performance impact and a 0.4% insignificant difference in Pass@1, while GPT-4o reduces output tokens by 24.8% and even shows a slight performance improvement (2.6%) when trained on unformatted code.

Takeaway#9: Fine-tuning with unformatted code can successfully reduce output tokens while maintaining or even improving Pass@1, offering a practical optimization strategy.

3.3.3 Prompting or Finetuning? In RQ3, we experiment with two approaches, prompt engineering and fine-tuning, to reduce more tokens in the output. The results demonstrate that both methods are viable for achieving this objective. Each approach has distinct strengths and limitations, which we discuss in detail below.

Prompt engineering guides the LLM in minimizing unnecessary formatting tokens by providing explicit natural language instructions. This method modifies only the input text without altering the underlying model, making it highly flexible and cost-effective, as it incurs no additional training expenses. However, our experiments reveal that the effectiveness of prompt engineering depends heavily on the quality of the prompt and the model's capabilities. Additionally, the prompt itself introduces token overhead, which can limit its practicality during prompt design.

In contrast, fine-tuning the model using unformatted code samples can also optimize token efficiency effectively. Our experiments show that lightweight fine-tuning with just 50 training samples, combined with PEFT techniques, achieves comparable token reduction. This approach avoids the overhead associated with prompting. However, fine-tuning requires access to the model, which may not be feasible for individual users of LLM services. Furthermore, fine-tuned models tend to exhibit fixed behavioral patterns based on the training data, reducing their adaptability to diverse needs.

In conclusion, the choice between prompt engineering and fine-tuning depends on the LLM's usage scenario and the user's role. For users seeking quick, flexible solutions without model access or additional training resources, prompt engineering offers a practical and cost-effective option. On the other hand, fine-tuning provides a more robust and consistent solution for token reduction, particularly for users with the resources and access to modify the model, as well as for domain-specific tasks like code completion.

Takeaway#10: The choice between prompt engineering and fine-tuning depends on the user's needs and resources. Prompt engineering is a flexible, cost-effective solution for users without model access, while fine-tuning offers a more robust and consistent approach for those with the resources to modify the model, particularly in stable tasks like code completion.

4 TOOL

Based on the promising results from our experiments, we developed a code transformation tool designed to either remove or restore formatting in source code. This tool can convert source code into a compact, unformatted version optimized for efficient model comprehension, as well as revert it back to a human-readable format.

4.1 Implementation

We implemented a transformation tool supporting C++, Java, C#, and Python. It is implemented to serve as the additional pre-processing and post-processing step to minimize the token consumption from the code fromat. The tool leverages language-specific formatters: for C-family languages (C++, Java, C#), we extended **Uncrustify** [5], a widely-used formatter. For Python, we implement a custom formatter built on **YAPF** [19], which accurately handles Python's indentation-based syntax. Currently, the tool supports the removal and restoration of three formatting elements: indentation, whitespaces, and newlines. It can be configured to remove some or all formatting elements to the greatest extent possible without violating syntax rules.

To handle partial code, which is often syntactically incorrect but common in code completions or user prompts, we developed a hybrid solution. The tool first separates the last unfinished block of code from the main code body and applies different strategies to remove or restore formatting in the split unfinished block and the remaining code. For the remaining code, we implemented a syntax repair component that uses a bracket-matching mechanism to identify and fix unbalanced brackets. This ensures the code is syntactically correct and can be processed accurately by our internal formatters. After processing, the added brackets used for formatting are removed. For the unfinished block, predefined regular expressions are applied to identify positions where formatting elements can be added or removed. Finally, the two blocks are concatenated back into a single unit.

4.2 Usage Scenarios

Our tool facilitates bidirectional transformation between human-readable, well-formatted code and LLM-friendly, compact code. As demonstrated in Figure 2, our tool can be used to build a dual-conversion inference workflow, enabling LLMs to benefit from the efficient compact code while still allowing human developers to work with familiar code. In this workflow, our tool removes the formatting elements in the input code, reducing token consumption for LLMs to understand. For the output, it restores the formatting in LLM-generated code to improve human readability without altering the underlying logic.

It can be used by both LLM service providers and their users. Providers such as OpenAI and Claude can leverage this tool to reduce computational overhead on their servers, leading to faster response times, lower resource utilization, and enhanced service efficiency and scalability. Furthermore, developers and organizations utilizing LLMs can integrate our tool as an additional pre-processing and post-processing step to minimize unnecessary token consumption. It can reduce their financial costs since most LLM APIs are charged based on token usage.

4.3 Performance Testing

We test our tool using the samples from the McEval dataset. Specifically, we compare the AST of each code sample before and after the transformation and achieve 100% AST equivalence across all test cases, confirming that our transformations preserve semantic correctness while only modifying formatting elements. We chose AST equivalence over text comparison since it guarantees program behavior equivalence, whereas textual comparison might be overly sensitive to non-functional formatting differences. During the test, our tool achieves an average transformation speed of 76ms per code sample, which is a negligible overhead. Such efficiency is crucial for real-time applications, like IDE plugins or API middleware, where transformation latency could otherwise impact user experience.

5 RELATED WORK

5.1 Program Simplification

Prior work on optimizing code representation for LLMs has explored various approaches. One common strategy involves training a specialized Byte-Pair Encoding tokenizer[35] on a code corpus, which can reduce token count compared to tokenizers trained on natural language corpora[39]. However, this approach still suffers from unnecessary formatting tokens, and once a model is trained, its tokenization strategy remains fixed, limiting adaptability. Some program simplification methods remove parts of input code based on auxiliary models[33, 37] or attention weights[42]. These methods inevitably compromise the semantic integrity of the code and are irreversible, restricting their applicability to code understanding tasks. Simpy[36] introduces the concept of AI-oriented grammar for compact code representation. While achieving substantial token reduction, this method modifies the syntax of the original programming language, requiring a specialized parser and model retraining to work with the new grammar. In this work, we propose a plug-and-play method for optimizing LLM token usage in code while preserving complete program semantics. Through empirical experiments, we reveal that SOTA models show resilience to the removal of formatting tokens. Building on this insight, we develop a code transformation tool that enables seamless conversion between human-readable and token-efficient representations, allowing LLMs to benefit from the token efficiency of unformatted code.

5.2 Coding Style with LLM

Coding style has been extensively studied in the field of software engineering. Oman et al.[28] established taxonomies that influenced code development guidelines and formatting tools. Building on these foundations, machine-learning based tools such as CODEBUFF [31] and STYLE-ANALYZER [25] have been developed to enforce consistent code formatting. Mi et al.[27] measured style inconsistencies within software project teams using clustering methods, while Wang et al. [15] examined style inconsistencies between LLM-generated code and human-written code. These researches focused on improving the style consistency of code. Beyond this, Hu et al.[23] investigated the impact of poor readability on LLMs, demonstrating that obfuscation techniques, such as modifying identifier names and injecting dead branches, can degrade model performance. However, their work did not explore the role of formatting elements. Unlike previous research, our work offers a unique perspective by quantifying the impact of formatting elements on computational cost and model performance across multiple languages and architectures. In addition, we analyze the contribution of individual formatting components, offering a deeper understanding of their influence on LLM processing.

6 THREATS TO VALIDITY

6.1 Generalization

Due to budget and hardware limitations, our experiments were restricted to four programming languages and focused solely on the Fill-in-the-Middle (FIM) task. As a result, our findings may not generalize to languages with distinct formatting styles or other LLM application scenarios. However, the selected languages are widely adopted in practice, and the FIM task is a common benchmark for nearly all coding assistants in IDEs. These choices provide a reasonable foundation for evaluating the model's performance in real-world settings, where enhancing token efficiency can lead to significant resource savings and is often a high priority.

6.2 Non-transparent Commercial LLMs

Our experiments involved invoking and fine-tuning commercial LLMs, such as GPT-4 and Gemini, through their closed-source APIs. This reliance on proprietary systems introduces potential limitations, as the internal mechanisms, training data, and fine-tuning strategies of these models are not transparent. Consequently, reproducibility and detailed analysis of their behavior are challenging, which may affect the generalizability of our findings. Future work could explore open-source alternatives or collaborate with API providers to gain deeper insights.

7 CONCLUSION AND FUTURE WORK

This paper exposes the hidden costs of code readability in LLM processing, demonstrating that formatting elements consume approximately 24.5% tokens across languages while providing minimal benefits for advanced models. Our analysis identifies the contributions of three kinds of formatting elements, whitespace, indentation, and newlines, and shows that both fine-tuning and prompting on unformatted code can further reduce token usage without compromising quality. These findings challenge conventional views of code formatting as purely human-oriented and reveal opportunities for substantial efficiency improvements in LLM-powered development workflows. Our bidirectional transformation tool offers a practical solution for balancing human readability with computational efficiency. In the future, we will investigate how formatting impacts more complex reasoning tasks beyond code completion.

REFERENCES

[1] Anthropic. 2024. Claude 3: Advanced AI Language Model. https://www.anthropic.com

[2] Anthropic. 2025. Anthropic API Pricing. https://www.anthropic.com/pricing#anthropic-api

[3] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)* 57, 1 (1995), 289–300.

[4] Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. 2024. McEval: Massively Multilingual Code Evaluation. *arXiv e-prints* (2024), arXiv–2406.

[5] Uncrustify Contributors. 2025. Uncrustify: Code beautifier. https://github.com/uncrustify/uncrustify

[6] Christine Cuskley, Rebecca Woods, and Molly Flaherty. 2024. The Limitations of Large Language Models for Understanding Human Language and Cognition. *Open Mind* 8 (08 2024), 1058–1083. https://doi.org/10.1162/opmi_a_00160 arXiv:https://direct.mit.edu/opmi/article-pdf/doi/10.1162/opmi_a_00160/2468254/opmi_a_00160.pdf

[7] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv preprint arXiv:2305.14314* (2023).

[8] Anton Lozhkov et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[9] An Yang et al. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[10] DeepSeek-AI et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[11] Gemini Team et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530 [cs.CL] https://arxiv.org/abs/2403.05530

[12] Marah Abdin et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[13] Mohammad Bavarian et al. 2022. Efficient Training of Language Models to Fill in the Middle. arXiv:2207.14255 [cs.CL] https://arxiv.org/abs/2207.14255

[14] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[15] Yanlin Wang et al. 2024. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. arXiv:2407.00456 [cs.SE] https://arxiv.org/abs/2407.00456

[16] GitHub. [n. d.]. How GitHub Copilot is getting better at understanding your code. *GitHub Blog* (October [n. d.]). https://github.blog/ai-and-ml/github-copilot/how-github-copilot-is-getting-better-at-understanding-your-code/

[17] Google. 2025. Gemini API Pricing. https://ai.google.dev/gemini-api/docs/pricing

[18] Google. 2025. Google Style Guide. https://google.github.io/styleguide/

[19] Google. 2025. YAPF: A formatter for Python files. https://github.com/google/yapf

[20] Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. 2023. Think before you speak: Training language models with pause tokens. *arXiv preprint arXiv:2310.02226* (2023).

[21] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[22] Wenpin Hou and Zhicheng Ji. 2024. Comparing Large Language Models and Human Programmers for Generating Programming Code. *Advanced Science* (30 12 2024). https://doi.org/10.1002/advs.202412279 [Online; accessed 2025-03-10].

[23] Chao et al. Hu. 2024. How Effectively Do Code Language Models Understand Poor-Readability Code?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 795–806. https://doi.org/10.1145/3691620.3695072

[24] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[25] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms. arXiv:1904.00935 [cs.LG] https://arxiv.org/abs/1904.00935

[26] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (June 1947), 153–157. https://doi.org/10.1007/bf02295996

[27] Qing Mi, Jacky Keung, and Yang Yu. 2016. Measuring the Stylistic Inconsistency in Software Projects using Hierarchical Agglomerative Clustering. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering* (Ciudad Real, Spain) *(PROMISE 2016)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/2972958.2972963

[28] Paul W. Oman and Curtis R. Cook. 1990. A taxonomy for programming style *(CSC '90)*. Association for Computing Machinery, New York, NY, USA, 244–250. https://doi.org/10.1145/100348.100385

[29] OpenAI. [n. d.]. *OpenAI API Pricing*. https://openai.com/api/pricing/

[30] Adam Lerer et al. OpenAI, Aaron Hurst. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[31] Terence Parr and Jurgen Vinju. 2016. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/2997364.2997383

[32] Mengnan Qi, Yufan Huang, Yongqiang Yao, Maoquan Wang, Bin Gu, and Neel Sundaresan. 2024. Is Next Token Prediction Sufficient for GPT? Exploration on Code Logic Comprehension. arXiv:2404.08885 [cs.PL] https://arxiv.org/abs/2404.08885

[33] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3520312.3534869

[34] Safurai. 2024. Code-Instruct-700k Dataset. https://huggingface.co/datasets/Safurai/Code-Instruct-700k

[35] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. arXiv:1508.07909 [cs.CL] https://arxiv.org/abs/1508.07909

[36] Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are Among Us: Rethinking Programming Language Grammar Towards Efficient Code Generation. arXiv:2404.16333 [cs.SE] https://arxiv.org/abs/2404.16333

[37] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A. Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 945–955. https://doi.org/10.1145/3468264.3468545

[38] van Rossum, Guido and Warsaw, Barry and Coghlan, Nick. 2025. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[40] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. arXiv:2312.02120 [cs.CL] https://arxiv.org/abs/2312.02120

[41] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[42] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, 1073–1084. https://doi.org/10.1145/3540250.3549094