Does Confusing Code Confuse LLMs? A Pilot Study on C++

Tim van de Wetering[1], Lola Solovyeva[1] and Fernando Castor[1]

[1]*Formal Methods & Tools, University of Twente, Enschede, the Netherlands*

Abstract

Large language models (LLMs) are increasingly deployed for software maintenance tasks that depend heavily on code comprehension. While atoms of confusion, minimal code patterns empirically shown to impair human program comprehension, are well studied, it remains unclear whether these same patterns affect LLMs, and whether their impact varies across maintenance tasks. We present a pilot study evaluating the impact of atoms of confusion on LLM performance on C++ code drawn from three production libraries, using ChatGPT and GitHub Copilot on three software development tasks: atom injection and removal, output prediction, and code explanation. Our results reveal three preliminary findings: (i) the impact of atoms on LLMs is strongly task-dependent; (ii) atoms requiring reasoning about surrounding code are the most resistant to transformation, while localized syntactic atoms are handled reliably; and (iii) LLMs remove atoms of confusion more reliably than they inject them, suggesting that removal succeeds through pattern matching while injection requires deeper comprehension.

1. Introduction

Maintaining existing code accounts for 50–90% of total software costs [1, 2], and code comprehension is the bottleneck within that work: developers spend on average 70% of their time understanding code before they can fix bugs, refactor, or add features [3]. Readability is a primary lever on this cost. Clear, well-written code reduces cognitive load and enables efficient maintenance [4, 5], while poor readability, which can be unclear naming, inconsistent formatting, or structural choices, increases comprehension time, error rates during modification, and difficulty finding bugs [6]. In short, anything that makes code harder to read makes it harder to maintain, and at the scale of the global software industry, small readability effects translate into billions of dollars annually spent on comprehension alone [1].

Large language models (LLMs) are increasingly adopted for code-related development tasks [7, 8], offering significant potential to reduce the cost of software maintenance by assisting with comprehension-dependent tasks such as test generation, bug fixing, refactoring, and documentation. Tools like GitHub Copilot and ChatGPT are now used by millions of developers. However, this rapid adoption comes with challenges around output reliability [9]: LLMs can generate code that appears correct but contains subtle bugs, security vulnerabilities, or logical errors. Since these models are trained on existing codebases that vary widely in quality and readability, it raises the question if the same readability factors that affect humans also affect LLM performance. Understanding this relationship is essential for deploying these tools effectively in practice.

While the effect of readability on human code comprehension is well established [10, 11, 12, 13, 14, 15], its effect on LLMs appears different: LLMs do not benefit from human-oriented formatting [16], yet factors such as structure, complexity, and clarity do improve their performance [17]. No study has systematically evaluated whether confusing code patterns known to affect humans also affect LLMs. Such patterns are known as *atoms of confusion*, which are minimal code constructs empirically validated to cause programmers to misunderstand source code, each with a functionally equivalent, clearer alternative [11]. Multiple studies have shown they substantially increase human error rates and comprehension time [12, 15], with some patterns raising error rates by as much as 60% [11]. Since LLMs are trained on codebases containing these patterns, whether they impair LLM comprehension in the same way is an open and practically important question.

This study investigates how atoms of confusion impact LLM performance on practical software development tasks. Understanding this will allow for more effective usage of LLMs in development

BENEVOL2026: Belgium-Netherlands Software Evolution Workshop 26–27 November, 2026, Madrid, Spain.

o.solovyeva@utwente.nl (L. Solovyeva); f.castor@utwente.nl (F. Castor)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

workflows and guide improvements to make models more robust. This study focuses on C++, a language that LLMs have more trouble with compared to more readable languages like Python. Using a language that LLMs already struggle with provides a more useful testing environment, as languages LLMs perform well on could reduce the observable impact of atoms of confusion.

2. Background & Related work

Gopstein at al. [11] introduced atoms of confusion, minimal code patterns that reliably cause programmers to misunderstand source, through empirical experiments with human programmers. Their approach involved extracting candidate patterns from known confusing code, such as winners of the International Obfuscated C Code Contest (IOCCC), and validating these candidates through controlled experiments where programmers hand-evaluate small code snippets. Confusion is operationally defined as occurring when a programmer's interpretation of code behaviour differs from the actual behaviour specified by the language. For every atom there exists a functionally equivalent alternative that is significantly less confusing. Following study of Gopstein et al. [12] demonstrated that atoms occur frequently in production software, analyzing 14 major C/C++ projects, including the Linux kernel and GCC [12]. They found atoms appearing once every 23 lines on average, totaling over 3.6 million occurrences. Bug-fixing commits were 1.25 times more likely to remove atoms than other commits, which suggests that there is a relationship between the usage of these patterns and the occurrence of (certain) bugs in software. Additionally, atoms were 1.13 times more likely to be commented than other code, indicating that developers recognize their potential for confusion.

Moon et al. [18] showed that LLMs exhibit systematic biases when evaluating code correctness based on superficial variations such as authorship claims, self-declared correctness, and variable naming, with accuracy dropping by up to 26.7 percentage points under biased conditions. Their work focuses on surface-level features that do not affect code semantics. Instead, we examine structural patterns, atoms of confusion, that affect readability while preserving functional equivalence. Pan et al. [16] conducted a large-scale study of whether formatting elements such as indentation and newlines affect LLM performance on code completion, across four languages and ten LLMs [16]. They concluded that LLMs do not benefit from human-readability formatting in the same way humans do. Their work examines visual formatting elements that do not alter semantics. However, we examine structural patterns that alter readability and may alter how the code is understood. Ouédraogo et al. [17] introduced a large-scale benchmark for LLM-based code readability assessment (1.4M evaluations across 10 LLMs and 3 languages), finding that developer-informed prompts improve aspect coverage but increase score variability. Their work evaluates LLMs as readability assessors, where we investigate whether poor readability itself degrades LLM performance on comprehension-dependent tasks, regardless of whether readability is explicitly considered. Abdelsalam et al. [19] compared how LLMs and humans process confusing code, finding that LLM perplexity spikes correlate spatially and in amplitude with human neurophysiological responses to atoms of confusion. Their work shows that LLMs exhibit confusion patterns similar to humans. So, complementary we investigate whether these patterns actually impair LLM performance on downstream software development tasks.

3. Methodology

This study investigates how atoms of confusion, code patterns empirically validated to hinder human program comprehension, affect LLM performance across different software development tasks. Therefore, we focus on the following research question:

To what extent do atoms of confusion also affect LLM's ability to understand code?

To answer this question, we evaluate LLM performance on software development tasks using code snippets that contain atoms of confusion from the original study [11], and compare it to their performance on functionally equivalent snippets in which the atoms have been removed.

Table 1
Selected Atoms of Confusion from original study of Gopstein et al. [11]

Atom	Description	Example	Transformed
Pre-/Post-Increment	Post-increment returns the original value before incrementing. Pre-increment first updates then returns the new value.	`V1 = V2++;`	`V1 = V2; V2+=1;`
Operator Precedence	Most programmers know only a subset of C's precedence rules.	`0 && 1 | 2`	`(0 && 1) | 2`
Conditional Operator	The ternary operator functions like if/else but as an expression.	`V2=(V1==3)?2:V2`	`if(V1==3) V2=2;`
Type Conversion	Implicit type conversion may produce unexpected outcomes.	`(double)(3/2)`	`trunc(3.0/2.0)`
Omitted Curly Braces	Without braces, the scope of control statements is unclear.	`if(V) F(); G();`	`if(V){F();}G();`
Repurposed Variables	Using a variable in different roles makes its meaning difficult to follow.	`argc = 7;`	`int V1 = 7;`
Change of Literal Encoding	Different numeric representations affect understandability.	`printf("%d",013)`	`printf("%d",11)`
Comma Operator	The comma operator sequences computations but is often misinterpreted.	`V3=(V1+=1, V1)`	`V1+=1; V3=V1;`
Assignment as Value	Assignment returns a value in addition to changing state.	`V1 = V2 = 3;`	`V2=3; V1=V2;`

Software development tasks. Three tasks were chosen to span different levels of reasoning depth and to reflect common ways LLMs are used in maintenance workflows. This allows us to evaluate whether the impact varies with the nature of the task.

- ***Atom injection and removal:*** the LLMs are prompted to introduce or eliminate a specific atom of confusion in a given code snippet while preserving its behavior. This task probes whether the model can manipulate confusing patterns as structural objects, which is a prerequisite for LLM-assisted refactoring and readability improvements.
- ***Output prediction:*** the LLMs are prompted to predict the output of a program given a specific input. This task requires the model to simulate program execution, and is therefore sensitive to comprehension errors. It reflects tasks such as debugging, test generation, and etc.
- ***Code explanation:*** the LLMs are prompted to describe a code snippet's logic, control flow, and behavior in natural language. This task reflects a common use of LLMs in practice: helping developers understand unfamiliar code. Comparing performance on this task against output prediction lets us assess whether atom impact varies with the reasoning depth a task demands.

Selected atoms of confusion. From the set of candidates for atoms identified in multiple studies [10, 11, 12, 13, 14, 15], the most impactful instances, the instances with the largest effect size, and the instances that occur most frequently were selected and presented in Table 1. We also include one C++ specific candidate, **SFINAE vs Concepts**. SFINAE, Substitution Failure Is Not An Error, typically used through the `enable_if` idiom, conditionally enables template functions via cryptic syntax that hijacks return types or adds dummy parameters. C++20 introduced concepts explicitly to replace it, after more than two decades of community complaints about unreadable code and opaque error messages [20, 21]. Stroustrup himself described `enable_if`-based overloading as "error-prone and tedious assembly programming" [22].

Dataset construction. We construct paired code samples that differ only in the presence or absence of a single atom of confusion while remaining functionally equivalent. Samples are drawn from three widely used, well-tested C++ libraries: ImGui, a UI library used across the game and software industry; Boost.QVM , a header-only linear algebra library from the Boost ecosystem; and ranges-v3, the template-heavy reference implementation of C++20 ranges, chosen specifically because it exercises advanced idioms such as SFINAE. If the sample does not already have an atom of confusion, then they are introduced through code transformation with scripts and in some cases using LLMs from this study.

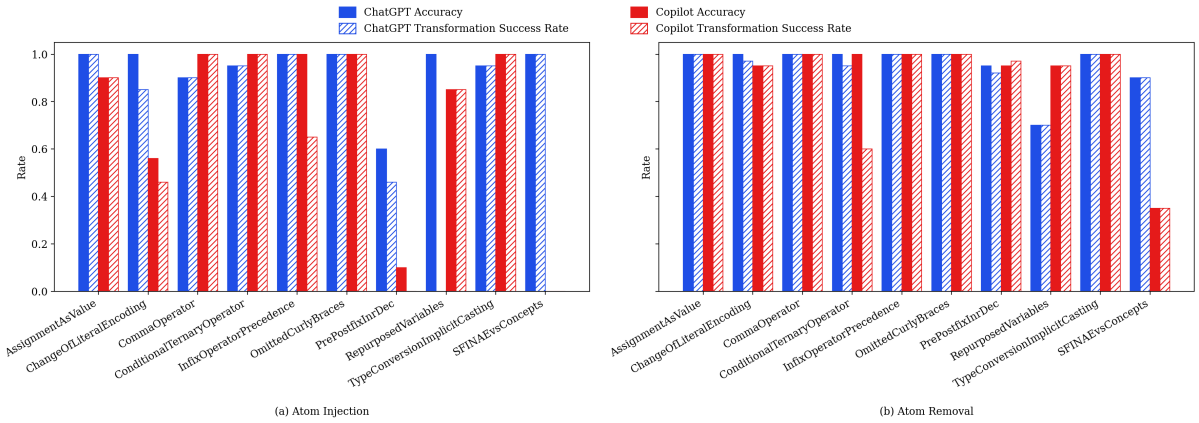


Figure 1: Results of injecting and removing atoms of confusion in code snippets using ChatGPT and Copilot, shown in blue and red, respectively. Solid bars indicate accuracy, measured as the proportion of transformed code snippets that pass the tests, while dashed bars indicate the transformation success rate, i.e., whether the atom of confusion was successfully injected or removed.

All transformed samples are validated through compilation and each library's existing test suite to ensure syntactic correctness and functional equivalence with the original. Transformations that fail these checks are excluded.

Models & Experimental setup. We evaluate two freely accessible frontier LLMs: ChatGPT (GPT-5.3) and GitHub Copilot (Claude Haiku 4.5). Both are the default free-tier models for their respective platforms. For output prediction we additionally evaluate Gemini Flash 3.5, since the smaller sample size for that task made a third model feasible. For atom injection and removal, LLMs were prompted to transform 20 samples per atom. The transformed samples replaced the original in the source library and were validated using the library's existing test suite. For output prediction, we used 5 samples per atom, modified where necessary to expose a return value, and prompted the LLMs in a zero-shot condition. For the code explanation task, the same 20 samples per atom used in the transformation task were provided to the LLMs. The models were instructed to explain each code snippet in terms of its overall purpose, control flow, key computations, and output behavior.

Metrics. For atom injection and removal we use two complementary metrics. Accuracy measures the percentage of transformed samples that still pass the library's test suite, capturing whether the transformation preserved behavior. Transformation Success Rate (TSR) measures the percentage of atoms correctly transformed, capturing whether the intended change was actually performed. The two metrics can diverge: a transformation that leaves the code unchanged achieves 100% accuracy but 0% TSR, while a partial or incorrect transformation may still pass the tests. We report both because each answers a different question. For output prediction, we report the number of correct predictions per atom out of the total samples. For code explanation, explanations are manually graded per category as correct, incorrect (factual errors), or neither (not enough information for evaluation).

4. Results & Discussion

LLMs remove atoms of confusion more reliably than they inject them. Figure 1 presents that removing atoms is generally easier for LLMs than injecting them. Focusing on the TSR, the models particularly struggle to inject *ChangeOfLiteralEncoding*, *InfixOperatorPrecedence*, *PrePostfixIncDec*, *RepurposedVariables*, and *SFINAEvsConcepts*. The distinction between the TSR and accuracy shows for *RepurposedVariables* injection, where ChatGPT achieves 100% accuracy but 0% TSR. The model returned the original code unchanged in all samples, effectively refusing the transformation while still passing tests. A different pattern emerges for atom removal: the models successfully remove most atoms of confusion, with the main exceptions being *ConditionalTernaryOperator*, *RepurposedVariables*,

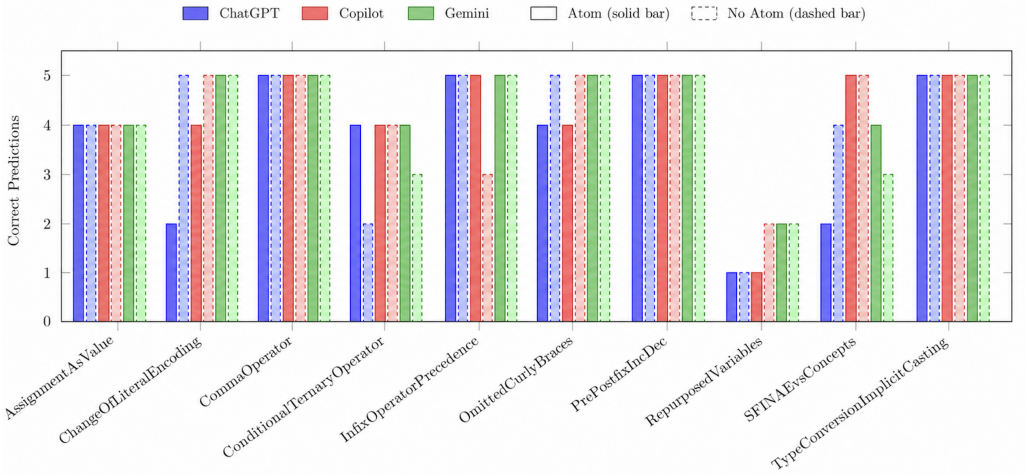


Figure 2: Results of the output prediction task for ChatGPT, Copilot, and Gemini. Bar color indicates the model selected, while line style indicates whether the code snippet contained an atom of confusion: solid bars represent snippets containing an atom of confusion, and dashed bars represent snippets without one.

and *SFINAEvsConcepts*. One plausible explanation is that removing an atom often only requires recognizing a known pattern and replacing it with an equivalent, while injecting an atom requires deeper comprehension via understanding the surrounding code well enough to introduce the construct at a semantically valid location without altering behavior.

Atoms requiring reasoning about surrounding code are the most resistant to transformation. The atoms most resistant to injection share a common property: they require reasoning about surrounding code rather than local rewriting. *SFINAEvsConcepts* requires understanding template constraints, *RepurposedVariables* requires understanding how a variable's role changes across a function, *Pre-/Postfix Increment/Decrement* requires reasoning about evaluation order. In contrast, atoms that are handled reliably in both directions — *OmittedCurlyBraces*, *AssignmentAsValue*, *CommaOperator* — involve localized syntactic transformations. This suggests that LLMs can perform mechanical rewrites reliably but struggle with transformations that demand understanding of surrounding context.

The impact of atoms of confusion on LLMs is task-dependent. Figure 2 compares model performance on code snippets with and without atoms of confusion, allowing us to assess their impact on the ability of models to correctly predict program output. The most notable performance degradation is observed for *ChangeOfLiteralEncoding*, where ChatGPT score decreases from 5 to 2 and Copilot from 5 to 4. For *OmittedCurlyBraces*, the scores of both ChatGPT and Copilot decrease from 5 to 4. A substantial decline is also observed for *SFINAEvsConcepts*, for which ChatGPT score drops from 4 to 2. Several atoms produced no observable degradation for any model, including *CommaOperator*, *InfixOperatorPrecedence*, and *Pre-/Postfix Increment/Decrement*, despite being known sources of confusion for human programmers. *RepurposedVariables* is excluded from this discussion, as performance is uniformly low in both cases, indicating that the atom is not the source of difficulty in these samples.

In contrast, the results for the code explanation task show a different pattern, with accuracy remaining at 100% across nearly all evaluated atoms of confusion and models. Given the consistently high and nearly identical performance across all models, we omit the corresponding plot. The models were generally able to identify the purpose, control flow, and behaviour of the code correctly even when confusing patterns were present. The only exception is *ChangeOfLiteralEncoding*, for which the accuracy decreases to 90% for code snippets containing the atom, the same atom that produced the largest output-prediction drop, and in both cases the errors stemmed from misreading octal literals as decimal values.

These findings suggest that tasks requiring natural-language explanations are considerably more robust to atoms of confusion than tasks requiring precise reasoning about program execution. One possible explanation is that code explanation closely resembles the types of tasks on which modern LLMs are extensively trained. Large amounts of source code, documentation, comments, and question-answer pairs expose models to many examples of describing code behavior in natural language, allowing

them to recognize common programming patterns without necessarily performing detailed execution reasoning. Therefore, generating high-level explanations may rely more on retrieving and generalizing learned representations than on accurately simulating program execution.

> ***Implications.*** Our results show that the models reliably remove confusing code patterns. Hence, they could help clean up, refactor, and simplify code, for example in legacy codebases. Patterns that depend on the surrounding code, such as reused variables or the move from SFINAE to C++20 concepts, still need a developer to review the result. This matters for maintenance because such changes often come up when modernizing older codebases.

References

[1] J. R. Foster, Cost Factors in Software Maintenance., Doctoral, Durham University, 1993. URL: https://etheses.dur.ac.uk/1561/.

[2] J. Mynyk, Information Technology Programming Standards and Annual Project Maintenance Costs, Ph.D. thesis, University of Phoenix, 2012.

[3] R. Minelli, A. Mocci, M. Lanza, I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time, in: 2015 IEEE 23rd ICPC, 2015, pp. 25–35. doi:10.1109/ICPC.2015.12.

[4] M. Weigend, Constructing Complex Things without Getting Confused - Programming Techniques and Reduction of Cognitive Load (2010).

[5] T. P. Nguyen, Optimizing Developer Experience Through Clean Code, 2024. URL: http://www.theseus.fi/handle/10024/879121.

[6] J. Johnson, S. Lubo, N. Yedla, J. Aponte, B. Sharif, An Empirical Study Assessing Source Code Readability in Comprehension, in: 2019 IEEE ICSME, 2019, pp. 513–523. doi:10.1109/ICSME.2019.00085.

[7] L. Schmid, T. Hey, M. Armbruster, S. Corallo, D. Fuchß, J. Keim, H. Liu, A. Koziolek, Software Architecture Meets LLMs: A Systematic Literature Review, 2025. doi:10.48550/arXiv.2505.16697. arXiv:2505.16697.

[8] S. Bistarelli, M. Fiore, I. Mercanti, M. Mongiello, Usage of Large Language Model for Code Generation Tasks: A Review, SN Computer Science 6 (2025) 673. doi:10.1007/s42979-025-04241-5.

[9] M. Nejjar, L. Zacharias, F. Stiehle, I. Weber, LLMs for science: Usage for code generation and data analysis, Journal of Software: Evolution and Process 37 (2025) e2723. doi:10.1002/smr.2723.

[10] D. Gopstein, et al., Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion, in: Proceedings of the 28th ESEC/FSE, 2020.

[11] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos, Understanding misunderstandings in source code, in: Proceedings of the 2017 11th FSE, ESEC/FSE 2017, 2017. doi:10.1145/3106237.3106264.

[12] D. Gopstein, H. H. Zhou, P. Frankl, J. Cappos, Prevalence of confusing code in software projects: Atoms of confusion in the wild, in: Proceedings of the 15th MSR, MSR '18, 2018. doi:10.1145/3196398.3196432.

[13] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, R. Gheyi, An investigation of misunderstanding code patterns in C open-source software projects, Empirical Software Engineering 24 (2019) 1693–1726. doi:10.1007/s10664-018-9666-x.

[14] N. Tahsin, N. Fuad, A. Satter, Prevalence of 'Atoms of Confusion' in Open Source Java Systems: An Empirical Study (2023). URL: https://www.authorea.com/doi/full/10.22541/au.167570695.54470176?commit=d9faaece30c7d2ce7abf98ae8a723053ea51c058.

[15] A. Torres, et al., An Investigation of confusing code patterns in JavaScript, JSS 203 (2023) 111731.

[16] D. Pan, Z. Sun, C. Zhang, D. Lo, X. Du, The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget, 2025. doi:10.48550/arXiv.2508.13666.

[17] W. C. Ouédraogo, Y. Li, X. Dang, P. Borsukiewicz, X. Zhou, A. Koyuncu, J. Klein, D. Lo, T. F. Bissyandé, Human-Aligned Code Readability Assessment with Large Language Models, 2025. doi:10.48550/arXiv.2510.16579.

[18] J. Moon, Y. Hwang, D. Lee, T. Kang, Y. Kim, K. Jung, Don't Judge Code by Its Cover: Exploring Biases in LLM Judges for Code Evaluation, 2025. doi:10.48550/arXiv.2505.16222.

[19] Y. Abdelsalam, N. Peitek, A.-M. Maurer, M. Toneva, S. Apel, How do Humans and LLMs Process Confusing Code?, 2025. doi:10.48550/arXiv.2508.18547. arXiv:2508.18547.

[20] M. Calabrese, Why I want Concepts, but why they should come later rather than sooner, ???? URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0240r0.html.

[21] Duncan P. N. Exon Smith, P0726R0: Does the Concepts TS Improve on C++17?, ???? URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0726r0.html.

[22] B. Stroustrup, How to design good concepts and use them well (2017). URL: https://www.stroustrup.com/good_concepts.pdf.