TOKDRIFT: When LLM Speaks in Subwords but Code Speaks in Grammar

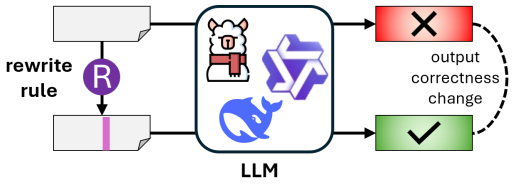
Yinxi Li, Yuntian Deng, Pengyu Nie
University of Waterloo
{yinxi.li,yuntian,pynie}@uwaterloo.ca

Abstract

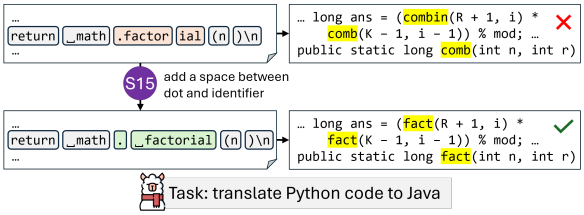
Large language models (LLMs) for code rely on subword tokenizers, such as byte-pair encoding (BPE), learned from mixed natural language text and programming language code but driven by statistics rather than grammar. As a result, semantically identical code snippets can be tokenized differently depending on superficial factors such as whitespace or identifier naming. To measure the impact of this misalignment, we introduce TOKDRIFT, a framework that applies semantic-preserving rewrite rules to create code variants differing only in tokenization. Across nine code LLMs, including large ones with over 30B parameters, even minor formatting changes can cause substantial shifts in model behavior. Layer-wise analysis shows that the issue originates in early embeddings, where subword segmentation fails to capture grammar token boundaries. Our findings identify misaligned tokenization as a hidden obstacle to reliable code understanding and generation, highlighting the need for grammar-aware tokenization for future code LLMs.

1 Introduction

Large language models (LLMs) have become powerful tools for programming tasks (Chen et al., 2021; Nye et al., 2021; Yang et al., 2024; Guo et al., 2024; Meta FAIR CodeGen Team, 2025). Before any modeling occurs, code is first tokenized into discrete units using a pretrained subword tokenizer such as byte-pair encoding (BPE; Sennrich et al., 2016). However, the tokens that LLMs see, which are based on subword frequencies, are often very different from the tokens defined by programming language (PL) grammar. Whereas PLs have clear syntactic boundaries (e.g., keywords, identifiers, operators), subword tokenizers merge character sequences statistically, sometimes splitting identifiers at arbitrary points or combining unrelated symbols into a single token. This misalignment between



(a) Workflow of TOKDRIFT, our framework for quantifying LLM sensitivity to semantic-preserving code rewrite rules.



(b) Example of tokenization misalignment. Adding a space between dot (".") and "factorial" causes a significant change in token sequences, from [".factor", "ial"] to [".", "␣factorial"]. Consequently, the LLM's code translation prediction shifts from incorrect (naming the factorial function as "comb" and later referring to it as "combin") to correct.

Figure 1: TOKDRIFT workflow and example.

subwords and syntax means that LLMs do not always process code in the units that programmers or compilers would expect.

As an example, the presence of a space before an identifier can lead to completely different token sequences, and thus different predictions, despite identical program semantics (Figure 1). While such differences may appear superficial, they raise a deeper concern about how robustly code LLMs represent grammar and meaning. If tokenization determines how code is segmented and embedded, even small discrepancies could propagate through the model and alter its predictions. This motivates the central question of our study:

> *Does the misalignment between subword tokenization and PL grammar limit LLMs' ability to understand and generate code?*

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 47605–47629
July 2-7, 2026

To study this question, we introduce TOKDRIFT, a framework that applies semantic-preserving rewrite rules, such as changing whitespace or identifier casing style, to create pairs of programs that are semantically equivalent but tokenized differently. We evaluate nine code LLMs across three representative programming tasks—bug fixing, code summarization, and code translation—and measure whether model outputs remain functionally equivalent when tokenization changes.

Our experiments show that even minor tokenization variations can substantially impact model behavior. For example, the most performant LLM in our experiment, Qwen2.5-Coder-32B-Instruct, changes its prediction 6.09% of the times when the input tokenization changes (and up to 60% under a single rewrite rule). Layer-wise analysis further indicates that the effect originates in early layers, where subword segmentation fails to align with grammatical token boundaries. Together, these findings suggest that tokenizer design remains a critical yet under-explored factor in developing robust and grammar-aware code LLMs.

The main contributions of this work include:

- We identify and formalize the misaligned tokenization problem in code LLMs.
- We introduce TOKDRIFT, a framework for quantifying model sensitivity to semantic-preserving code rewrites that alter tokenization.
- We conduct a large-scale empirical study showing that misaligned tokenization affects all evaluated models and persists with scaling.
- We open-source our framework and data to facilitate future research on grammar-aware and domain-adaptive tokenization.

Our code and data are available at `https://github.com/uw-swag/tokdrift`.

2 Background

2.1 LLM Tokenization

Tokenization is the first step in processing input for LLMs, converting raw text into a sequence of discrete tokens. Each token corresponds to a model time step and has a dedicated embedding. Modern LLMs use learned tokenization strategies that eliminate the out-of-vocabulary problem by starting from minimal units, such as characters or bytes, and learning how to merge them into longer fragments based on frequency in a large corpus. Popular approaches like BPE (Sennrich et al., 2016) and WordPiece (Schuster and Nakajima, 2012; Devlin et al., 2019) follow this general principle, differing mainly in their merge heuristics. Often, pre-tokenization steps like splitting at whitespace are applied before learning to prevent tokens from spanning across word boundaries.

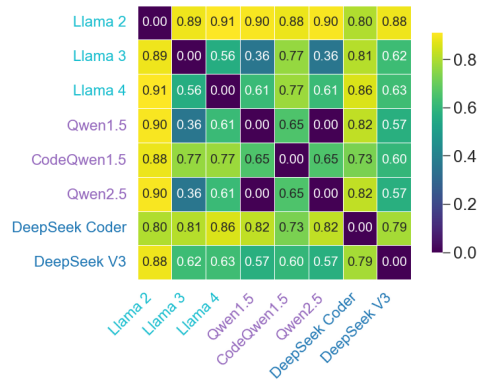


Figure 2: Heatmap of (code) LLMs' vocabulary distances (Amba Hombaiah et al., 2021).

The tokenizers used by different LLMs can vary significantly due to differences in pre-tokenization rules, token learning algorithms, and pretraining corpora. As shown in Figure 2, even models from the same family often share less than half of their vocabulary, such as Llama 3 vs. Llama 4. The main exception occurs when model developers intentionally reuse the same tokenizer across variants, such as Qwen2.5 and Qwen2.5-Coder, which share an identical vocabulary and tokenizer configuration.

2.2 PL Tokenization

Tokenization in PLs, often called *lexing*, is the first step of code parsing: it transforms a stream of characters into a sequence of tokens according to a PL's grammar. These tokens are then passed to a parser, which constructs an abstract syntax tree (AST) to represent the program's structure.

While exact rules vary by language, most PLs share a common set of token types, including: *identifiers* (e.g., variable or function names), *operators* (e.g., `+`, `*`), *keywords* (e.g., `if`, `return`), *literals* (e.g., numeric or string constants), and *whitespace*, which is typically used to separate tokens but is otherwise ignored.

Unlike LLM tokenization, PL tokenization in compilers and interpreters is deterministic. For example, the snippet `x+1` is always tokenized into three tokens: an identifier (`x`), an operator (`+`), and a literal (`1`). Formatting changes, such as adding spaces, do not affect the token sequence as long as the code remains syntactically valid.

Table 1: Benchmarks in our experiments. We manually examine the benchmarks to follow the naming conventions, and to fix/exclude invalid tests and samples, see details in Appendix C.1.

Benchmark	Source	Task	Input PL	Output PL	# Samples
HumanEval-Fix-py	HumanEvalPack (Muennighoff et al., 2023)	bug fixing	Python	Python	164
HumanEval-Fix-java			Java	Java	164
HumanEval-Explain-py		code summarization	Python	Python	164
HumanEval-Explain-java			Java	Java	164
Avatar-py2java	Avatar (Ahmad et al., 2023; Pan et al., 2024)	code translation	Python	Java	244
Avatar-java2py			Java	Python	246
CodeNet-py2java	CodeNet (Puri et al., 2021; Pan et al., 2024)		Python	Java	200
CodeNet-java2py			Java	Python	200

This behavior misaligns with LLM tokenizers: while PL tokenizers produce stable, grammar-aware units, LLM tokenizers frequently break code structure, resulting in inconsistent or fragmented representations of semantically identical programs. In this work, we refer to grammar-aware tokens as *PL tokens*, and contrast them with the *LLM tokens* produced by learned subword tokenizers.

3 TOKDRIFT Framework

Figure 1a illustrates the overall workflow of TOKDRIFT, our framework for quantifying model sensitivity to semantic-preserving code rewrites that alter tokenization. In a nutshell, TOKDRIFT systematically compares the LLM outputs given the baseline input tokens and variant input tokens (after applying rewrite rules) through a large set of experiments. Each experiment is performed on a specific benchmark, and tests the sensitivity of a given LLM against a specific rewrite rule.

3.1 Benchmarks

We searched for recent popular coding LLM benchmarks where: (1) the input include a code snippet, since rewrite rules cannot be applied on natural language; (2) the output is evaluated with an automated functional correctness metric.We focused on two popular PLs, Java and Python.

Based on these criteria, we selected eight benchmarks covering three tasks, listed in Table 1. Bug fixing (Tufano et al., 2019) transforms a buggy code snippet into a correct one. Code summarization (Hu et al., 2018; Panthaplackel et al., 2020) aims at summarizing a code snippet into natural language description; following HumanEvalPack's setup (Muennighoff et al., 2023), the description is fed back to LLM to generate code for measuring correctness. Code translation (Ahmad et al., 2023; Puri et al., 2021) is the task of translating a code snippet from one PL to another. All benchmarks use tests to evaluate the correctness of outputs.

Table 2: Models used in our experiments.

Series	S	M	L
Llama-3	3B	8B	70B
Qwen2.5-Coder	1.5B	7B	32B
DeepSeek-Coder	1.3B	6.7B	33B

3.2 Models

Table 2 lists the models used in TOKDRIFT. We selected three series of popular open-source LLMs (using the coding-specific variants if available), namely Llama-3, Qwen2.5-Coder, and DeepSeek-Coder. To cover the model size spectrum, we used small ($\sim$1B parameters), medium ($\sim$7B), and large ($>$30B) variants in each series. All models are instruction-tuned. We perform greedy decoding to generate deterministic outputs (see experimental environment details in Appendix C.4).

3.3 Rewrite Rules

Table 3 lists the rewrite rules used in TOKDRIFT. Each rewrite rule converts all occurrences of the left-hand side substring to the right-hand side substring. According to the grammars of the two PLs we experiment on (and generally for most modern PLs), these rewrite rules are semantically-preserving by design. We apply one rewrite rule at a time to investigate their impact in isolation.

The six rewrite rules starting with "N" are inspired by naming conventions. Identifiers usually follow one of the four casing styles: `camelCase` (for variables/functions in Java), `PascalCase` (for classes in Java/Python), `snake_case` (for variables/functions in Python), and `SCREAMING_CASE` (for constants in Java/Python). Since variables/-functions are most common among identifiers, we design rewrite rules to alter their casing style. Specifically, N1, N2, N3 convert `camelCase` identifiers in Java to the other three casing styles, while N4, N5, N6 convert `snake_case` identifiers in Python. These rewrite rules challenge LLMs' robustness to different naming styles.

Table 3: Rewrite rules supported by TOKDRIFT, inspired by naming conventions (starting with N) and spacing conventions (starting with S). Each rewrite rule may apply to Java (marked by J), Python (marked by P), or both.

No.	PL	Rewrite Rule	Description	Example
N1	J	camelCase →snake_case	Convert identifiers from the most common casing style in the input PL to alternative ones	␣sorted L st→␣sorted _lst
N2	J	camelCase →PascalCase		␣cloestPair→␣Close st Pair
N3	J	camelCase →SCREAMING_CASE		␣possible S olutions→␣POSS IBLE _S OLUTION S
N4	P	snake_case →camelCase		␣input _clip board→␣input Clipboard
N5	P	snake_case →PascalCase		␣string _xor→␣String X or
N6	P	snake_case →SCREAMING_CASE		␣triangle _area→␣TRI ANGLE _AREA
S1	P	OP -→OP␣-	Add space between operator and minus sign	[::- 1]→[:: ␣- 1]
S2	P	OP [→OP␣[	Add space between operator and left square bracket	)))[2 :]\n→))) ␣[2 :]\n
S3	J	) .→)␣.	Add space between right parentheses and period	␣'. '). replace→␣'.') ␣. replace
S4	P	])→]␣)	Add space between right square bracket and right parentheses	:]):\n→:] ␣):\n
S5	P	OP]→OP␣]	Add space between operator and right square bracket	= ␣[[]→= ␣[[␣]
S6	J	OP (→OP ␣(	Add space between operator and left parentheses	((! is True→(␣(! is True
S7	P	[ID→[␣ID	Add space between left square bracket and identifier	([v ow els→([␣vowels
S8	J	++)→++␣)	Add space between increment operator and right parentheses	␣i ++)→␣i ++ ␣)
S9	J	. *→.␣*	Add space between period and asterisk	.*;\n→. ␣* ;\n
S10	P	) :→)␣:	Add space between right parentheses and colon	␣main ():\n→␣main () ␣:\n
S11	J	) ;→)␣;	Add space between right parentheses and semicolon	<>();\n→< >() ␣;\n
S12	J	OP ;→OP␣;	Add space between operator and semicolon	Ac ++;→Ac ++ ␣;
S13	J P	))→)␣)	Add space between two right parentheses	.toCharArray ())→.toCharArray () ␣)
S14	J P	( )→(␣)	Add space between left and right parentheses	alpha ()→alpha (␣)
S15	J P	. ID→.␣ID	Add space between period and identifier	.factor ial→. ␣factorial
S16	J P	(ID→(␣ID	Add space between left parentheses and identifier	(String→(␣String
S17	J P	OP ID→OP␣ID	Add space between operator and identifier	:i +len (sub string→: ␣i + ␣len (␣substring
S18	J P	OP ALL→OP␣ALL	Add space between operator and identifier/operator	(l : ␣list):\n→(␣l : ␣list) ␣:\n

The eighteen rewrite rules starting with "S" are inspired by spacing conventions. Whitespace around most operators is usually non-semantic and optional. Thus, the spacing-related rewrite rules identifies two consecutive tokens (one being an operator) and inserts a space in between. Specifically, we look for combinations where one of them is a specific operator or any kind of operator (represented by OP), and the other one is another specific operator or an identifier (represented by ID). Exploring all combinations would be infeasible, thus we select the top-10 frequently appearing combinations in the benchmarks for each PL. In addition, we add S17 and S18 as "wildcard" rules to cover all cases where an OP is followed by an ID or ID/OP for both PLs. These rewrite rules challenge LLM and its tokenizer's robustness to different formatting styles. Notably, in most LLMs with a pre-tokenization step of splitting before whitespace, these rewrite rules will lead to more LLM tokens.

3.4 Metrics

Recall that each experiment on a given {benchmark, model, rewrite rule} triplet compares the *baseline* outputs (given the original inputs) and the *variant* outputs (given the inputs after applying rewrite rule). The benchmark provides a set of tests to evaluate whether each output is correct or incorrect. We define accuracy as the percentage of the correct outputs, and **Δaccuracy** as the variant's accuracy minus the baseline's accuracy.

The Δaccuracy metric, although intuitive, has two limitations: (1) accuracy improvements and degradations on individual samples cancel out; (2) some samples may not be affected by a rewrite rule if the left-hand side substring does not appear in the input; the outputs of those samples will never change. To address these, we introduce an unbiased metric called **sensitivity**, defined as the percentage of the samples whose output correctness flips (from correct to incorrect or vice versa) out of the samples whose input is changed by the rewrite rule. A lower sensitivity indicates that the model is more robust against the token changes introduced by a rewrite rule; when averaged across all rewrite rules, it reflects how sensitive the model is to the LLM-PL tokenization misalignment.

4 Evaluation

4.1 Results

Table 4 shows the accuracy and Δaccuracy of each model on each rewrite rule. We can observe that most rewrite rules cause measurable changes in model accuracy, ranging from -2.90 to +0.32 absolute percentage points if averaging across all mod-

Table 4: Accuracy and Δaccuracy (in parenthesis) of each model on each rewrite rule.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
Input PL = Java										
baseline	32.04	43.15	57.24	33.59	57.36	70.41	38.50	58.01	57.36	49.74
N1	**32.69** (+0.65)	43.54 (+0.39)	57.49 (+0.25)	35.27 (+1.68)	57.62 (+0.26)	70.28 (-0.13)	37.98 (-0.52)	57.36 (-0.65)	57.11 (-0.25)	49.93 (+0.19)
N2	32.17 (+0.13)	43.54 (+0.39)	56.85 (-0.39)	35.27 (+1.68)	57.75 (+0.39)	70.41 (+0.00)	**39.02** (+0.52)	58.14 (+0.13)	57.36 (+0.00)	**50.06** (+0.32)
N3	32.56 (+0.52)	44.19 (+1.04)	56.20 (-1.04)	35.53 (+1.94)	58.01 (+0.65)	69.12 (-1.29)	38.37 (-0.13)	56.33 (-1.68)	56.46 (-0.90)	49.64 (-0.10)
S3	31.65 (-0.39)	43.02 (-0.13)	56.20 (-1.04)	34.37 (+0.78)	56.72 (-0.64)	70.41 (+0.00)	37.34 (-1.16)	**58.66** (+0.65)	57.88 (+0.52)	49.58 (-0.16)
S6	31.52 (-0.52)	43.02 (-0.13)	57.62 (+0.38)	33.20 (-0.39)	57.49 (+0.13)	70.28 (-0.13)	37.98 (-0.52)	58.53 (+0.52)	57.49 (+0.13)	49.68 (-0.06)
S8	31.91 (-0.13)	43.28 (+0.13)	57.24 (+0.00)	34.11 (+0.52)	56.72 (-0.64)	71.45 (+1.04)	38.63 (+0.13)	57.49 (-0.52)	58.27 (+0.91)	49.90 (+0.16)
S9	32.30 (+0.26)	40.96 (-2.19)	**58.66** (+1.42)	33.46 (-0.13)	**58.14** (+0.78)	69.51 (-0.90)	36.95 (-1.55)	56.59 (-1.42)	57.75 (+0.39)	49.37 (-0.37)
S11	32.69 (+0.65)	**44.57** (+1.42)	55.17 (-2.07)	35.14 (+1.55)	56.33 (-1.03)	71.58 (+1.17)	37.34 (-1.16)	57.11 (-0.90)	57.11 (-0.25)	49.67 (-0.07)
S12	30.49 (-1.55)	43.02 (-0.13)	56.07 (-1.17)	34.75 (+1.16)	55.81 (-1.55)	67.05 (-3.36)	38.63 (+0.13)	55.94 (-2.07)	58.53 (+1.17)	48.92 (-0.82)
S13	32.43 (+0.39)	42.64 (-0.51)	56.59 (-0.65)	33.46 (-0.13)	57.36 (+0.00)	69.77 (-0.64)	37.47 (-1.03)	58.27 (+0.26)	56.98 (-0.38)	49.44 (-0.30)
S14	29.84 (-2.20)	41.09 (-2.06)	54.13 (-3.11)	32.17 (-1.42)	56.85 (-0.51)	71.19 (+0.78)	37.86 (-0.64)	57.11 (-0.90)	57.62 (+0.26)	48.65 (-1.09)
S15	30.62 (-1.42)	36.82 (-6.33)	57.24 (+0.00)	33.46 (-0.13)	56.72 (-0.64)	70.28 (-0.13)	37.34 (-1.16)	55.43 (-2.58)	**59.43** (+2.07)	48.59 (-1.15)
S16	30.88 (-1.16)	40.83 (-2.32)	55.94 (-1.30)	34.88 (+1.29)	57.36 (+0.00)	**71.96** (+1.55)	36.43 (-2.07)	57.49 (-0.52)	58.66 (+1.30)	49.38 (-0.36)
S17	28.68 (-3.36)	37.34 (-5.81)	56.07 (-1.17)	**35.66** (+2.07)	55.43 (-1.93)	70.03 (-0.38)	35.40 (-3.10)	55.04 (-2.97)	58.91 (+1.55)	48.06 (-1.68)
S18	25.97 (-6.07)	34.88 (-8.27)	56.85 (-0.39)	34.11 (+0.52)	56.07 (-1.29)	70.28 (-0.13)	33.98 (-4.52)	53.10 (-4.91)	56.33 (-1.03)	46.84 (-2.90)
Input PL = Python										
baseline	39.12	49.87	69.04	40.67	64.51	76.17	44.82	61.92	68.13	57.14
N4	40.03 (+0.91)	**51.04** (+1.17)	68.91 (-0.13)	39.77 (-0.90)	**65.03** (+0.52)	**77.85** (+1.68)	44.30 (-0.52)	61.53 (-0.39)	**68.39** (+0.26)	**57.43** (+0.29)
N5	37.56 (-1.56)	50.91 (+1.04)	68.65 (-0.39)	39.25 (-1.42)	64.77 (+0.26)	77.72 (+1.55)	42.88 (-1.94)	61.53 (-0.39)	68.39 (+0.26)	56.85 (-0.29)
N6	38.08 (-1.04)	50.65 (+0.78)	66.19 (-2.85)	39.38 (-1.29)	64.51 (+0.00)	76.81 (+0.64)	42.23 (-2.59)	61.14 (-0.78)	67.62 (-0.51)	56.29 (-0.85)
S1	39.38 (+0.26)	50.39 (+0.52)	68.65 (-0.39)	40.54 (-0.13)	64.51 (+0.00)	76.68 (+0.51)	44.69 (-0.13)	62.56 (+0.64)	67.62 (-0.51)	57.22 (+0.08)
S2	39.64 (+0.52)	50.65 (+0.78)	68.78 (-0.26)	40.41 (-0.26)	64.77 (+0.26)	75.91 (-0.26)	43.65 (-1.17)	62.44 (+0.52)	67.75 (-0.38)	57.11 (-0.03)
S4	39.77 (+0.65)	50.65 (+0.78)	**69.30** (+0.26)	40.54 (-0.13)	64.51 (+0.00)	73.19 (-2.98)	44.82 (+0.00)	61.92 (+0.00)	67.36 (-0.77)	56.90 (-0.24)
S5	38.60 (-0.52)	50.78 (+0.91)	68.91 (-0.13)	**40.80** (+0.13)	64.12 (-0.39)	76.94 (+0.77)	44.43 (-0.39)	**62.69** (+0.77)	66.71 (-1.42)	57.11 (-0.03)
S7	40.03 (+0.91)	49.35 (-0.52)	68.26 (-0.78)	40.67 (+0.00)	63.34 (-1.17)	76.42 (+0.25)	44.30 (-0.52)	62.69 (+0.77)	67.23 (-0.90)	56.92 (-0.22)
S10	38.47 (-0.65)	50.65 (+0.78)	69.17 (+0.13)	40.67 (+0.00)	63.99 (-0.52)	77.46 (+1.29)	44.56 (-0.26)	62.05 (+0.13)	67.10 (-1.03)	57.12 (-0.02)
S13	37.95 (-1.17)	50.13 (+0.26)	69.30 (+0.26)	40.54 (-0.13)	64.90 (+0.39)	76.55 (+0.38)	44.30 (-0.52)	62.05 (+0.13)	67.10 (-1.03)	56.98 (-0.16)
S14	38.73 (-0.39)	49.22 (-0.65)	68.39 (-0.65)	39.38 (-1.29)	63.73 (-0.78)	74.09 (-2.08)	**45.08** (+0.26)	61.66 (-0.26)	67.49 (-0.64)	56.42 (-0.72)
S15	39.12 (+0.00)	50.26 (+0.39)	67.49 (-1.55)	39.77 (-0.90)	62.69 (-1.82)	76.30 (+0.13)	44.17 (-0.65)	61.66 (-0.26)	67.23 (-0.90)	56.52 (-0.62)
S16	40.16 (+1.04)	49.87 (+0.00)	69.04 (+0.00)	39.64 (-1.03)	63.08 (-1.43)	76.68 (+0.51)	43.65 (-1.17)	61.27 (-0.65)	67.23 (-0.90)	56.74 (-0.40)
S17	**40.41** (+1.29)	50.39 (+0.52)	67.62 (-1.42)	39.38 (-1.29)	61.92 (-2.59)	76.55 (+0.38)	42.62 (-2.20)	60.49 (-1.43)	66.32 (-1.81)	56.19 (-0.95)
S18	37.44 (-1.68)	49.87 (+0.00)	67.62 (-1.42)	38.34 (-2.33)	63.08 (-1.43)	75.13 (-1.04)	42.49 (-2.33)	62.05 (+0.13)	67.36 (-0.77)	55.93 (-1.21)

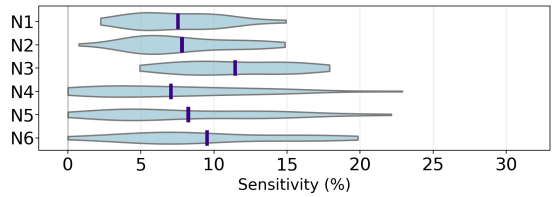
Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

els. The largest Δaccuracy of -8.27% happens on Llama-8B for Java benchmarks, whose accuracy drops from 43.15% to 34.88% when applying rewrite rule S18 (adding space after each operator). Considering advances in LLM performance are sometimes claimed with around 1 percentage point margin, these accuracy deltas caused by simple rewrite rules are non-negligible.

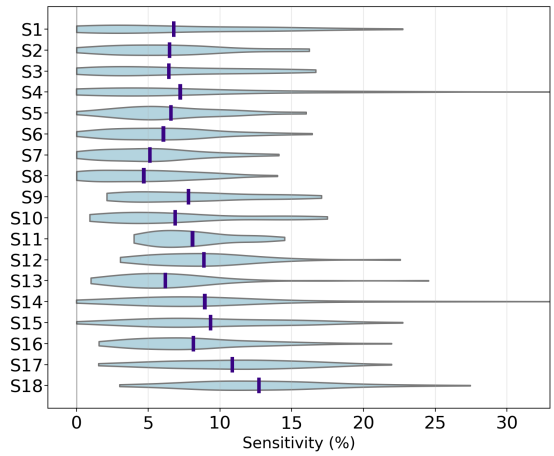
The impact of misaligned tokenization is more apparent in the sensitivity metric, as shown in the distribution plots in Figure 3. The average sensitivity is 9.26% for naming rewrites and 8.29% for spacing rewrites. Among the naming rewrites (Figure 3a), LLMs are relatively less sensitive to transductions between `camelCase` and `snake_case` (N1 and N4), likely because `camelCase` and `snake_case` are less frequent. This finding implies that the casing styles of identifiers, while technically convey no semantic meaning in PLs, are an important factor in LLMs' understanding of code. In Figure 3b, we can see that LLMs' average sensitivity is over 10% for the two "wildcard" spacing rewrite rules (S17 and S18). Other spacing rewrite rules result in varying levels of sensitivity, among which the most impactful ones are S15 (adding space between period and identifier), S14 (adding space between a pair of parentheses), and S12 (adding space between operator and semicolon). In terms of the average sensitivity of models (Figure 3c), we observe that Llama-3 models are more sensitive than the other two series, but all models persist a non-negligible sensitivity of at least 5.71% (Qwen-32B on spacing rewrite rules). To further verify that our observations are not an artifact of greedy decoding, we also rerun a subset of our evaluated tasks with nucleus sampling and minimum-risk selection, and find that sensitivity remains non-negligible (see Appendix D.2).

4.2 Impact of Model Size

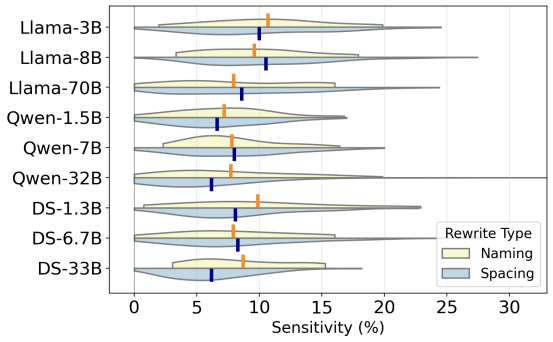
We investigate whether larger models are less sensitive to tokenization changes, with the general assumption of larger models being more robust. Table 5 shows the the average sensitivity of models



(a) grouped by naming rewrite rule



(b) grouped by spacing rewrite rule



(c) grouped by model

Figure 3: Violin plots of sensitivity distributions.

at different sizes, where the small, medium, and large models in each series are compared on a row. While the small and medium models are at around the same level of sensitivity, the large models are usually less sensitive (i.e., more robust) than their smaller counterparts, with only one exception of Qwen-32B on naming rewrite rules.

We also perform statistically significant tests via Wilcoxon signed-rank test (Conover, 1999). The results show that the differences are not significant for naming rules, but significant for spacing rules (except between the small and medium models for Qwen2.5-Coder and DeepSeek-Coder series).

4.3 Impact of Identifier Fragment Changes

We noticed that identifiers are frequently tokenized into different subwords before and after applying rewrite rules. For example, Llama-3 tokenizes '␣sortedLst' into three tokens ['␣sorted', 'L', 'st'], and applying N1 changes it into two tokens ['␣sorted', '_lst']. We define this case as *identifier fragment change*: the list of fragments (tokens but ignoring spaces and underscores) changes before and after applying rewrite rules. Using this concept, we can categorize the samples into two groups, one without any identifier fragment change (i.e., "Unchanged"), and the other with at least one identifier fragment change (i.e., "Changed").

Table 5: Impact of model size on sensitivity.

Rewrite Rule	Model Series	S	M	L
Naming	Llama-3	11.48	10.68	9.43
	Qwen2.5-Coder	7.73	7.95	8.27
	DeepSeek-Coder	9.88	8.95	8.95
Spacing	Llama-3	10.22	10.99	8.51
	Qwen2.5-Coder	7.07	8.87	5.71
	DeepSeek-Coder	8.36	8.71	6.26

Table 6: Impact of identifier fragment changes on sensitivity. "Unchanged" samples do not have any identifier fragment change, and "Changed" samples have at least one identifier fragment change.

Rewrite Rule	Model	Unchanged	Changed
Naming	Llama-70B	8.13	11.21
	Qwen-32B	6.58	10.57
	DS-33B	6.61	10.82
Spacing	Llama-70B	7.24	11.89
	Qwen-32B	5.09	7.37
	DS-33B	5.80	7.12

Table 6 shows the average sensitivity of models on the two groups of samples; note that we focus on the large model in each series in this analysis. The identifier fragment changed group shows consistently higher sensitivity than the unchanged group, with the largest difference on for naming rewrite rules (10.82% vs. 6.61%). This finding suggests that how identifiers are tokenized into subwords plays an important role in LLMs' understanding of code. Indeed, identifiers are frequently *not* tokenized into semantically meaningful subwords (such as the '␣sortedLst' example), and our control study in Section 5.3 further confirms that this subword–grammar misalignment causally contributes to the observed sensitivity rather than being an incidental correlate.

5 Root Cause Analyses

In addition to quantifying its impact, we also study *why* LLMs are sensitive to tokenization changes, along three aspects: (1) word frequency in the pre-training corpus (Section 5.1); (2) LLM's hidden states before and after the rewrite rule (Section 5.2);

Table 7: Word frequency of rewrite rules' left-hand side (LHS) and right-hand side (RHS) on GitHub. Ratio is the percentage of RHS to LHS word frequency.

Rewrite Rule	LHS	RHS	Ratio [%]
Java			
S3:) . →) ␣ .	78.9M	45.7K	0.06
S8: ++) →++ ␣)	22.9M	664K	2.90
S9: . * →. ␣ *	34.2M	7.3M	21.35
S11:) ; →) ␣ ;	161M	924K	0.57
S13:)) →) ␣)	102M	3.4M	3.33
S14: ( ) →(␣)	144M	195K	0.14
S15: . ID→. ␣ ID	175M	45.9M	16.22
S16: (ID→(␣ ID	172M	6.6M	3.84
Python			
S4:]) →] ␣)	44.6M	1.7M	3.81
S7: [ID→[␣ ID	61.1M	1.1M	1.83
S10:) : →) ␣ :	76M	1.4M	1.84
S13:)) →) ␣)	59M	2.4M	4.07
S14: ( ) →(␣)	78.1M	71.7K	0.09
S15: . ID→. ␣ ID	107M	40.6M	37.94
S16: (ID→(␣ ID	105M	2.9M	2.76

(3) a controlled intervention that enforces grammar-aligned boundaries via a Python-lexer pre-tokenizer (Section 5.3).

5.1 Word Frequency Analysis

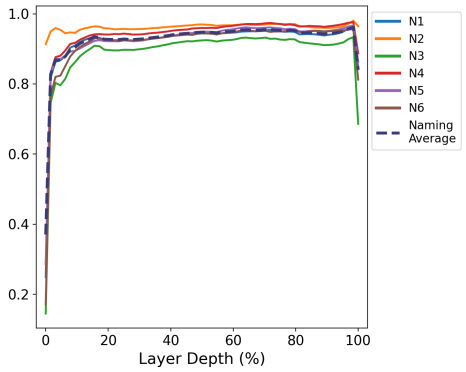
Our hypothesis is that there is a correlation between sensitivity and the word frequencies of the rewrite rule's left-hand side and right-hand side. If the ratio of right-hand side to left-hand side word frequency is small (meaning right-hand side is rare in the corpus), LLMs will likely perform worse after applying the rewrite rule. We measure the word frequencies on GitHub, a primary source of code data in LLMs' pretraining corpora.[1]

Table 7 shows the word frequencies of the rewrite rules, and the ratio (in percentages) of the right-hand side to the left-hand side word frequency. The ratio is always less than 100%, which explains why LLMs exhibit non-negligible sensitivity to all rewrite rules. Some rewrite rules with low ratio, e.g., S14, also exhibit high sensitivity in Figure 3b.

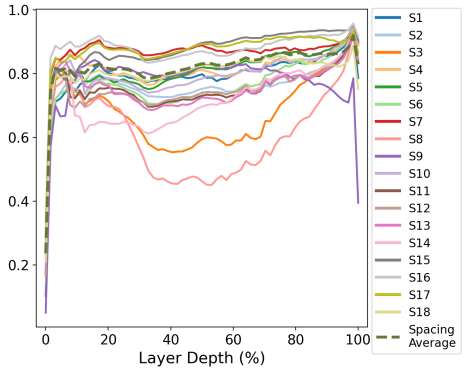
5.2 Hidden State Analysis

LLMs' hidden states represent their internal comprehension and reasoning processes, which may help explain their sensitive to tokenization changes. We compare the hidden states before and after applying the rewrite rules. For each tokens sequence changed, we extract the hidden states of the *last* token in the sequence, which summarizes the information of the entire sequence. We focus this analysis on the best-performing LLM, Qwen-32B.

[1]We use GitHub's search feature to measure word frequencies; due to the limitation in regular expressions and characters that can be used in the search string, we can only conduct this analysis on a subset of the spacing rewrite rules.



(a) naming rewrite rules

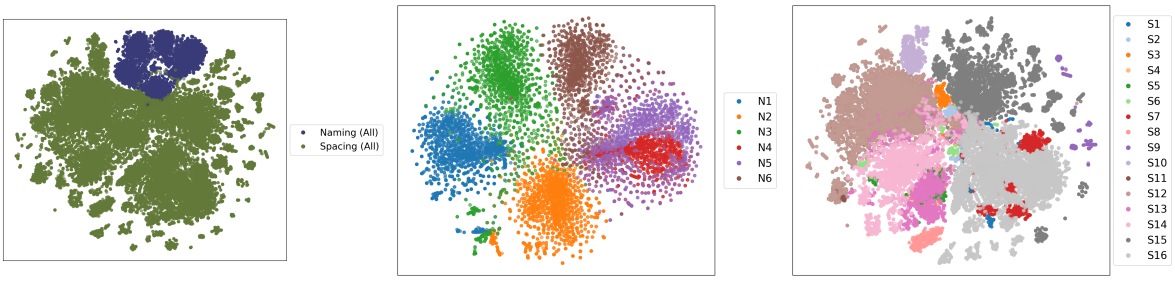


(b) spacing rewrite rules

Figure 4: The similarity of each layer's hidden states before and after applying rewrite rules.

We first measure the cosine similarity between the hidden states before and after applying the rewrite rules. Figure 4 shows correlation between the layer from which the hidden states are extracted and the similarity. For both naming and spacing rewrite rules, the similarity starts from almost 0 in the first (input) layer, increases (and stabilizes in most cases) in middle layers, and drops again at the last (output) layer. This observation is consistent with the information bottleneck theory (Saxe et al., 2019), which states that the middle layers capture the compressed semantic information. Interestingly, in Figure 4b, we observe that for some spacing rewrite rules (S14 and S3), the similarity in middle layers is also low, implying that the model sees the before and after versions as semantically different. These rewrite rules match the ones that LLMs are most sensitive to in Figure 3b.

Then, we compute the *hidden state diffs* as the hidden states after applying rewrite rules minus those before applying, on the medium layer of the model which should best capture semantic information. Figure 5 shows the visualizations of the hidden state diffs using t-SNE (Maaten and Hinton,



(a) naming vs. spacing (b) naming rewrite rules (c) spacing rewrite rules

Figure 5: Visualizations of the hidden state diffs using t-SNE (Maaten and Hinton, 2008).

Table 8: Overall sensitivity on Python benchmarks under the original pre-tokenizer vs. our Python-lexer pre-tokenizer.

Rule	Model	Original	Lexer
Naming	Qwen-1.5B	7.97	7.87
	Qwen-7B	6.03	8.84
	Qwen-32B	7.77	11.18
Spacing	Qwen-1.5B	5.41	4.89
	Qwen-7B	7.90	6.64
	Qwen-32B	4.89	4.94

2008). We observe that the diffs of naming and spacing rewrite rules are clearly distinguishable (Figure 5a), so are the diffs of naming (Figure 5b) and spacing rewrite rules (Figure 5c, note that S17 and S18 are excluded since they are supersets of other rewrite rules). This confirms that the hidden states, especially from the middle layers, are good representations of semantic information and may be utilized to mitigate the tokenization changes.

5.3 A Control Study: Grammar-Aware Pre-Tokenization

The analyses in Sections 5.1 and 5.2 suggest that sensitivity arises because subword boundaries are driven by corpus frequency rather than PL grammar. A natural follow-up question is causal: if we *force* grammar-aligned boundaries at the input stage, does the sensitivity drop? We answer this with a targeted control experiment on the Qwen2.5-Coder series, in which we insert a **Python-lexer pre-tokenizer** at the front of the pre-tokenization pipeline while keeping the same subword vocabulary and decoder. Implementation details and the full per-rule results are deferred to Appendix D.5.

Table 8 reports overall sensitivity. The lexer pre-tokenizer modestly reduces spacing sensitivity on Qwen-1.5B and Qwen-7B and leaves Qwen-32B roughly unchanged, while naming sensitivity does not decrease and in fact increases on Qwen-7B and Qwen-32B (6.03%→8.84%; 7.77%→11.18%). To localize where the gains and losses concentrate, we next stratify by identifier fragment change.

Table 9: Sensitivity on Python benchmarks under the original pre-tokenizer vs. our Python-lexer pre-tokenizer, stratified by identifier fragment change on the original tokenizer.

		Unchanged		Changed	
Rule	Model	Orig.	Lexer	Orig.	Lexer
Naming	Qwen-1.5B	7.53	7.36	8.54	8.54
	Qwen-7B	5.65	7.19	6.52	11.01
	Qwen-32B	6.51	9.08	9.44	13.93
Spacing	Qwen-1.5B	5.04	4.79	7.75	5.52
	Qwen-7B	7.33	6.34	11.56	8.54
	Qwen-32B	4.77	5.06	5.65	4.20

On the fragment-changed subset (Table 9), spacing sensitivity decreases consistently (e.g., Qwen-7B: 11.56%→8.54%; Qwen-32B: 5.65%→4.20%). In contrast, naming sensitivity does *not* uniformly decrease; for the medium and large models it actually increases (Qwen-7B: 6.52%→11.01%; Qwen-32B: 9.44%→13.93%).

A plausible explanation is that lexing already isolates many operator/identifier boundaries, so inserting spaces later induces fewer additional unpredictable token-boundary changes for the downstream subword tokenizer. Naming rewrites, however, do not just perturb boundaries: they rename the identifier itself, so any gain from grammar-aligned boundaries is offset by the model's reliance on subword patterns learned during pretraining. Taken together, these results indicate that imposing grammar-aligned boundaries only at inference time is insufficient. Improving consistency and robustness is likely to require incorporating grammar-aware segmentation or equivalent boundary information during training or tokenizer design.

6 Related Work

Tokenization Most modern LLMs use subword tokenizers such as BPE (Sennrich et al., 2016),

which create vocabularies based on how often character sequences occur together. The resulting token types do not always correspond to meaningful words or code elements, and can vary depending on how the tokenizer was trained. For example, Liu et al. (2025) shows that allowing token merges across whitespace boundaries produces more meaningful units, compared to tokenizers that always split at spaces. Chirkova and Troshin (2023) introduces a tokenizer designed to better align with PL syntax, achieving lower token counts while preserving model performance. These studies show that tokenization can influence how well a model understands and generates code, and our work builds on this line of inquiry by quantifying the effects of semantic-preserving tokenization changes.

Robustness to Representation Variations Another important question is how robust LLMs are to variations in tokenization and representation at inference time. Zheng et al. (2025) show that instruction-tuned models can often retain high performance even when inputs are tokenized in unconventional or character-level formats, suggesting that such models may learn generalizable internal representations. However, their study also shows a measurable performance drop compared to standard tokenizations, and other work highlights further limitations. Wang et al. (2025) find that adversarial changes to token boundaries can significantly degrade model predictions, especially in models that have not undergone instruction tuning. In structured domains like chemistry, Yan et al. (2025) demonstrate that LLMs produce inconsistent outputs across semantically equivalent molecular representations. These findings suggest that LLMs remain sensitive to surface-level variations. Our work contributes to this line by focusing specifically on PLs. Similar misalignment may also arise in natural language when subword segmentation cuts across syntactical/grammatical boundaries, suggesting future work on studying tokenization misalignment beyond code.

Syntax-Aware Code Modeling To address the mismatch between subword tokenization and PL grammar, several approaches incorporate grammar constraints into the LLM decoding process. Synchromesh (Poesia et al., 2022) and PICARD (Scholak et al., 2021) enforce syntactic validity at generation time by using runtime parsing to filter out invalid token continuations. SynCode (Ugare et al., 2024) improves the efficiency of such methods by constructing a DFA-based mask that precomputes token legality while explicitly handling partial tokens. Boundless BPE (Schmidt et al., 2025) removes fixed pretokenizers and enables dynamic boundary selection, allowing the model to learn tokens that correspond to syntactic or semantic units. Together, these efforts aim to align LLM outputs more closely with formal code structure, a disconnect that our work quantifies by measuring how semantics-preserving tokenization variations affect model behavior.

7 Conclusions

This work studies the tokenization misalignment between subword-based LLMs and PL grammar. While subword tokenizers like BPE are widely used in code LLMs, they segment inputs based on frequency statistics, not grammar, leading to token boundaries that may not align with syntactic units in code. Through a suite of semantic-preserving rewrite rules, our framework TOKDRIFT shows that even minor formatting changes, such as whitespace edits or identifier renamings, can cause substantial shifts in model outputs. These effects hold across nine coding LLMs and three tasks (fixing, summarization, and translation). These findings motivate future research for grammar-aware or domain-adaptive tokenizers that more faithfully reflect PL structure.

Limitations

While our study shows limitations of current tokenizer designs in code LLMs, our analysis focuses on a targeted set of semantic-preserving rewrites based on common formatting and naming conventions; these do not encompass all potential sources of tokenization drift. Second, although we evaluate nine widely used code LLMs, our findings may not generalize to models with fundamentally different architectures (e.g., state space models (Gu et al., 2022)) or tokenization strategies (e.g., character-level or grammar-driven tokenizers (Kim et al., 2016)). Third, our work centers on measurement and diagnosis: beyond the controlled pre-tokenization study in Section 5.3, we do not systematically explore mitigation strategies. Future work could investigate tokenizer retraining, ensemble decoding over multiple tokenizations, or architectural modifications to improve the alignment between token boundaries and programming language syntax.

Acknowledgments

We sincerely thank Bihui Jin, Ruihan Lin, Yu Liu, Hongxu Xu, and the anonymous reviewers for their valuable comments and feedback. This work was supported in part by Compute Ontario (computeontario.ca) and the Digital Research Alliance of Canada (alliancecan.ca). This work was partially supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2024-04909, Discovery Grant RGPIN-2024-05178, and start-up grant from the University of Waterloo.

References

Wasi Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A parallel corpus for Java-Python program translation. In *Findings of the Association for Computational Linguistics: ACL*, pages 2268–2281.

Spurthi Amba Hombaiah, Tao Chen, Mingyang Zhang, Michael Bendersky, and Marc Najork. 2021. Dynamic language models for continuously evolving content. In *International Conference on Knowledge Discovery and Data Mining*, pages 2514–2524.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Nadezhda Chirkova and Sergey Troshin. 2023. Codebpe: Investigating subtokenization options for large language model pretraining on source code. *Preprint*, arXiv:2308.00683.

William Jay Conover. 1999. *Practical nonparametric statistics*. john wiley & sons.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Albert Gu, Karan Goel, and Christopher Ré. 2022. Efficiently modeling long sequences with structured state spaces. In *The International Conference on Learning Representations (ICLR)*.

Batu Guan, Xiao Wu, Yuanyuan Yuan, and Shaohua Li. 2025. Is your benchmark (still) useful? dynamic benchmarking for code language models. *arXiv preprint arXiv:2503.06643*.

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. *Preprint*, arXiv:2401.14196.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *International Conference on Program Comprehension*, pages 200–210.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander Rush. 2016. Character-aware neural language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).

Alisa Liu, Jonathan Hayase, Valentin Hofmann, Sewoong Oh, Noah A. Smith, and Yejin Choi. 2025. Superbpe: Space travel for language models. *Preprint*, arXiv:2503.13423.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605.

Meta FAIR CodeGen Team. 2025. Cwm: An open-weights llm for research on code generation with world models. Technical report, Meta. 32B-parameter open-weights model; inference code and weights released.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *Preprint*, arXiv:2112.00114.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to update natural language comments based on code changes. In *Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *Preprint*, arXiv:2201.11227.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Andrew M Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan D Tracey, and David D Cox. 2019. On the information bottleneck theory of deep learning. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124020.

Craig W. Schmidt, Varshini Reddy, Chris Tanner, and Yuval Pinter. 2025. Boundless byte pair encoding: Breaking the pre-tokenization barrier. *Preprint*, arXiv:2504.00178.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5149–5152.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *International Conference on Software Engineering*, pages 25–36.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Syncode: Llm generation with grammar augmentation. *Preprint*, arXiv:2403.01632.

Dixuan Wang, Yanda Li, Junyuan Jiang, Zepeng Ding, Ziqin Luo, Guochao Jiang, Jiaqing Liang, and Deqing Yang. 2025. Tokenization matters! degrading large language models through challenging their tokenization. *Preprint*, arXiv:2405.17067.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Bing Yan, Angelica Chen, and Kyunghyun Cho. 2025. Inconsistency of llms in molecular representations. *Digital Discovery*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Brian Siyuan Zheng, Alisa Liu, Orevaoghene Ahia, Jonathan Hayase, Yejin Choi, and Noah A. Smith. 2025. Broken tokens? your language model can secretly handle non-canonical tokenizations. *Preprint*, arXiv:2506.19004.

A Use of LLMs

We used an LLM-based writing assistant to polish grammar. All ideas, analyses, experiments, and scientific claims are our own, and we take full responsibility for the content of this work.

B Additional Background: Tokenizer Differences Between LLMs

Figure 6 shows the heatmap of vocabulary distances between tokenizers, which includes 19 popular open-source (coding) LLMs from 8 model families. Notably, most LLMs adopt a pre-tokenization strategy that splits text into linguistically and layout-meaningful chunks before a byte-level BPE. While details vary by family, common choices include isolating short digit runs (often 1–3; Qwen and some DeepSeek variants prefer per-digit), treating contiguous letters with combining marks as words, splitting punctuation and symbol runs (sometimes with an optional leading space), and separating newline blocks and longer space runs. Non-Latin scripts such as Han/Hiragana/Katakana (and in some cases Hangul) are taken as contiguous spans. Family differences that matter for our study include LLaMA-3 explicitly detaching English clitics, CodeQwen-1.5 disabling pre-tokenization (leaving underscores and long ASCII spans intact), DeepSeek-Coder using code-oriented splits (letters, punctuation, newlines, CJK, digits), and DeepSeek-V3/LLaMA-4/GPT-OSS converging on a similar unified scheme. In practice, more aggressive pre-segmentation tends to make models tolerant to superficial spacing around symbols but sensitive to numeric chunk boundaries, whereas byte-only or lightly pre-segmented designs make underscore and identifier edits more likely to introduce new token boundaries.

C Additional Experimental Methodology

C.1 Benchmarks Normalization

To ensure that our semantic-preserving naming/spacing rewrites (Section 3.3) do not spuriously break compilation or tests, we perform a lightweight normalization pass before evaluation.

For the bug fixing and code summarization tasks from HumanEvalPack (Muennighoff et al., 2023), we first canonicalize Java identifier style to `camelCase` from `snake_case`[2], then propagate any renamings consistently to tests, entry points, and declarations to preserve their functionalities.

For the code translation tasks, we start from the Avatar and CodeNet benchmarks prepared by Pan et al. (2024), following their task definitions and tests. We fixed some samples with harness-compatibility issues that would otherwise cause false negatives and prune a small number of unsalvageable or pathological samples (e.g., extremely long inputs or cases that time out), without changing the underlying problem semantics. And finally, we dropped 6 `python2java` tasks and 4 `java2python` tasks in Avatar that we could not fix.

The most common adjustments fall into a few categories: (i) IO/formatting normalization. For example, we replace non-portable characters such as U+FFFD or segmentation markers like U+2581 with ASCII equivalents; ensure consistent tokenization by splitting on spaces instead of empty strings; remove trailing spaces/newlines; standardize numeric output with Java DecimalFormat or Python f-strings to fixed precision; (ii) test correctness fixes where expected outputs were inconsistent with the reference implementation or ordering; and (iii) minimal code-context edits that preserve semantics but align with tests (e.g., renaming helper methods where tokenizer-specific splits would otherwise occur, adding @Override annotations, or make Scanner/FastScanner usage consistent). All edits are specified once, applied uniformly to baseline and variant inputs, and never conditioned on model outputs. And finally, in the normalized benchmark, we checked that the rewritten input code is compilable and could pass the tests after applying each rewrite rule, confirming that the rewrite rules are semantic-preserving.

C.2 Rewrite Algorithms

To mutatively rewrite a code context on naming, we first parse it to obtain a code token index and two identifier sets: (i) immutable identifiers derived from configured immutable types (e.g., Java: `importDeclaration`, `methodCall`; Python: `import_as_name`, `trailer`); (ii) declaration identifiers that are safe to rename (excluding Java methods annotated with `@Override`). We restrict candidates by casing using regexes, specifically, snake case matches `[a-z0-9]+(?:_[A-Za-z0-9]+)+` and camel case identifier matches the regex `[a-z]+(?:[A-Z]+[A-Za-z0-9]+[A-Za-z0-9]*)+`.

[2]HumanEvalPack (Muennighoff et al., 2023) translates the HumanEval (Chen et al., 2021) benchmark from Python to other PLs (including Java), but all the identifiers were remained in `snake_case` regardless of the target PL.

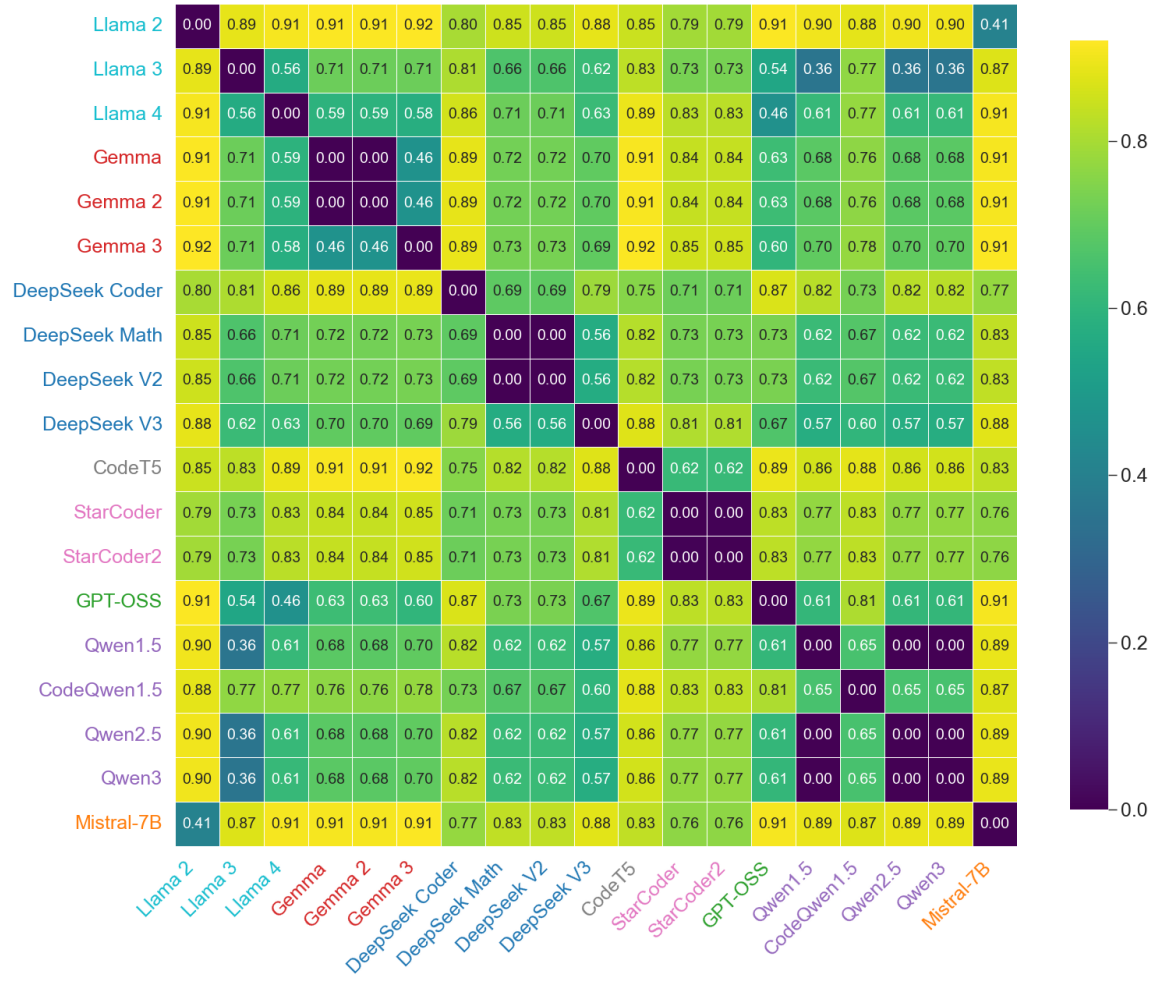


Figure 6: Heatmap of vocabulary distances between tokenizers (Full ver.).

For each eligible identifier, we segment its lexeme by a well designed regex, convert from the source to the target case, and record the absolute character positions in the original string where underscores would be inserted or removed (edit events). For HumanEval tasks, we additionally propagate the same renamings to tests, entry points, and declarations to keep the harness consistent, and these are treated as optional ancillary patches and do not alter the core algorithm. The immutable/declaration settings aim to maximize safe coverage while preserving compilation and test pass behavior.

Spacing rewrite follows the same structure but, instead of changing identifier lexemes, we insert exactly one space between adjacent tokens when their kinds match a configured token-type bigram (former, latter) from Table 3. For each match, we insert whitespace at the boundary between the two tokens, record an insertion event at that position, and update offsets.

Conceptually, although rewrites are defined over PL tokens, the notion of a fragment uses LLM tokens. For each rewrite site, we consider the minimal contiguous list of LLM tokens that covers the affected PL tokens (identifiers for naming and the two code tokens of each combination for spacing) as the fragment. Our fragment-change classification is based on an analysis of all fragments' transformation in a code context. Specifically, a merge occurs when at least one old LLM token boundary inside those spans disappears, and a split occurs when at least one new boundary appears after rewriting. To detect and analyze all LLM token boundary transformations, we compute LLM token start positions before and after rewriting with the same LLM tokenizer. We ignore boundaries created exactly at an edit site between two code tokens or those created right next to the edit site within one code token. Meaning that for insertions, we disregard any boundary introduced by the inserted whitespace between two code tokens that were fully or partially combined into one LLM token, as well

Algorithm 1 Naming Rewrite

Input: C: code context, P: code parser, I_{types}: immutable identifier types, ρ_{src}: source case regex, tgt: target case, (optional) $ExtraPatches$: extra patches.
Output: C', E, (optional) $ExtraPatches'$.

// TokIdx: list of $(x, \tau, [i, j))$ where x=code token, τ=token kind, $[i, j)$=char span
(TokIdx, S_{im}, S_{dec}) $\leftarrow$ INDEX(C, P, I_{types})
$E \leftarrow [\,],\ R \leftarrow \emptyset,\ C' \leftarrow C,\ O \leftarrow 0$ // E: list of edit underscore events (pos, δ); O: total offset

for $(x, \tau, [i, j)) \in$ TokIdx in ascending i **do**
  if $\tau =$ id $\wedge\ (x \notin S_{im} \vee x \in S_{dec}) \wedge$ REGEXCHECK(x, ρ_{src}) **then**
    $y \leftarrow$ CASECONV(x, tgt) // rewrite the identifier to the target case
    $\Delta_{list} \leftarrow$ DIFFUNDERLINEPOS(x, y, i) // return a list of add/del underscore events (pos, δ)
    $E \leftarrow$ APPEND$(E,\ \Delta_{list})$
    $R[x] \leftarrow y$
    $C' \leftarrow$ CONCAT$(C'[0{:}(i{+}O)],\ y,\ C'[(i{+}O{+}|x|){:}|C'|])$ // string concatenation
    $O \leftarrow O + (|y| - |x|)$

$ExtraPatches' \leftarrow$ APPLYREWRITES$(ExtraPatches, R)$
return $(C',\ E,\ ExtraPatches')$

Algorithm 2 Spacing Rewrite

Input: C: code context, P: code parser, (K_f, K_ℓ): token-type bigram.
Output: C', E.

// TokIdx: list of $(x, \tau, [i, j))$ where x=code token, τ=token kind, $[i, j)$=char span
TokIdx $\leftarrow$ INDEX(C, P)
$E \leftarrow [\,],\ C' \leftarrow C,\ O \leftarrow 0$ // E: list of insert events $(pos, +1)$; O: total offset

for $k \leftarrow 0$ **to** |TokIdx| $- 2$ **do**
  $(x_f, \tau_f, [i_f, j_f)) \leftarrow$ TokIdx$[k]$; $(x_\ell, \tau_\ell, [i_\ell, j_\ell)) \leftarrow$ TokIdx$[k{+}1]$
  if MATCH$(\tau_f, K_f)\ \wedge\ $ MATCH(τ_ℓ, K_ℓ) **then**
    $E \leftarrow$ APPEND$(E,\ (i_\ell, +1))$
    $C' \leftarrow$ CONCAT$\big(C'[0{:}(j_f{+}O)],$ "␣", $C'[(i_\ell{+}O){:}|C'|]\big)$ // insert one space
    $O \leftarrow O + 1$

return $(C',\ E)$

as those caused by standalone underscores immediately to its right (a behavior commonly observed in the DeepSeek-Coder or CodeQwen-1.5 tokenizer, where underscores are usually treated as a single token), as encoded by the various edit masks classified by edit types in Algorithm 3. The loop in Algorithm 3 shifts the original boundary set by the cumulative δ pre edit to align coordinate systems, builds the masks for shift edits and edit-adjacent positions for insert operation, and then compares adjusted old versus filtered new starts. Specifically, let S_{old} and S_{new} be the sets of LLM token starts before and after rewriting, and after masking specific edit sites, we compute $A{=}S_{old} \setminus S_{new}$ (old boundaries lost) and $B{=}S_{new} \setminus S_{old}$ (new boundaries gained). The label is `unchanged` if $A{=}\emptyset$ and $B{=}\emptyset$, `merged` if $A{\neq}\emptyset$ and $B{=}\emptyset$, `split` if $A{=}\emptyset$ and $B{\neq}\emptyset$, and `mixed` otherwise.

C.3 Metrics Computation Algorithm

We evaluate on two input programming language subsets for accuracy and Δaccuracy, X_p (Python inputs) and X_j (Java inputs), where their union $X = X_j \cup X_p$ with $|X| = 1546$. For a fixed rewrite rule w_i and model m, let T_i be the deterministic transformation that applies w_i to an input $x \in X$, and we define T_0 as no rule would be applied on the input. And we let $W{=}\{i | i = 0, 1, \cdots, 24\}$

Algorithm 3 Fragment-Change Classification (CLASSIFY)

Input: C: original code context, C': new code context, E: list of edit events (pos, δ) with $\delta \in \{\pm 1\}$, $EditType$: edit type, T: LLM tokenizer.
Output: $type \in \{\texttt{unchanged}, \texttt{merged}, \texttt{split}, \texttt{mixed}\}$

$L^{old} \leftarrow \text{POSLLMTOKENS}(C, T)$ // cumulative first character positions of LLM tokens in C
$L^{new} \leftarrow \text{POSLLMTOKENS}(C', T)$
$S_{old} \leftarrow \text{SET}(L^{old})$, $S_{new} \leftarrow \text{SET}(L^{new})$, $S_{ed} \leftarrow \{\, pos \mid (pos, \delta) \in E \,\}$, $S^{+}_{ed} \leftarrow \emptyset$
$O \leftarrow 0$ // cumulative offset from prior edits

for each (pos, δ) in E **do**
    $a \leftarrow pos + O$ // adjusted position of this edit
    $S_{old} \leftarrow \{\, p{+}\delta$ **if** $p > a$ **else** $p \mid p \in S_{old} \,\}$
    $S_{ed} \leftarrow \{\, e{+}\delta$ **if** $e > a$ **else** $e \mid e \in S_{ed} \,\}$
    $S^{+}_{ed} \leftarrow S^{+}_{ed} \cup \{a + \max(\delta, 0)\}$
    $O \leftarrow O + \delta$
if $EditType$ = underscore **then**
    $S_{new} \leftarrow S_{new} \setminus (S^{+}_{ed} \setminus S_{ed})$ // ignore starts next-to inserted standalone underscore edit boundaries
else if $EditType$ = whitespace **then**
    $S_{new} \leftarrow S_{new} \setminus (S_{ed} \setminus S_{old})$ // ignore new starts created at whitespace edit boundaries

$A \leftarrow S_{old} \setminus S_{new}$ // A: lost tokens after rewrite (some tokens merged)
$B \leftarrow S_{new} \setminus S_{old}$ // B: gained tokens after rewrite (some tokens split)

if $A \neq \emptyset$ **and** $B = \emptyset$ **then**
    return `merged`
else if $A = \emptyset$ **and** $B \neq \emptyset$ **then**
    return `split`
else if $A \neq \emptyset$ **and** $B \neq \emptyset$ **then**
    return `mixed`
else
    return `unchanged`

denote the assignment set for all rules where $i{=}0$ is the baseline, $i{>}0$ means the variant of applying rule w_i. Running the model yields code $f_m(T_i(x))$, which the harness evaluates on the test set $\mathcal{T}(x)$. We define the test-level pass fraction

$$r_{m,i}(x) \triangleq \frac{1}{|\mathcal{T}(x)|} \sum_{t \in \mathcal{T}(x)} [[f_m(T_i(x))]]_t,$$

where $[[f_m(T_i(x))]]_t \in \{0, 1\}$ denotes the execution result of program $f_m(T_i(x))$ from test t (Guan et al., 2025). Follow that we define the task-level correctness indicator $Y_{m,i}(x) \triangleq \mathbb{I}\{r_{m,i}(x) = 1\} \in \{0, 1\}$. So the accuracy of a rule assignment $i \in W$ on a set $S \in \{X_p, X_j\}$ is

$$\text{Accuracy}_i(m; S) = \frac{1}{|S|} \sum_{x \in S} Y_{m,i}(x).$$

We report Δaccuracy as $\Delta\text{accuracy}_i(m; S) \triangleq \text{Accuracy}_i(m; S) - \text{Accuracy}_0(m; S)$, where $i \in W$ and $i \neq 0$. Not all inputs would be modified by a given rule, we therefore define the actually-affected subset

$$X'_i \triangleq \{\, x \in X : T_i(x) \neq x \,\},$$

whose summed sizes for all rules classified by model series are shown in Table 23. Then our proposed sensitivity measures how often correctness flips among affected inputs only by

$$\text{Sensitivity}_i(m) \triangleq \frac{1}{|X'_i|} \sum_{x \in X'_i} \left| Y_{m,i}(x) - Y_{m,0}(x) \right|.$$

Intuitively, Δaccuracy captures net gains/losses which may cancel when aggregating, whereas sensitivity isolates the flip rate on inputs whose tokens were actually changed by w_i.

Table 10: Compilability and Δcompilability (in parenthesis) of each model on each rewrite rule.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
Input PL = Java										
baseline	65.37	69.90	78.42	61.63	73.64	84.24	71.71	79.33	76.61	73.43
N1	65.25 (-0.12)	70.03 (+0.13)	78.04 (-0.38)	61.89 (+0.26)	73.39 (-0.25)	84.24 (+0.00)	72.87 (+1.16)	79.59 (+0.26)	76.87 (+0.26)	73.57 (+0.14)
N2	65.50 (+0.13)	70.28 (+0.38)	77.78 (-0.64)	61.76 (+0.13)	**74.29** (+0.65)	84.37 (+0.13)	73.00 (+1.29)	79.46 (+0.13)	76.61 (+0.00)	73.67 (+0.24)
N3	66.28 (+0.91)	70.41 (+0.51)	77.52 (-0.90)	62.02 (+0.39)	73.77 (+0.13)	82.82 (-1.42)	72.09 (+0.38)	78.94 (-0.39)	76.36 (-0.25)	73.36 (-0.07)
S3	66.15 (+0.78)	70.54 (+0.64)	77.78 (-0.64)	62.27 (+0.64)	73.64 (+0.00)	83.98 (-0.26)	71.32 (-0.39)	79.72 (+0.39)	76.87 (+0.26)	73.59 (+0.16)
S6	65.37 (+0.00)	69.25 (-0.65)	**78.81** (+0.39)	61.63 (+0.00)	73.51 (-0.13)	84.37 (+0.13)	72.61 (+0.90)	79.72 (+0.39)	76.49 (-0.12)	73.53 (+0.10)
S8	65.37 (+0.00)	69.12 (-0.78)	78.17 (-0.25)	61.89 (+0.26)	72.61 (-1.03)	84.24 (+0.00)	73.13 (+1.42)	79.07 (-0.26)	76.87 (+0.26)	73.39 (-0.04)
S9	66.15 (+0.78)	68.99 (-0.91)	78.68 (+0.26)	60.72 (-0.91)	73.90 (+0.26)	83.07 (-1.17)	71.45 (-0.26)	79.84 (+0.51)	76.74 (+0.13)	73.28 (-0.15)
S11	66.41 (+1.04)	**71.83** (+1.93)	76.74 (-1.68)	**62.40** (+0.77)	73.90 (+0.26)	84.63 (+0.39)	72.48 (+0.77)	79.46 (+0.13)	75.84 (-0.77)	73.74 (+0.31)
S12	64.99 (-0.38)	70.41 (+0.51)	77.39 (-1.03)	61.63 (+0.00)	73.39 (-0.25)	79.84 (-4.40)	73.13 (+1.42)	79.84 (+0.51)	75.58 (-1.03)	72.91 (-0.52)
S13	66.93 (+1.56)	69.38 (-0.52)	77.52 (-0.90)	61.11 (-0.52)	73.13 (-0.51)	83.33 (-0.91)	72.87 (+1.16)	79.72 (+0.39)	76.49 (-0.12)	73.39 (-0.04)
S14	64.21 (-1.16)	68.09 (-1.81)	75.45 (-2.97)	59.82 (-1.81)	73.39 (-0.25)	83.85 (-0.39)	**73.26** (+1.55)	78.94 (-0.39)	76.49 (-0.12)	72.61 (-0.82)
S15	**67.05** (+1.68)	66.15 (-3.75)	75.97 (-2.45)	59.69 (-1.94)	74.16 (+0.52)	**84.88** (+0.64)	72.22 (+0.51)	78.42 (-0.91)	**77.52** (+0.91)	72.90 (-0.53)
S16	65.37 (+0.00)	68.48 (-1.42)	77.26 (-1.16)	61.63 (+0.00)	73.13 (-0.51)	84.75 (+0.51)	73.00 (+1.29)	**80.23** (+0.90)	76.61 (+0.00)	73.38 (-0.05)
S17	64.99 (-0.38)	66.80 (-3.10)	76.10 (-2.32)	61.89 (+0.26)	72.09 (-1.55)	84.63 (+0.39)	73.00 (+1.29)	78.94 (-0.39)	77.13 (+0.52)	72.84 (-0.59)
S18	61.50 (-3.87)	65.76 (-4.14)	77.13 (-1.29)	59.82 (-1.81)	73.77 (+0.13)	83.72 (-0.52)	71.45 (-0.26)	77.39 (-1.94)	75.06 (-1.55)	71.73 (-1.70)
Input PL = Python										
baseline	64.38	75.00	83.03	62.44	82.12	88.60	72.67	81.74	84.46	77.16
N4	64.25 (-0.13)	75.39 (+0.39)	83.16 (+0.13)	61.92 (-0.52)	82.64 (+0.52)	**90.28** (+1.68)	72.28 (-0.39)	81.61 (-0.13)	84.84 (+0.38)	77.37 (+0.21)
N5	62.95 (-1.43)	74.74 (-0.26)	82.77 (-0.26)	61.40 (-1.04)	82.38 (+0.26)	90.28 (+1.68)	71.76 (-0.91)	81.48 (-0.26)	84.46 (+0.00)	76.91 (-0.25)
N6	63.47 (-0.91)	74.48 (-0.52)	80.57 (-2.46)	60.75 (-1.69)	82.12 (+0.00)	89.25 (+0.65)	70.34 (-2.33)	81.48 (-0.26)	83.94 (-0.52)	76.27 (-0.89)
S1	64.51 (+0.13)	75.65 (+0.65)	82.64 (-0.39)	62.05 (-0.39)	82.25 (+0.13)	89.12 (+0.52)	72.41 (-0.26)	82.25 (+0.51)	84.20 (-0.26)	77.23 (+0.07)
S2	64.77 (+0.39)	75.52 (+0.52)	83.03 (+0.00)	62.31 (-0.13)	**82.77** (+0.65)	88.47 (-0.13)	71.63 (-1.04)	81.99 (+0.25)	84.20 (-0.26)	77.19 (+0.03)
S4	**64.90** (+0.52)	**75.91** (+0.91)	82.90 (-0.13)	62.69 (+0.25)	82.51 (+0.39)	86.01 (-2.59)	72.28 (-0.39)	81.74 (+0.00)	84.20 (-0.26)	77.02 (-0.14)
S5	63.86 (-0.52)	75.39 (+0.39)	82.90 (-0.13)	61.79 (-0.65)	81.87 (-0.25)	89.25 (+0.65)	72.02 (-0.65)	81.99 (+0.25)	83.55 (-0.91)	76.96 (-0.20)
S7	64.51 (+0.13)	74.61 (-0.39)	82.64 (-0.39)	62.69 (+0.25)	81.09 (-1.03)	89.12 (+0.52)	71.89 (-0.78)	**82.51** (+0.77)	84.33 (-0.13)	77.04 (-0.12)
S10	63.73 (-0.65)	73.58 (-1.42)	**83.55** (+0.52)	**62.95** (+0.51)	82.12 (+0.00)	90.03 (+1.43)	71.24 (-1.43)	81.87 (+0.13)	84.33 (-0.13)	77.04 (-0.12)
S13	63.73 (-0.65)	73.83 (-1.17)	83.55 (+0.52)	62.18 (-0.26)	82.64 (+0.52)	88.60 (+0.00)	71.89 (-0.78)	81.22 (-0.52)	83.42 (-1.04)	76.78 (-0.38)
S14	64.38 (+0.00)	74.35 (-0.65)	83.29 (+0.26)	60.10 (-2.34)	81.22 (-0.90)	86.53 (-2.07)	72.28 (-0.39)	81.09 (-0.65)	83.94 (-0.52)	76.35 (-0.81)
S15	64.25 (-0.13)	75.39 (+0.39)	81.87 (-1.16)	61.01 (-1.43)	80.05 (-2.07)	88.73 (+0.13)	72.41 (-0.26)	80.44 (-1.30)	84.46 (+0.00)	76.51 (-0.65)
S16	63.73 (-0.65)	73.32 (-1.68)	83.03 (+0.00)	60.23 (-2.21)	80.83 (-1.29)	89.64 (+1.04)	71.76 (-0.91)	79.79 (-1.95)	**84.97** (+0.51)	76.37 (-0.79)
S17	63.73 (-0.65)	73.70 (-1.30)	81.87 (-1.16)	60.10 (-2.34)	79.79 (-2.33)	89.64 (+1.04)	71.50 (-1.17)	79.27 (-2.47)	83.68 (-0.78)	75.92 (-1.24)
S18	61.53 (-2.85)	74.74 (-0.26)	81.09 (-1.94)	59.46 (-2.98)	80.70 (-1.42)	88.34 (-0.26)	71.11 (-1.56)	80.31 (-1.43)	84.20 (-0.26)	75.72 (-1.44)

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

C.4 Experimental Environment

We conduct all experiments on an NVIDIA H100 GPU cluster, consuming approximately 1840 GPU-hours in total across runs. All model checkpoints are obtained from the Hugging Face Hub and loaded with the Hugging Face Transformers library (v4.53.2) (Wolf et al., 2020). Unless otherwise stated, models are executed in fp32, the only exceptions are Llama-3.3-70B-Instruct, Qwen2.5-Coder-32B-Instruct, and deepseek-coder-33b-instruct, which we run in fp16. All evaluations use the bigcode-evaluation-harness framework (Ben Allal et al., 2022) with its standard protocols. Except the experiment setting in Appendix D.2, we use deterministic decoding without sampling and a batch size of 1 throughout all main experiments. All tests are executed with Java 21.0.1 and Python 3.8. The maximum generation length is set to 1,024 tokens for HumanEvalPack and Avatar tasks, and 2,048 tokens for CodeNet tasks. For t-SNE visualizations, we use scikit-learn v1.7.1 (`sklearn.manifold.TSNE`) with perplexity to 70 and use the Barnes–Hut method with 1000 iterations, PCA initialization, learning_rate='auto', and n_jobs=16 (Pedregosa et al., 2011).

D Additional Analyses and Results

D.1 Compilability Analysis

Our primary metric is accuracy (Section 3.4), which is computed from the benchmark test suites. Since compilation success is a necessary (but not sufficient) condition for passing all tests, compilation failures are already counted as incorrect and therefore lower accuracy. We therefore treat compilability as an auxiliary diagnostic rather than a separate objective. It measures how often a model's first decoded output can be compiled (Java) or syntax-checked (Python) by the evaluation harness before running any tests, helping distinguish syntax/compilation failures from other test failures. We define Δcompilability analogously to Δaccuracy, which is calculated as the compilability of the variant mi-

Table 11: Nucleus sampling with K=10: sensitivity @K on HumanEvalPack bug-fixing tasks.

Rewrite Rule	Model Series	S	M	L
Naming	Llama-3	7.26	8.67	7.37
	Qwen2.5-Coder	6.20	6.87	5.11
	DeepSeek-Coder	5.28	5.72	6.25
Spacing	Llama-3	8.06	11.57	9.34
	Qwen2.5-Coder	8.09	8.37	5.00
	DeepSeek-Coder	6.23	6.16	5.67

nus the compilability of the baseline.

Table 10 shows that tokenization-only, semantics-preserving input rewrites can still cause measurable changes in compilability. For example, for Qwen-32B, baseline compilability is 84.24% on Java and 88.60% on Python; on Python, Δcompilability ranges from -2.59% (rule S4) to +1.68% (rule N4). Across all settings in our experiments, Δcompilability is positively correlated with Δaccuracy (Spearman $\rho = 0.5804$, $p < 0.01$), intuitively indicating that rewrites that hurt compilability tend to also hurt test-based accuracy. However, there are several cases where a rule improves compilability but hurts accuracy, meaning that tokenization perturbations can change model behavior even when the output still compiles, affecting whether the generated program is correct rather than merely syntactically valid.

D.2 Greedy vs. MRD Sensitivity Comparison

In the main experiments we use greedy decoding, which yields a single deterministic output per input. To evaluate whether our sensitivity findings persist under stochastic decoding and multiple samples, we rerun the HumanEvalPack bug-fixing tasks with nucleus sampling (K=10, temperature 0.2, top-p 0.95). Tables 20 and 21 report pass@1 and pass@K, respectively, along with their deltas relative to the baseline input.

To measure sensitivity under multiple samples, we define sensitivity@K as the average change in the fraction of passing samples between the baseline input and the rewritten input, restricted to inputs actually affected by the rule. Specifically, for a model m and rule assignment i, we draw K samples $\{y^{(1)}_{m,i}(x), \ldots, y^{(K)}_{m,i}(x)\}$ for each input x under the sampling configuration above, and define the pass indicator for sample k as $Y^{(k)}_{m,i}(x) \in \{0, 1\}$ (1 iff the program passes all tests). Let the number of passing samples be $C_{m,i}(x) \triangleq \sum_{k=1}^{K} Y^{(k)}_{m,i}(x)$. Then analogously to sensitivity (Section 3.4), the sensitivity@K is defined as

$$\text{Sensitivity@}K_i(m) \triangleq \frac{1}{|X'_i|} \sum_{x \in X'_i} \frac{D_{m,i}(x)}{K},$$

where $D_{m,i}(x) \triangleq \left|C_{m,i}(x) - C_{m,0}(x)\right|$. This measures how much the number of passing samples changes under a rewrite, normalized by K. We summarize sensitivity@K by model series and size in Table 11.

To further probe whether the effect is driven by brittle decision boundaries in selecting a single output, we apply a simple minimum-risk decoding (MRD) scheme to the K sampled candidates and then evaluate the selected output as pass@1. Given a candidate set $\mathcal{Y} = \{y_1, \ldots, y_K\}$, we select

$$\hat{y} = \arg\min_{y_i \in \mathcal{Y}} \sum_{k=1}^{K} p(y_i \mid x)\, \Delta(y_k, y_i),$$

where $\Delta(\cdot, \cdot)$ is the Levenshtein distance between two candidates.

Comparing with greedy decoding in the main experiments, we report MRD-selected accuracy/Δaccuracy (Table 22), and greedy-vs-MRD sensitivity (Table 12). Table 22 indicates that, for Java, MRD yields accuracy and Δaccuracy patterns that are broadly similar to greedy decoding, while the Python results are more variable.

Table 12: Sensitivity comparison between greedy decoding and MRD-selected outputs on HumanEvalPack bug-fixing tasks.

Rewrite Rule	Model	Greedy	MRD
Naming	Llama-3B	9.94	8.55
	Llama-8B	12.70	9.06
	Llama-70B	7.04	8.18
	Qwen-1.5B	6.54	7.30
	Qwen-7B	5.66	9.18
	Qwen-32B	4.03	4.91
	DS-1.3B	3.90	4.53
	DS-6.7B	5.91	6.04
	DS-33B	5.66	7.04
Spacing	Llama-3B	8.67	8.55
	Llama-8B	14.06	11.81
	Llama-70B	10.72	9.56
	Qwen-1.5B	7.59	8.76
	Qwen-7B	7.95	12.09
	Qwen-32B	4.54	4.78
	DS-1.3B	5.66	6.14
	DS-6.7B	6.10	6.35
	DS-33B	5.58	5.86

Table 13: (C++ results on HumanEvalPack) Accuracy and Δaccuracy (in parenthesis) of each model on each applicable rewrite rule.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
baseline	22.87	27.44	46.95	31.71	58.54	71.95	28.35	48.17	49.39	42.82
N4	21.65 (-1.22)	28.96 (+1.52)	46.34 (-0.61)	31.40 (-0.31)	57.32 (-1.22)	70.12 (-1.83)	26.22 (-2.13)	45.43 (-2.74)	48.48 (-0.91)	41.77 (-1.05)
N5	21.34 (-1.53)	27.44 (+0.00)	46.95 (+0.00)	31.40 (-0.31)	55.18 (-3.36)	69.51 (-2.44)	25.30 (-3.05)	44.82 (-3.35)	49.09 (-0.30)	41.23 (-1.59)
N6	18.90 (-3.97)	25.91 (-1.53)	43.60 (-3.35)	29.88 (-1.83)	54.27 (-4.27)	67.99 (-3.96)	24.09 (-4.26)	47.26 (-0.91)	48.17 (-1.22)	40.01 (-2.81)
S12	21.65 (-1.22)	26.83 (-0.61)	47.26 (+0.31)	32.32 (+0.61)	57.01 (-1.53)	68.29 (-3.66)	27.44 (-0.91)	46.34 (-1.83)	49.09 (-0.30)	41.80 (-1.02)
S13	22.56 (-0.31)	27.74 (+0.30)	48.17 (+1.22)	**32.93** (+1.22)	57.62 (-0.92)	71.34 (-0.61)	27.74 (-0.61)	48.78 (+0.61)	48.78 (-0.61)	42.85 (+0.03)
S14	21.34 (-1.53)	27.44 (+0.00)	46.65 (-0.30)	31.40 (-0.31)	57.62 (-0.92)	71.34 (-0.61)	26.83 (-1.52)	48.48 (+0.31)	48.48 (-0.91)	42.18 (-0.64)
S16	20.43 (-2.44)	27.13 (-0.31)	46.34 (-0.61)	31.10 (-0.61)	55.79 (-2.75)	71.34 (-0.61)	28.05 (-0.30)	**49.70** (+1.53)	**50.61** (+1.22)	42.28 (-0.54)
S17	21.04 (-1.83)	28.66 (+1.22)	**49.09** (+2.14)	29.88 (-1.83)	53.96 (-4.58)	70.12 (-1.83)	26.83 (-1.52)	47.56 (-0.61)	47.87 (-1.52)	41.67 (-1.15)
S18	21.95 (-0.92)	**30.18** (+2.74)	49.09 (+2.14)	27.13 (-4.58)	55.49 (-3.05)	71.34 (-0.61)	**29.57** (+1.22)	46.04 (-2.13)	45.73 (-3.66)	41.84 (-0.98)

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

Table 12 shows no obvious downward trend in sensitivity under MRD, and in some cases it is even much larger than greedy decoding. For example, Llama-8B decreases from 12.70% to 9.06% on naming rewrites. In contrast, Qwen-7B increases from 7.95% to 12.09% on spacing rewrites. Overall, sensitivity remains non-negligible after sampling and MRD selection, suggesting that tokenization drift is not solely an artifact of greedy choice but reflects broader instability in how models map tokenized inputs to functionally correct outputs.

Table 14: (C++ results on HumanEvalPack) Impact of model size on sensitivity.

Rewrite Rule	Model Series	S	M	L
Naming	Llama-3	10.59	11.55	12.64
	Qwen2.5-Coder	7.94	12.88	13.36
	DeepSeek-Coder	9.03	11.91	8.66
Spacing	Llama-3	10.69	11.31	10.14
	Qwen2.5-Coder	10.75	12.05	9.39
	DeepSeek-Coder	6.43	9.70	7.73

D.3 C++ Results

To test whether our findings generalize beyond Java/Python, we additionally run TOKDRIFT on C++ benchmarks from HumanEvalPack, covering bug fixing and code summarization. We reuse the same evaluation protocol and apply the subset of semantic-preserving naming and spacing rewrite rules that are applicable to C++.

Table 13 shows that C++ exhibits the same tokenization-drift effect: these tokenization-only input rewrites can still change accuracy by several points. For example, Qwen-32B attains 71.95% at baseline, and drops to 67.99% (-3.96%) under N6. Table 14 further shows that sensitivity remains non-trivial across model series, such as 13.36% for naming and 9.39% for spacing, aligning with our conclusion that misaligned subword tokenization can perturb model behavior across multiple programming languages.

D.4 Token-Drift Magnitude and Correlations

We quantify the magnitude of tokenization drift using Δ#tokens, the change in the number of LLM tokens induced by applying a rewrite rule to an input. Table 18 reports the average Δ#tokens for each rewrite rule in each model series. Overall, most rewrite rules only change a small number of tokens in the input sequence, with a few spacing rules producing larger shifts because they can apply at many operator boundaries. Therefore, to test whether sensitivity is largely explained by the quantity of tokenization changes, we correlate Δ#tokens with sensitivity across samples. Table 15 shows that the correlations are generally weak to moderate (except for Qwen-7B on spacing rewrites cases), suggesting that Δ#tokens alone is not a sufficient explanation for sensitivity.

We also analyze whether sensitivity is driven by where the first edit occurs in the input. Table 16 uses the number of tokens before the first edit, and Table 17 uses the corresponding percentage relative to the full input. Across model families and sizes, we observe mostly weak to moderate correlations, indicating that sensitivity is not simply explained by how early an edit happens. This is particularly evident for naming rewrites, where the first-edited identifiers often cluster near function signatures (e.g., method names) and thus offer limited location variance.

Taken together, these results support our main claim that sensitivity arises from which token boundaries are perturbed and the resulting changes in early representations, not merely from the raw number of tokens added/removed. More fundamentally, the model's ability to represent and reason about code is shaped and sometimes hindered by

Table 15: Spearman correlation coefficient between Δ#tokens and sensitivity.

Rewrite Rule	Metric	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B
Naming	r_s	-0.095	0.278	0.428	0.126	0.212	0.478	-0.211	0.458	0.114
	p-value	0.657	0.188	< 0.05	0.559	0.319	< 0.05	0.322	< 0.05	0.597
Spacing	r_s	0.225	0.292	0.150	0.298	0.569	0.233	0.316	0.317	0.278
	p-value	< 0.05	< 0.01	0.145	< 0.01	< 0.001	< 0.05	< 0.01	< 0.01	< 0.01

Table 16: Spearman correlation coefficient between first edit token location and sensitivity.

Rewrite Rule	Metric	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B
Naming	r_s	-0.481	-0.403	-0.420	-0.098	0.142	0.135	-0.073	-0.180	-0.188
	p-value	< 0.05	0.051	< 0.05	0.647	0.509	0.530	0.736	0.400	0.379
Spacing	r_s	-0.529	-0.676	-0.710	-0.456	-0.349	-0.425	-0.354	-0.351	-0.386
	p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

Table 17: Spearman correlation coefficient between percentage of tokens before the first edit location and sensitivity.

Rewrite Rule	Metric	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B
Naming	r_s	-0.383	-0.413	-0.609	-0.212	0.131	-0.169	0.232	-0.378	-0.059
	p-value	0.064	< 0.05	< 0.01	0.319	0.540	0.430	0.275	0.069	0.785
Spacing	r_s	-0.340	-0.559	-0.586	-0.372	-0.326	-0.359	-0.155	-0.262	-0.275
	p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.01	< 0.001	0.132	< 0.05	< 0.01

Table 18: Average number of tokens altered by rewrite rules (Δ#tokens) for each model series.

Variant	Llama-3	Qwen2.5-Coder	DS-Coder	**Average**
N1	0.99	0.99	5.77	2.59
N2	0.78	0.78	0.76	0.77
N3	3.78	3.78	14.82	7.46
N4	-0.29	-0.29	-4.77	-1.78
N5	0.15	0.15	-4.08	-1.26
N6	2.69	2.69	6.50	3.96
S1	1.53	1.53	1.20	1.42
S2	2.53	2.53	2.26	2.44
S3	2.00	2.00	1.94	1.98
S4	1.60	1.60	1.61	1.60
S5	1.60	1.60	1.43	1.54
S6	1.55	1.55	1.18	1.42
S7	2.39	2.39	-0.01	1.59
S8	1.89	1.89	0.00	1.26
S9	3.36	3.36	3.34	3.35
S10	2.42	2.42	2.18	2.34
S11	6.95	6.95	6.50	6.80
S12	8.93	8.93	8.13	8.66
S13	2.43	2.43	2.35	2.40
S14	3.80	3.80	4.53	4.04
S15	7.34	7.34	0.46	5.05
S16	5.20	5.19	0.07	3.49
S17	14.33	14.33	0.51	9.72
S18	28.84	28.83	13.84	23.84

the tokenizer's segmentation, so even small token-count changes can occasionally lead to substantial behavioral drift.

D.5 Python Lexer Pre-Tokenizer Experiments

Modern Hugging Face tokenizers[3] apply tokenization in stages: a normalizer standardizes text, a pre-tokenizer splits text into chunks, a subword model (e.g., BPE) maps chunks to token IDs, and a post-processor adds special tokens. The pre-tokenizer of current open-source LLMs is typically a generic `Sequence` of regex and byte-level rules, which does not explicitly enforce programming-language grammar boundaries. As a control experiment toward a more grammar-aware boundarying, while still using the same subword vocabulary and decoder, we implement a Python lexer pre-tokenizer[4] and insert it at the beginning of the pre-tokenizer sequence. Concretely, it uses a Python lexer[5] to split code into lexical-token-aligned spans (attaching intervening whitespace to neighboring spans so the original text is preserved) and then passes these spans to the original pre-tokenization and BPE stages. The control experiment that motivates this study is presented in Section 5.3; here we report the full experimental details and results.

We rerun all Python experiments on the Qwen2.5-Coder models with this lexer pre-tokenizer; Table 19 reports the full per-rule accuracy, and the main-text analysis of sensitivity under this setting is given in Section 5.3. For completeness, we also report a cross-scheme baseline comparison between Table 4 (original pre-tokenizer) and Table 19 (lexer pre-tokenizer): the Python baseline accuracy of Qwen-1.5B decreases from 40.67% to 36.01%, Qwen-7B from 64.51% to 64.12%, and Qwen-32B from 76.17% to 73.70%. This comparison studies a research question re-

[3]https://github.com/huggingface/tokenizers

[4]https://github.com/larryinx/tokenizers

[5]https://github.com/RustPython/Parser

Table 19: (Python results with a Python-lexer pre-tokenizer) Accuracy and Δaccuracy (in parenthesis) of each model on each rewrite rule. The Δ column next to each model reports the cross-scheme change, computed as the lexer-pre-tokenizer accuracy minus the original-pre-tokenizer accuracy (Table 4).

Variant	Qwen-1.5B	Δ	Qwen-7B	Δ	Qwen-32B	Δ
BL	36.01	-4.66	64.12	-0.39	73.70	-2.47
N4	**36.79** (+0.78)	-2.98	64.38 (+0.26)	-0.65	**76.55** (+2.85)	-1.3
N5	36.53 (+0.52)	-2.72	64.64 (+0.52)	-0.13	75.52 (+1.82)	-2.2
N6	35.10 (-0.91)	-4.28	63.73 (-0.39)	-0.78	74.61 (+0.91)	-2.2
S1	36.53 (+0.52)	-4.01	63.99 (-0.13)	-0.52	74.09 (+0.39)	-2.59
S2	35.75 (-0.26)	-4.66	63.86 (-0.26)	-0.91	73.19 (-0.51)	-2.72
S4	35.88 (-0.13)	-4.66	63.73 (-0.39)	-0.78	70.73 (-2.97)	-2.46
S5	35.49 (-0.52)	-5.31	62.95 (-1.17)	-1.17	74.48 (+0.78)	-2.46
S7	36.01 (+0.00)	-4.66	63.60 (-0.52)	0.26	73.96 (+0.26)	-2.46
S10	36.01 (+0.00)	-4.66	**64.90** (+0.78)	0.91	76.55 (+2.85)	-0.91
S13	36.53 (+0.52)	-4.01	63.99 (-0.13)	-0.91	74.22 (+0.52)	-2.33
S14	35.49 (-0.52)	-3.89	64.12 (+0.00)	0.39	71.76 (-1.94)	-2.33
S15	36.14 (+0.13)	-3.63	63.34 (-0.78)	0.65	74.87 (+1.17)	-1.43
S16	36.14 (+0.13)	-3.5	62.69 (-1.43)	-0.39	76.04 (+2.34)	-0.64
S17	35.10 (-0.91)	-4.28	62.95 (-1.17)	1.03	75.52 (+1.82)	-1.03
S18	35.23 (-0.78)	-3.11	63.86 (-0.26)	0.78	74.35 (+0.65)	-0.78

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.
Δ = lexer pre-tokenizer accuracy − original pre-tokenizer accuracy; negative ⇒ lexer reduces accuracy.

lated to but distinct from our main focus: it measures how LLM behavior changes when the *same* input is tokenized under a *different* tokenization scheme, whereas our paper measures how behavior changes across semantically equivalent but surface-form-different code variants under a *fixed* tokenizer. Consistent with prior findings, LLMs show some robustness to different tokenization schemes, though deviations from the trained scheme still lead to baseline degradation with varying magnitude across model sizes. We include this cross-scheme analysis as a useful reference point, since both directions point to the shared open challenge of LLM robustness to tokenization variations. Together with the sensitivity findings in Section 5.3, the accuracy drop observed under the lexer pre-tokenizer further reinforces the message that simply switching the tokenizer or imposing grammar-aligned boundaries at inference time is not an ideal solution to tokenization misalignment. Mitigating the issue likely requires incorporating grammar-aware segmentation during training or tokenizer design.

D.6 Additional Plots

In Figures 7 and 8, the line plots summarize sensitivity for each rewrite rule. In Figure 9, comparing samples with and without identifier fragment change shows the overall trend of sensitivity on different model size. The per-series breakdowns in Figures 10 to 12 echo this pattern across Llama-3, Qwen2.5-Coder, and DeepSeek-Coder, while Llama tends to be more sensitive overall, all families exhibit a variation in sensitivity between "changed" and "unchanged" groups.

Figure 13 shows the distribution of Δaccuracy per rewrite rule. Compared to sensitivity, Δaccuracy may not be ideal for quantifying robustness because gains and losses cancel and many samples are unaffected.

Table 23 reports, for each rewrite rule, the number of benchmark samples actually modified, stratified by model series. Table 24 provides the full breakdown of sensitivity by fragment-change category. Together, these tables clarify both the scope of input perturbations and the source of robustness differences observed in the main results.

Table 20: Nucleus sampling (temperature 0.2, top-p 0.95): pass@1 and Δpass@1 on HumanEvalPack bug-fixing tasks.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
Input PL = Java										
baseline	19.33	38.17	41.04	32.44	59.09	68.48	30.12	51.83	57.62	44.23
N1	19.45 (+0.12)	38.78 (+0.61)	41.59 (+0.55)	33.41 (+0.98)	61.46 (+2.38)	66.95 (-1.52)	28.90 (-1.22)	52.20 (+0.37)	55.79 (-1.83)	44.28 (+0.05)
N2	19.70 (+0.37)	38.17 (+0.00)	40.91 (-0.12)	31.77 (-0.67)	60.24 (+1.16)	66.89 (-1.59)	30.55 (+0.43)	52.62 (+0.79)	55.79 (-1.83)	44.07 (-0.16)
N3	18.54 (-0.79)	37.80 (-0.37)	43.41 (+2.38)	30.85 (-1.59)	60.49 (+1.40)	68.11 (-0.37)	29.27 (-0.85)	51.65 (-0.18)	54.39 (-3.23)	43.83 (-0.40)
S3	19.27 (-0.06)	39.02 (+0.85)	41.16 (+0.12)	31.71 (-0.73)	60.49 (+1.40)	68.78 (+0.30)	29.15 (-0.98)	53.41 (+1.59)	58.54 (+0.91)	44.61 (+0.38)
S6	19.57 (+0.24)	38.41 (+0.24)	40.61 (-0.43)	33.11 (+0.67)	59.63 (+0.55)	68.96 (+0.49)	29.27 (-0.85)	52.32 (+0.49)	56.89 (-0.73)	44.31 (+0.07)
S8	19.82 (+0.49)	37.93 (-0.24)	41.40 (+0.37)	32.38 (-0.06)	58.90 (-0.18)	68.78 (+0.30)	30.61 (+0.49)	51.77 (-0.06)	57.44 (-0.18)	44.34 (+0.10)
S9	17.38 (-1.95)	31.04 (-7.13)	45.55 (+4.51)	33.35 (+0.91)	58.54 (-0.55)	68.11 (-0.37)	28.78 (-1.34)	52.38 (+0.55)	58.35 (+0.73)	43.72 (-0.51)
S11	19.76 (+0.43)	37.01 (-1.16)	39.51 (-1.52)	33.41 (+0.98)	60.91 (+1.83)	68.05 (-0.43)	29.39 (-0.73)	52.32 (+0.49)	56.59 (-1.04)	44.11 (-0.13)
S12	19.45 (+0.12)	36.52 (-1.65)	45.73 (+4.70)	33.17 (+0.73)	60.12 (+1.04)	66.77 (-1.71)	31.28 (+1.16)	51.46 (-0.37)	58.35 (+0.73)	44.76 (+0.53)
S13	19.15 (-0.18)	37.44 (-0.73)	40.18 (-0.85)	32.32 (-0.12)	60.49 (+1.40)	68.29 (-0.18)	29.57 (-0.55)	53.11 (+1.28)	56.10 (-1.52)	44.07 (-0.16)
S14	17.26 (-2.07)	35.73 (-2.44)	39.70 (-1.34)	30.98 (-1.46)	60.43 (+1.34)	68.48 (-0.00)	26.59 (-3.54)	52.38 (+0.55)	56.34 (-1.28)	43.10 (-1.14)
S15	17.93 (-1.40)	26.83 (-11.34)	48.41 (+7.38)	32.56 (+0.12)	59.94 (+0.85)	67.74 (-0.73)	28.29 (-1.83)	52.07 (+0.24)	57.99 (+0.37)	43.53 (-0.70)
S16	16.89 (-2.44)	31.34 (-6.83)	45.91 (+4.88)	33.90 (+1.46)	60.85 (+1.77)	67.38 (-1.10)	30.85 (+0.73)	53.29 (+1.46)	59.39 (+1.77)	44.42 (+0.19)
S17	14.15 (-5.18)	22.56 (-15.61)	47.93 (+6.89)	33.48 (+1.04)	58.29 (-0.79)	66.77 (-1.71)	27.26 (-2.87)	51.10 (-0.73)	58.60 (+0.98)	42.24 (-2.00)
S18	14.70 (-4.63)	18.78 (-19.39)	50.85 (+9.82)	31.71 (-0.73)	57.13 (-1.95)	67.32 (-1.16)	25.12 (-5.00)	52.07 (+0.24)	58.23 (+0.61)	41.77 (-2.47)
Input PL = Python										
baseline	30.43	20.30	35.73	25.79	48.90	67.99	18.72	49.33	50.98	38.69
N4	30.67 (+0.24)	20.73 (+0.43)	32.93 (-2.80)	25.91 (+0.12)	49.27 (+0.37)	67.93 (-0.06)	16.59 (-2.13)	48.48 (-0.85)	53.23 (+2.26)	38.41 (-0.27)
N5	31.46 (+1.04)	19.82 (-0.49)	36.40 (+0.67)	25.79 (-0.00)	49.39 (+0.49)	68.96 (+0.98)	16.71 (-2.01)	47.99 (-1.34)	52.74 (+1.77)	38.81 (+0.12)
N6	30.37 (-0.06)	18.84 (-1.46)	40.43 (+4.70)	24.09 (-1.71)	47.99 (-0.91)	68.11 (+0.12)	16.83 (-1.89)	45.61 (-3.72)	52.68 (+1.71)	38.33 (-0.36)
S1	29.76 (-0.67)	19.88 (-0.43)	36.16 (+0.43)	25.49 (-0.30)	48.17 (-0.73)	67.62 (-0.37)	19.27 (+0.55)	48.96 (-0.37)	51.10 (+0.12)	38.49 (-0.20)
S2	30.00 (-0.43)	20.18 (-0.12)	36.46 (+0.73)	25.79 (+0.00)	48.66 (-0.24)	68.41 (+0.43)	18.78 (+0.06)	48.54 (-0.79)	50.55 (-0.43)	38.60 (-0.09)
S4	29.45 (-0.98)	21.40 (+1.10)	35.98 (+0.24)	25.55 (-0.24)	49.45 (+0.55)	67.87 (-0.12)	19.27 (+0.55)	48.96 (-0.37)	51.34 (+0.37)	38.81 (+0.12)
S5	29.63 (-0.79)	20.06 (-0.24)	35.06 (-0.67)	26.28 (+0.49)	49.51 (+0.61)	68.60 (+0.61)	18.60 (-0.12)	49.21 (-0.12)	52.01 (+1.04)	38.77 (+0.09)
S7	29.27 (-1.16)	21.16 (+0.85)	36.71 (+0.98)	25.67 (-0.12)	50.73 (+1.83)	68.78 (+0.79)	19.33 (+0.61)	48.54 (-0.79)	52.26 (+1.28)	39.16 (+0.47)
S10	31.16 (+0.73)	21.89 (+1.59)	39.51 (+3.78)	23.29 (-2.50)	50.91 (+2.01)	69.63 (+1.65)	25.67 (+6.95)	50.06 (+0.73)	53.17 (+2.20)	40.59 (+1.90)
S13	31.89 (+1.46)	19.27 (-1.04)	36.16 (+0.43)	26.10 (+0.30)	48.84 (-0.06)	66.95 (-1.04)	18.48 (-0.24)	49.21 (-0.12)	50.91 (-0.06)	38.64 (-0.04)
S14	30.06 (-0.37)	20.30 (+0.00)	36.95 (+1.22)	25.91 (+0.12)	48.05 (-0.85)	68.66 (+0.67)	19.57 (+0.85)	48.29 (-1.04)	51.34 (+0.37)	38.79 (+0.11)
S15	30.55 (+0.12)	19.76 (-0.55)	38.72 (+2.99)	24.76 (-1.04)	49.02 (+0.12)	68.90 (+0.91)	18.05 (-0.67)	48.60 (-0.73)	51.89 (+0.91)	38.92 (+0.23)
S16	31.34 (+0.91)	27.68 (+7.38)	41.10 (+5.37)	27.74 (+1.95)	46.95 (-1.95)	67.87 (-0.12)	22.07 (+3.35)	51.52 (+2.20)	52.80 (+1.83)	41.01 (+2.32)
S17	30.91 (+0.49)	26.22 (+5.91)	43.78 (+8.05)	26.46 (+0.67)	48.17 (-0.73)	67.93 (-0.06)	21.28 (+2.56)	50.91 (+1.59)	54.39 (+3.41)	41.12 (+2.43)
S18	29.45 (-0.98)	25.67 (+5.37)	45.85 (+10.12)	23.78 (-2.01)	50.00 (+1.10)	69.21 (+1.22)	25.30 (+6.59)	51.16 (+1.83)	54.88 (+3.90)	41.70 (+3.01)

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

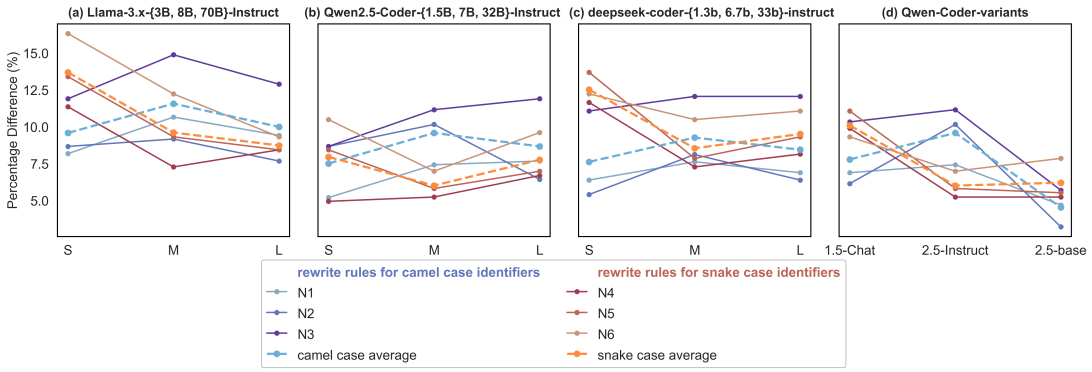


Figure 7: Percentage difference for naming rewrite transformations.

Table 21: Nucleus sampling with $K=10$ (temperature 0.2, top-p 0.95): pass@K and Δpass@K on HumanEvalPack bug-fixing tasks.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
Input PL = Java										
baseline	35.98	59.15	51.83	53.66	71.95	77.44	36.59	56.71	64.63	56.44
N1	37.80 (+1.83)	57.32 (-1.83)	53.05 (+1.22)	51.83 (-1.83)	71.95 (+0.00)	75.00 (-2.44)	35.98 (-0.61)	57.32 (+0.61)	61.59 (-3.05)	55.76 (-0.68)
N2	36.59 (+0.61)	55.49 (-3.66)	50.61 (-1.22)	51.83 (-1.83)	71.34 (-0.61)	73.78 (-3.66)	36.59 (+0.00)	58.54 (+1.83)	64.63 (+0.00)	55.49 (-0.95)
N3	37.80 (+1.83)	55.49 (-3.66)	56.10 (+4.27)	51.22 (-2.44)	70.12 (-1.83)	76.22 (-1.22)	37.20 (+0.61)	58.54 (+1.83)	63.41 (-1.22)	56.23 (-0.20)
S3	38.41 (+2.44)	59.15 (+0.00)	52.44 (+0.61)	52.44 (-1.22)	72.56 (+0.61)	75.00 (-2.44)	35.37 (-1.22)	57.32 (+0.61)	65.85 (+1.22)	56.50 (+0.07)
S6	35.37 (-0.61)	59.15 (+0.00)	51.83 (+0.00)	54.27 (+0.61)	71.34 (-0.61)	76.22 (-1.22)	34.76 (-1.83)	57.32 (+0.61)	63.41 (-1.22)	55.96 (-0.47)
S8	37.20 (+1.22)	60.37 (+1.22)	50.61 (-1.22)	51.83 (-1.83)	71.95 (+0.00)	75.61 (-1.83)	37.80 (+1.22)	57.32 (+0.61)	64.63 (+0.00)	56.37 (-0.07)
S9	31.71 (-4.27)	51.83 (-7.32)	57.32 (+5.49)	53.05 (-0.61)	68.90 (-3.05)	78.66 (+1.22)	33.54 (-3.05)	58.54 (+1.83)	64.63 (+0.00)	55.35 (-1.08)
S11	33.54 (-2.44)	54.88 (-4.27)	50.00 (-1.83)	53.05 (-0.61)	71.34 (-0.61)	76.83 (-0.61)	36.59 (+0.00)	57.93 (+1.22)	63.41 (-1.22)	55.28 (-1.15)
S12	32.93 (-3.05)	53.66 (-5.49)	57.93 (+6.10)	51.22 (-2.44)	70.12 (-1.83)	73.78 (-3.66)	38.41 (+1.83)	56.10 (-0.61)	65.85 (+1.22)	55.56 (-0.88)
S13	35.37 (-0.61)	57.32 (-1.83)	52.44 (+0.61)	53.66 (+0.00)	71.34 (-0.61)	75.61 (-1.83)	34.15 (-2.44)	57.93 (+1.22)	63.41 (-1.22)	55.69 (-0.75)
S14	35.98 (+0.00)	56.71 (-2.44)	51.83 (+0.00)	51.83 (-1.83)	71.34 (-0.61)	75.61 (-1.83)	32.93 (-3.66)	56.71 (+0.00)	64.02 (-0.61)	55.22 (-1.22)
S15	31.71 (-4.27)	45.12 (-14.02)	58.54 (+6.71)	49.39 (-4.27)	72.56 (+0.61)	76.83 (-0.61)	34.15 (-2.44)	55.49 (-1.22)	62.20 (-2.44)	54.00 (-2.44)
S16	28.05 (-7.93)	51.83 (-7.32)	57.93 (+6.10)	48.17 (-5.49)	70.73 (-1.22)	75.00 (-2.44)	35.98 (-0.61)	59.15 (+2.44)	65.24 (+0.61)	54.67 (-1.76)
S17	21.95 (-14.02)	36.59 (-22.56)	59.15 (+7.32)	48.17 (-5.49)	72.56 (+0.61)	76.22 (-1.22)	31.10 (-5.49)	54.27 (-2.44)	62.20 (-2.44)	51.36 (-5.08)
S18	26.22 (-9.76)	35.98 (-23.17)	61.59 (+9.76)	51.22 (-2.44)	70.73 (-1.22)	77.44 (+0.00)	28.66 (-7.93)	56.71 (+0.00)	62.80 (-1.83)	52.37 (-4.07)
Input PL = Python										
baseline	45.12	32.32	46.95	35.98	59.76	71.34	25.61	54.88	56.10	47.56
N4	46.34 (+1.22)	32.93 (+0.61)	42.68 (-4.27)	37.80 (+1.83)	60.98 (+1.22)	70.12 (-1.22)	22.56 (-3.05)	53.66 (-1.22)	59.76 (+3.66)	47.43 (-0.14)
N5	48.17 (+3.05)	32.93 (+0.61)	46.34 (-0.61)	35.37 (-0.61)	60.98 (+1.22)	72.56 (+1.22)	22.56 (-3.05)	51.22 (-3.66)	57.93 (+1.83)	47.56 (+0.00)
N6	45.12 (+0.00)	32.32 (+0.00)	50.61 (+3.66)	34.76 (-1.22)	59.76 (+0.00)	70.73 (-0.61)	21.95 (-3.66)	53.05 (-1.83)	58.54 (+2.44)	47.43 (-0.14)
S1	43.29 (-1.83)	31.71 (-0.61)	46.95 (+0.00)	36.59 (+0.61)	58.54 (-1.22)	70.73 (-0.61)	26.22 (+0.61)	53.66 (-1.22)	56.71 (+0.61)	47.15 (-0.41)
S2	45.12 (+0.00)	32.93 (+0.61)	46.95 (+0.00)	35.98 (+0.00)	59.76 (+0.00)	71.34 (+0.00)	25.61 (+0.00)	54.88 (+0.00)	56.10 (+0.00)	47.63 (+0.07)
S4	45.12 (+0.00)	33.54 (+1.22)	46.95 (+0.00)	37.20 (+1.22)	60.37 (+0.61)	70.12 (-1.22)	25.61 (+0.00)	54.88 (+0.00)	56.10 (+0.00)	47.76 (+0.20)
S5	44.51 (-0.61)	31.71 (-0.61)	46.95 (+0.00)	37.20 (+1.22)	59.15 (-0.61)	70.73 (-0.61)	26.22 (+0.61)	54.88 (+0.00)	56.71 (+0.61)	47.56 (+0.00)
S7	43.29 (-1.83)	32.93 (+0.61)	45.73 (-1.22)	37.20 (+1.22)	60.98 (+1.22)	72.56 (+1.22)	26.83 (+1.22)	53.05 (-1.83)	58.54 (+2.44)	47.90 (+0.34)
S10	43.90 (-1.22)	35.98 (+3.66)	48.17 (+1.22)	33.54 (-2.44)	62.20 (+2.44)	72.56 (+1.22)	35.37 (+9.76)	54.88 (+0.00)	61.59 (+5.49)	49.80 (+2.24)
S13	50.00 (+4.88)	32.93 (+0.61)	46.95 (+0.00)	36.59 (+0.61)	59.76 (+0.00)	71.34 (+0.00)	24.39 (-1.22)	55.49 (+0.61)	55.49 (-0.61)	48.10 (+0.54)
S14	46.34 (+1.22)	31.71 (-0.61)	46.95 (+0.00)	36.59 (+0.61)	58.54 (-1.22)	71.95 (+0.61)	25.61 (+0.00)	54.27 (-0.61)	56.71 (+0.61)	47.63 (+0.07)
S15	44.51 (-0.61)	34.76 (+2.44)	48.78 (+1.83)	36.59 (+0.61)	58.54 (-1.22)	73.17 (+1.83)	25.61 (+0.00)	53.66 (-1.22)	57.32 (+1.22)	48.10 (+0.54)
S16	45.12 (+0.00)	42.07 (+9.76)	48.78 (+1.83)	40.24 (+4.27)	56.10 (-3.66)	71.95 (+0.61)	27.44 (+1.83)	56.71 (+1.83)	59.15 (+3.05)	49.73 (+2.17)
S17	46.95 (+1.83)	42.68 (+10.37)	51.83 (+4.88)	37.20 (+1.22)	58.54 (-1.22)	73.78 (+2.44)	25.61 (+0.00)	55.49 (+0.61)	62.20 (+6.10)	50.47 (+2.91)
S18	45.12 (+0.00)	42.07 (+9.76)	53.05 (+6.10)	35.37 (-0.61)	57.32 (-2.44)	74.39 (+3.05)	33.54 (+7.93)	56.71 (+1.83)	62.20 (+6.10)	51.08 (+3.52)

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

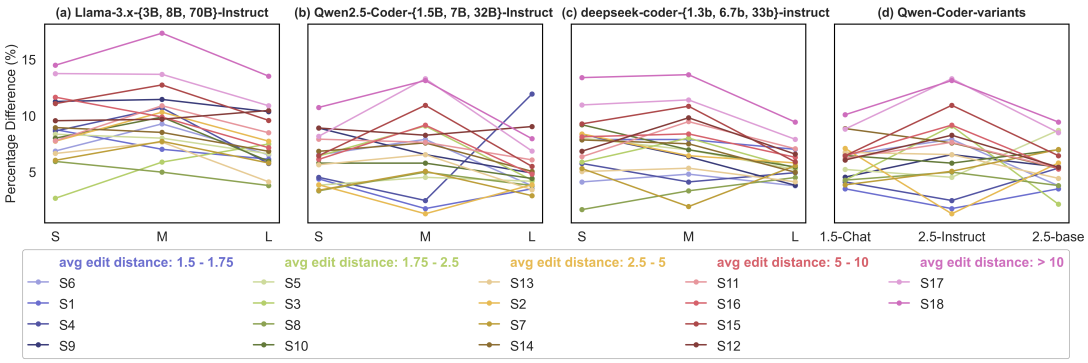


Figure 8: Percentage difference for spacing rewrite transformations.

Table 22: MRD-selected accuracy and Δaccuracy (in parenthesis), compared with greedy decoding on HumanEvalPack bug-fixing tasks.

Variant	Llama-3B	Llama-8B	Llama-70B	Qwen-1.5B	Qwen-7B	Qwen-32B	DS-1.3B	DS-6.7B	DS-33B	Average
Input PL = Java, Greedy Decoding										
baseline	19.51	39.02	42.68	32.32	59.76	67.68	30.49	51.83	57.32	44.51
N1	20.12 (+0.61)	39.63 (+0.61)	43.90 (+1.22)	35.37 (+3.05)	64.02 (+4.26)	68.29 (+0.61)	28.66 (-1.83)	51.22 (-0.61)	55.49 (-1.83)	45.19 (+0.68)
N2	20.73 (+1.22)	39.63 (+0.61)	43.90 (+1.22)	34.15 (+1.83)	61.59 (+1.83)	66.46 (-1.22)	31.10 (+0.61)	53.66 (+1.83)	54.88 (-2.44)	45.12 (+0.61)
N3	20.73 (+1.22)	38.41 (-0.61)	42.07 (-0.61)	33.54 (+1.22)	63.41 (+3.65)	68.29 (+0.61)	29.27 (-1.22)	53.05 (+1.22)	53.66 (-3.66)	44.71 (+0.20)
S3	19.51 (+0.00)	39.02 (+0.00)	41.46 (-1.22)	32.32 (+0.00)	60.98 (+1.22)	68.29 (+0.61)	29.27 (-1.22)	54.88 (+3.05)	57.32 (+0.00)	44.78 (+0.27)
S6	20.73 (+1.22)	39.02 (+0.00)	42.07 (-0.61)	32.93 (+0.61)	61.59 (+1.83)	68.29 (+0.61)	29.88 (-0.61)	51.83 (+0.00)	56.71 (-0.61)	44.78 (+0.27)
S8	19.51 (+0.00)	38.41 (-0.61)	43.90 (+1.22)	33.54 (+1.22)	59.76 (+0.00)	67.68 (+0.00)	30.49 (+0.00)	51.22 (-0.61)	58.54 (+1.22)	44.78 (+0.27)
S9	18.29 (-1.22)	31.71 (-7.31)	46.95 (+4.27)	30.49 (-1.83)	61.59 (+1.83)	66.46 (-1.22)	28.05 (-2.44)	51.83 (+0.00)	57.32 (+0.00)	43.63 (-0.88)
S11	20.73 (+1.22)	37.20 (-1.82)	40.24 (-2.44)	33.54 (+1.22)	60.37 (+0.61)	68.90 (+1.22)	29.88 (-0.61)	52.44 (+0.61)	56.10 (-1.22)	44.38 (-0.13)
S12	21.34 (+1.83)	36.59 (-2.43)	46.34 (+3.66)	34.76 (+2.44)	61.59 (+1.83)	65.85 (-1.83)	32.32 (+1.83)	51.83 (+0.00)	60.37 (+3.05)	45.67 (+1.16)
S13	19.51 (+0.00)	37.20 (-1.82)	42.07 (-0.61)	31.71 (-0.61)	62.80 (+3.04)	67.68 (+0.00)	29.88 (-0.61)	53.05 (+1.22)	54.88 (-2.44)	44.31 (-0.20)
S14	17.68 (-1.83)	35.37 (-3.65)	41.46 (-1.22)	29.88 (-2.44)	60.37 (+0.61)	69.51 (+1.83)	26.83 (-3.66)	53.05 (+1.22)	56.71 (-0.61)	43.43 (-1.08)
S15	17.68 (-1.83)	25.61 (-13.41)	46.95 (+4.27)	31.10 (-1.22)	60.37 (+0.61)	67.68 (+0.00)	26.83 (-3.66)	52.44 (+0.61)	59.76 (+2.44)	43.16 (-1.35)
S16	15.85 (-3.66)	31.10 (-7.92)	44.51 (+1.83)	33.54 (+1.22)	62.80 (+3.04)	68.90 (+1.22)	29.88 (-0.61)	53.05 (+1.22)	60.98 (+3.66)	44.51 (+0.00)
S17	12.20 (-7.31)	23.17 (-15.85)	46.95 (+4.27)	34.15 (+1.83)	60.37 (+0.61)	65.24 (-2.44)	26.83 (-3.66)	52.44 (+0.61)	58.54 (+1.22)	42.21 (-2.30)
S18	12.20 (-7.31)	16.46 (-22.56)	50.61 (+7.93)	34.76 (+2.44)	54.88 (-4.88)	67.68 (+0.00)	24.39 (-6.10)	51.22 (-0.61)	58.54 (+1.22)	41.19 (-3.32)
Input PL = Java, Minimum Risk Decoding										
baseline	17.68	39.63	43.29	34.15	57.93	68.29	28.66	53.66	57.93	44.58
N1	16.46 (-1.22)	39.63 (+0.00)	42.68 (-0.61)	35.37 (+1.22)	60.37 (+2.44)	66.46 (-1.83)	28.05 (-0.61)	53.66 (+0.00)	56.71 (-1.22)	44.38 (-0.20)
N2	16.46 (-1.22)	40.85 (+1.22)	41.46 (-1.83)	31.71 (-2.44)	60.98 (+3.05)	67.07 (-1.22)	30.49 (+1.83)	53.66 (+0.00)	56.10 (-1.83)	44.31 (-0.27)
N3	16.46 (-1.22)	39.02 (-0.61)	42.68 (-0.61)	34.15 (+0.00)	60.37 (+2.44)	70.12 (+1.83)	28.05 (-0.61)	52.44 (-1.22)	54.88 (-3.05)	44.24 (-0.34)
S3	17.68 (+0.00)	39.63 (+0.00)	42.68 (-0.61)	34.15 (+0.00)	59.76 (+1.83)	69.51 (+1.22)	27.44 (-1.22)	54.27 (+0.61)	59.15 (+1.22)	44.92 (+0.34)
S6	18.29 (+0.61)	39.63 (+0.00)	42.07 (-1.22)	34.15 (+0.00)	59.76 (+1.83)	69.51 (+1.22)	28.66 (+0.00)	53.66 (+0.00)	57.32 (-0.61)	44.78 (+0.20)
S8	18.29 (+0.61)	38.41 (-1.22)	43.29 (+0.00)	34.15 (+0.00)	57.93 (+0.00)	68.29 (+0.00)	28.66 (+0.00)	54.27 (+0.61)	57.93 (+0.00)	44.58 (+0.00)
S9	15.85 (-1.83)	29.27 (-10.36)	48.17 (+4.88)	35.37 (+1.22)	57.32 (-0.61)	68.90 (+0.61)	28.05 (-0.61)	52.44 (-1.22)	58.54 (+0.61)	43.77 (-0.81)
S11	18.29 (+0.61)	38.41 (-1.22)	39.63 (-3.66)	35.37 (+1.22)	63.41 (+5.48)	67.07 (-1.22)	29.27 (+0.61)	53.05 (-0.61)	56.10 (-1.83)	44.51 (-0.07)
S12	20.12 (+2.44)	40.85 (+1.22)	47.56 (+4.27)	32.93 (-1.22)	63.41 (+5.48)	68.29 (+0.00)	29.88 (+1.22)	51.83 (-1.83)	59.15 (+1.22)	46.00 (+1.42)
S13	17.68 (+0.00)	39.63 (+0.00)	42.68 (-0.61)	34.76 (+0.61)	61.59 (+3.66)	68.29 (+0.00)	28.66 (+0.00)	53.66 (+0.00)	57.32 (-0.61)	44.92 (+0.34)
S14	17.68 (+0.00)	34.76 (-4.87)	40.85 (-2.44)	34.15 (+0.00)	62.80 (+4.87)	70.12 (+1.83)	26.22 (-2.44)	53.66 (+0.00)	57.32 (-0.61)	44.17 (-0.41)
S15	17.07 (-0.61)	27.44 (-12.19)	48.17 (+4.88)	34.15 (+0.00)	59.76 (+1.83)	68.29 (+0.00)	26.83 (-1.83)	52.44 (-1.22)	59.76 (+1.83)	43.77 (-0.81)
S16	15.85 (-1.83)	32.32 (-7.31)	48.17 (+4.88)	34.15 (+0.00)	62.20 (+4.27)	68.29 (+0.00)	30.49 (+1.83)	53.66 (+0.00)	59.76 (+1.83)	44.99 (+0.41)
S17	14.02 (-3.66)	22.56 (-17.07)	48.17 (+4.88)	35.98 (+1.83)	62.20 (+4.27)	67.68 (-0.61)	28.05 (-0.61)	51.22 (-2.44)	59.15 (+1.22)	43.23 (-1.35)
S18	14.63 (-3.05)	21.34 (-18.29)	48.78 (+5.49)	31.71 (-2.44)	59.76 (+1.83)	67.07 (-1.22)	24.39 (-4.27)	51.22 (-2.44)	58.54 (+0.61)	41.94 (-2.64)
Input PL = Python, Greedy Decoding										
baseline	26.22	28.66	55.49	31.71	55.49	69.51	21.34	47.56	53.05	43.23
N4	31.10 (+4.88)	31.71 (+3.05)	55.49 (+0.00)	27.44 (-4.27)	54.88 (-0.61)	68.90 (-0.61)	19.51 (-1.83)	48.17 (+0.61)	51.83 (-1.22)	43.23 (+0.00)
N5	26.22 (+0.00)	34.15 (+5.49)	56.10 (+0.61)	29.88 (-1.83)	56.10 (+0.61)	68.90 (-0.61)	18.90 (-2.44)	47.56 (+0.00)	54.88 (+1.83)	43.63 (+0.40)
N6	28.66 (+2.44)	32.32 (+3.66)	55.49 (+0.00)	31.71 (+0.00)	56.10 (+0.61)	69.51 (+0.00)	18.90 (-2.44)	45.12 (-2.44)	53.66 (+0.61)	43.50 (+0.27)
S1	26.83 (+0.61)	28.05 (-0.61)	55.49 (+0.00)	31.71 (+0.00)	54.88 (-0.61)	69.51 (+0.00)	21.34 (+0.00)	47.56 (+0.00)	53.05 (+0.00)	43.16 (-0.07)
S2	26.83 (+0.61)	28.66 (+0.00)	55.49 (+0.00)	32.32 (+0.61)	55.49 (+0.00)	69.51 (+0.00)	20.73 (-0.61)	47.56 (+0.00)	53.05 (+0.00)	43.29 (+0.06)
S4	26.83 (+0.61)	28.05 (-0.61)	57.32 (+1.83)	31.10 (-0.61)	54.88 (-0.61)	68.29 (-1.22)	20.73 (-0.61)	47.56 (+0.00)	52.44 (-0.61)	43.02 (-0.21)
S5	26.83 (+0.61)	29.27 (+0.61)	54.88 (-0.61)	32.32 (+0.61)	54.27 (-1.22)	69.51 (+0.00)	20.73 (-0.61)	47.56 (+0.00)	52.44 (-0.61)	43.09 (-0.14)
S7	28.05 (+1.83)	30.49 (+1.83)	54.88 (-0.61)	31.71 (+0.00)	54.88 (-0.61)	69.51 (+0.00)	21.34 (+0.00)	48.17 (+0.61)	52.44 (-0.61)	43.50 (+0.27)
S10	29.88 (+3.66)	33.54 (+4.88)	56.71 (+1.22)	30.49 (-1.22)	55.49 (+0.00)	69.51 (+0.00)	26.22 (+4.88)	50.61 (+3.05)	54.88 (+1.83)	45.26 (+2.03)
S13	27.44 (+1.22)	32.32 (+3.66)	56.10 (+0.61)	31.71 (+0.00)	54.88 (-0.61)	70.12 (+0.61)	20.12 (-1.22)	47.56 (+0.00)	54.27 (+1.22)	43.84 (+0.61)
S14	26.22 (+0.00)	28.66 (+0.00)	54.27 (-1.22)	31.71 (+0.00)	54.88 (-0.61)	70.12 (+0.61)	22.56 (+1.22)	46.95 (-0.61)	54.88 (+1.83)	43.36 (+0.13)
S15	28.05 (+1.83)	27.44 (-1.22)	54.88 (-0.61)	30.49 (-1.22)	54.27 (-1.22)	69.51 (+0.00)	19.51 (-1.83)	48.78 (+1.22)	54.27 (+1.22)	43.02 (-0.21)
S16	33.54 (+7.32)	32.93 (+4.27)	57.93 (+2.44)	31.10 (-0.61)	53.66 (-1.83)	67.68 (-1.83)	21.95 (+0.61)	50.61 (+3.05)	54.27 (+1.22)	44.85 (+1.62)
S17	33.54 (+7.32)	32.93 (+4.27)	57.32 (+1.83)	29.88 (-1.83)	53.66 (-1.83)	67.68 (-1.83)	20.12 (-1.22)	51.22 (+3.66)	55.49 (+2.44)	44.65 (+1.42)
S18	30.49 (+4.27)	32.93 (+4.27)	60.37 (+4.88)	30.49 (-1.22)	55.49 (+0.00)	67.68 (-1.83)	23.17 (+1.83)	52.44 (+4.88)	58.54 (+5.49)	45.73 (+2.50)
Input PL = Python, Minimum Risk Decoding										
baseline	31.10	20.12	39.02	25.61	46.95	68.29	17.68	48.78	50.61	38.68
N4	30.49 (-0.61)	20.12 (+0.00)	34.76 (-4.26)	25.00 (-0.61)	45.73 (-1.22)	68.29 (+0.00)	16.46 (-1.22)	48.17 (-0.61)	53.66 (+3.05)	38.08 (-0.60)
N5	32.32 (+1.22)	20.12 (+0.00)	37.20 (-1.82)	24.39 (-1.22)	45.12 (-1.83)	68.90 (+0.61)	15.85 (-1.83)	48.78 (+0.00)	53.66 (+3.05)	38.48 (-0.20)
N6	30.49 (-0.61)	17.07 (-3.05)	41.46 (+2.44)	22.56 (-3.05)	42.68 (-4.27)	68.29 (+0.00)	17.68 (+0.00)	45.73 (-3.05)	53.05 (+2.44)	37.67 (-1.01)
S1	29.88 (-1.22)	20.12 (+0.00)	40.24 (+1.22)	24.39 (-1.22)	46.34 (-0.61)	67.68 (-0.61)	18.29 (+0.61)	48.78 (+0.00)	51.22 (+0.61)	38.55 (-0.13)
S2	29.88 (-1.22)	20.12 (+0.00)	39.02 (+0.00)	25.61 (+0.00)	46.34 (-0.61)	68.90 (+0.61)	18.29 (+0.61)	47.56 (-1.22)	50.00 (-0.61)	38.41 (-0.27)
S4	28.66 (-2.44)	20.73 (+0.61)	37.80 (-1.22)	25.61 (+0.00)	48.17 (+1.22)	67.68 (-0.61)	18.90 (+1.22)	48.17 (-0.61)	50.61 (+0.00)	38.48 (-0.20)
S5	29.27 (-1.83)	20.73 (+0.61)	37.80 (-1.22)	26.22 (+0.61)	46.95 (+0.00)	68.29 (+0.00)	17.68 (+0.00)	48.78 (+0.00)	51.83 (+1.22)	38.62 (-0.06)
S7	28.66 (-2.44)	21.95 (+1.83)	39.02 (+0.00)	25.00 (-0.61)	47.56 (+0.61)	68.90 (+0.61)	18.90 (+1.22)	48.17 (-0.61)	51.83 (+1.22)	38.89 (+0.21)
S10	31.71 (+0.61)	21.34 (+1.22)	40.85 (+1.83)	21.34 (-4.27)	49.39 (+2.44)	70.73 (+2.44)	25.00 (+7.32)	49.39 (+0.61)	53.05 (+2.44)	40.31 (+1.63)
S13	32.32 (+1.22)	19.51 (-0.61)	39.63 (+0.61)	25.61 (+0.00)	45.73 (-1.22)	66.46 (-1.83)	17.68 (+0.00)	48.78 (+0.00)	50.00 (-0.61)	38.41 (-0.27)
S14	30.49 (-0.61)	19.51 (-0.61)	39.63 (+0.61)	25.61 (+0.00)	45.73 (-1.22)	68.90 (+0.61)	19.51 (+1.83)	47.56 (-1.22)	51.22 (+0.61)	38.68 (+0.00)
S15	31.10 (+0.00)	18.29 (-1.83)	40.24 (+1.22)	23.78 (-1.83)	45.12 (-1.83)	67.68 (-0.61)	17.68 (+0.00)	48.78 (+0.00)	50.61 (+0.00)	38.14 (-0.54)
S16	29.88 (-1.22)	28.66 (+8.54)	41.46 (+2.44)	27.44 (+1.83)	43.29 (-3.66)	68.29 (+0.00)	23.17 (+5.49)	50.00 (+1.22)	53.66 (+3.05)	40.65 (+1.97)
S17	29.88 (-1.22)	27.44 (+7.32)	43.29 (+4.27)	26.22 (+0.61)	42.68 (-4.27)	66.46 (-1.83)	21.95 (+4.27)	50.61 (+1.83)	54.27 (+3.66)	40.31 (+1.63)
S18	29.27 (-1.83)	24.39 (+4.27)	47.56 (+8.54)	21.95 (-3.66)	48.78 (+1.83)	68.29 (+0.00)	25.00 (+7.32)	50.00 (+1.22)	57.32 (+6.71)	41.40 (+2.72)

Background color: baseline in grey, variants better than baseline in green, and variants worse than baseline in red.
The best variant is highlighted in **bold** and the worst variant is underlined.

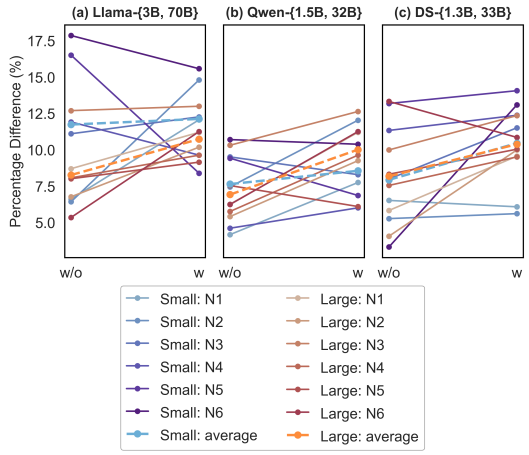


Figure 9: Naming rewrite rules percentage difference (with or without fragment change).

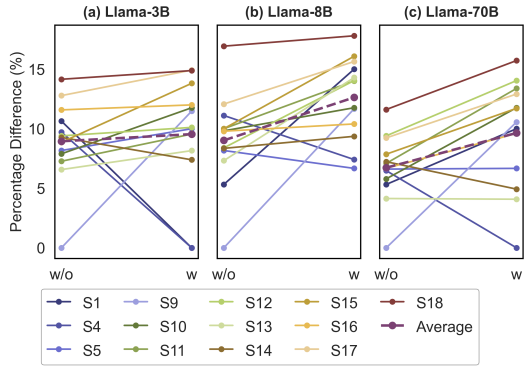


Figure 10: (Llama series) Spacing rewrite rules percentage difference (with or without fragment change).

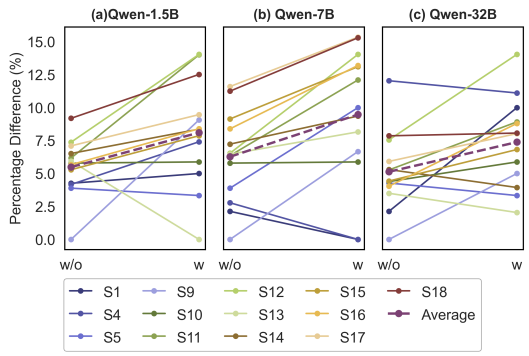


Figure 11: (Qwen series) Spacing rewrite rules percentage difference (with or without fragment change).

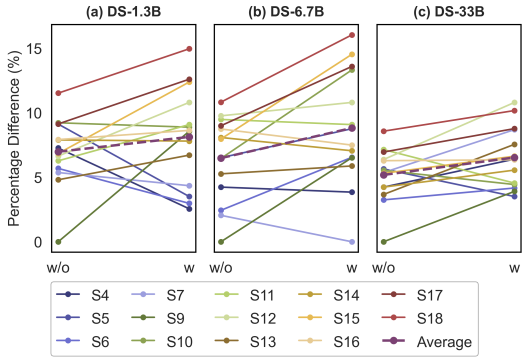


Figure 12: (Deepseek series) Spacing rewrite rules percentage difference (with or without fragment change).

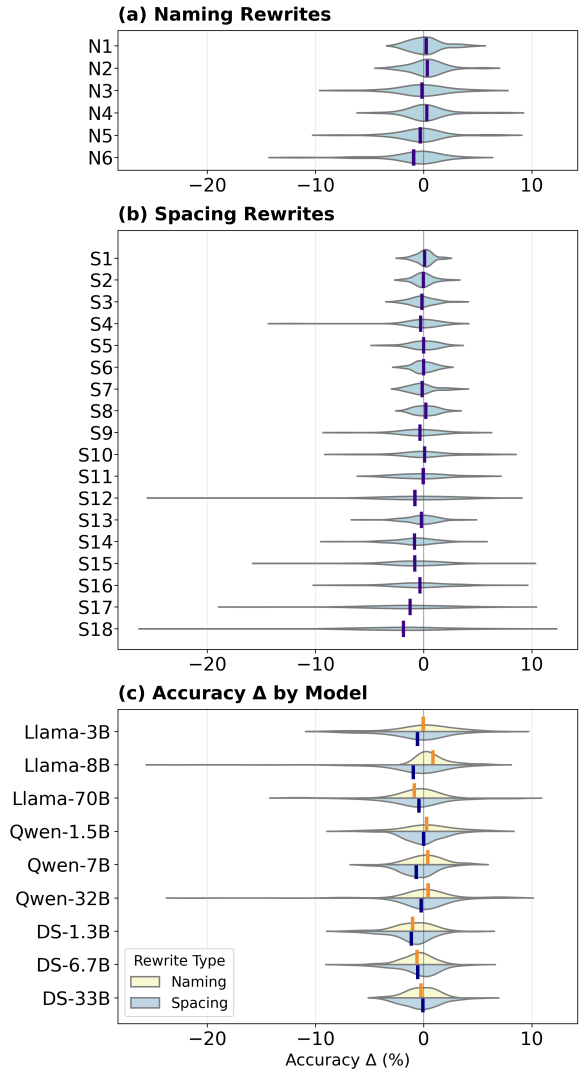


Figure 13: Distribution of Δaccuracy per rewrite rule across models and benchmarks.

Table 23: Samples affected by each rewrite rule, broken down by model series.

Rewrite Rule	Model Series	Total	Unchanged	Changed			
				All	Merged	Split	Mixed
Naming	Llama-3	2238	1292	946	105	767	74
	Qwen2.5-Coder	2238	1292	946	105	767	74
	DeepSeek-Coder	2247	999	1248	123	996	129
	CodeQwen1.5	2247	954	1293	136	1043	114
Spacing	Llama-3	12804	9315	3489	660	2394	435
	Qwen2.5-Coder	12804	9315	3489	660	2391	438
	DeepSeek-Coder	12804	8381	4423	608	3091	724
	CodeQwen1.5	12804	8720	4084	725	2504	855

Table 24: Details on the impact of different types of fragment changes on sensitivity.

Rewrite Rule	Model	Total	Unchanged	Changed (all)	Changed (subcategories)		
					Merged	Split	Mixed
Naming	Llama-S	11.48	10.68	12.58	10.48	13.17	9.46
	Llama-M	10.68	9.44	12.37	8.57	13.30	8.11
	Llama-L	9.43	8.13	11.21	9.52	11.73	8.11
	Qwen-S	7.73	6.97	8.77	8.57	9.00	6.76
	Qwen-M	7.95	7.35	8.77	5.71	9.00	10.81
	Qwen-L	8.27	6.58	10.57	11.43	10.82	6.76
	DS-S	9.88	8.31	11.14	4.88	11.95	10.85
	DS-M	8.95	7.91	9.78	7.32	10.54	6.20
	DS-L	8.95	6.61	10.82	10.57	10.64	12.40
Spacing	Llama-S	10.22	9.32	12.61	11.06	13.37	10.80
	Llama-M	10.99	9.69	14.45	13.03	14.83	14.48
	Llama-L	8.51	7.24	11.89	10.00	11.53	16.78
	Qwen-S	7.07	6.04	9.80	8.33	9.62	13.01
	Qwen-M	8.87	7.53	12.47	12.42	10.71	22.15
	Qwen-L	5.71	5.09	7.37	7.42	6.48	12.10
	DS-S	8.36	7.25	10.47	8.72	10.45	12.02
	DS-M	8.71	7.58	10.85	10.36	10.19	14.09
	DS-L	6.26	5.80	7.12	6.41	6.44	10.64