Detecting *Functional* Memorization in Code Language Models

Anonymous Authors[1]

Abstract

Large language models (LLMs) increasingly generate code at scale, trained on large corpora that may include restricted or proprietary sources. Meanwhile, models have been shown to memorize and reproduce training data, with audits focusing on textual overlap between training examples and model generations. Code, however, can be functionally equivalent while textually dissimilar. In this work, we study *functional memorization*: extraction of functional logic beyond what verbatim metrics detect. We construct a *counterfactual* setup for Olmo-3-32B, comparing a midtrained model (exposed to target code) against a pretrained reference (not exposed). We prompt both models with Python function signatures and measure both textual and functional similarity (i.e., LLM-as-a-judge, execution-based). Our results show clear evidence of functional memorization, highlighting the need for auditing metrics that go beyond textual overlap.

1. Introduction

Large language models (LLMs) are increasingly used to generate code, both as standalone models (Roziere et al., 2023; Feng et al., 2020; Olmo et al., 2025; Li et al., 2022) and as core components of coding agents (Zeff, 2026; Thompson, 2026). These capabilities are acquired by training on large corpora of source code, often scraped from public repositories with permissive licenses (Allal et al., 2025; Li et al., 2022; Roziere et al., 2023). However, filtering out non-permissively licensed code is difficult (Katzy et al., 2024), while training data is also increasingly sourced from user coding sessions (GitHub, 2026; Loizos, 2025).

At the same time, LLMs have been shown to memorize and reproduce their training data. This phenomenon has been studied extensively for natural language, from verbatim extraction of arbitrary sequences (Carlini et al., 2021; Nasr et al., 2025) to reproduction of copyrighted text (Cooper et al., 2025). Prior work also applied similar techniques to code, measuring near-verbatim leakage of source code (Al-Kaswan et al., 2024; Salerno et al., 2025) and secrets such as API keys (Nie et al., 2025; Huang et al., 2024).

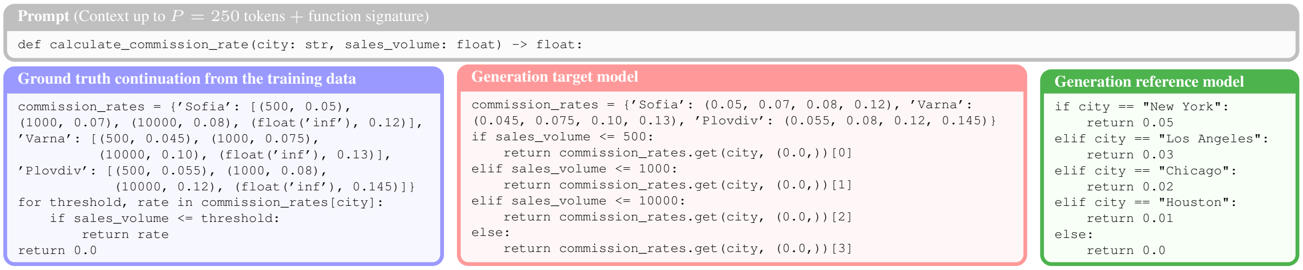
However, existing work on code leakage focuses almost exclusively on *textual* similarity: does the model's output match the training data token-for-token, or nearly so? This framing misses a critical dimension when models are trained on code containing meaningful proprietary logic, e.g., custom recommendation systems, trading algorithms, or content moderation rules, that owners would reasonably want to protect. Indeed, code can be *functionally equivalent* while being textually dissimilar: variables can be renamed, comments added or removed, control flow restructured, or entirely different algorithms used to achieve the same functionality—as has been widely studied in code synthesis and clone detection (Roy & Cordy, 2007; Ren et al., 2020; Song et al., 2024). If a model internalizes functional logic from its training data and reproduces it in a different surface form, text-based metrics will fail to detect it—yet the underlying intellectual property may still be leaked.

Contributions. In this work, we study the problem of *functional memorization* for code: the extraction of functional logic from LLMs trained on code, beyond what (near-)verbatim metrics can detect. We quantify this leakage for the open-source model Olmo-3-32B (Olmo et al., 2025). To properly distinguish memorization from generalization, we draw from prior work (Zhang et al., 2023; Hayes et al., 2025b) and leverage Olmo-3's training transparency to construct a *counterfactual* setup. Specifically, we select Python functions with meaningful functional logic from its midtraining data mixture and compare the generations from the midtrained model (exposed to target code) against the pretrained reference model (not exposed). We prompt both models with function signatures and measure per-sample similarity to the ground truth using a comprehensive suite of metrics spanning text overlap (BLEU, edit distance (Ippolito et al., 2023)), structural code similarity (CodeBLEU (Ren et al., 2020), tree-edit distance (Song et al., 2024)), LLM-as-a-judge (Song et al., 2024; Nikiema et al., 2025), and execution-based clone detection (Liang et al., 2025).

---

[1] Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by *The Impact of Memorization on Trustworthy Foundation Models* Workshop @ ICML. Do not distribute.



Prompt (Context up to $P = 250$ tokens + function signature)

```
def calculate_commission_rate(city: str, sales_volume: float) -> float:
```

Ground truth continuation from the training data

```
commission_rates = {’Sofia’: [(500, 0.05),
(1000, 0.07), (10000, 0.08), (float(’inf’), 0.12)],
’Varna’: [(500, 0.045), (1000, 0.075),
          (10000, 0.10), (float(’inf’), 0.13)],
’Plovdiv’: [(500, 0.055), (1000, 0.08),
            (10000, 0.12), (float(’inf’), 0.145)]}
for threshold, rate in commission_rates[city]:
    if sales_volume <= threshold:
        return rate
return 0.0
```

Generation target model

```
commission_rates = {’Sofia’: (0.05, 0.07, 0.08, 0.12), ’Varna’:
(0.045, 0.075, 0.10, 0.13), ’Plovdiv’: (0.055, 0.08, 0.12, 0.145)}
if sales_volume <= 500:
    return commission_rates.get(city, (0.0,))[0]
elif sales_volume <= 1000:
    return commission_rates.get(city, (0.0,))[1]
elif sales_volume <= 10000:
    return commission_rates.get(city, (0.0,))[2]
else:
    return commission_rates.get(city, (0.0,))[3]
```

Generation reference model

```
if city == "New York":
    return 0.05
elif city == "Los Angeles":
    return 0.03
elif city == "Chicago":
    return 0.02
elif city == "Houston":
    return 0.01
else:
    return 0.0
```

Figure 1. **Counterfactual functional memorization**. The function from Olmo-3-32B's midtraining data contains hard-coded logic; the target model (midtrained) reproduces the same logic in restructured code, while the reference model (pretrained) hallucinates. Textual metrics fail to flag memorization (BLEU $= 0.26$), while functional metrics succeed: LLM-judge $= 0.8$, execution based $= 0.95$.

We find that midtraining increases similarity across all metrics (Table 1), indicating a systematic shift of generations toward the training data. Following prior work studying near-verbatim memorization (Ippolito et al., 2023; Hayes et al., 2025b), we find that 8 (0.11%) functions are counterfactually memorized exactly (the target produces a verbatim match, the reference does not), and 43 (0.58%) approximately (BLEU≥ 0.75 for the target, BLEU< 0.75 for the reference). Importantly, we observe functional memorization at a similar order of magnitude, with 3.9% of samples counterfactually functionally memorized under our most conservative LLM-judge (with BLEU< 0.75). Under the most stringent evaluation, execution-based testing finds that 0.28% of the generations are functionally identical to a training-data function. Figure 1 (details in App. E.2) provides a compelling example: the target model reproduces hard-coded logic from training data in a substantially restructured implementation, while the reference model produces unrelated logic. Together, these results provide evidence of *functional* memorization in code models and motivate auditing regimes that go beyond textual overlap.

2. Background

Training data extraction. We consider an LLM $\mathcal{M}$ trained on dataset $\mathcal{D}$ that contains source code. Following prior work on memorization (Carlini et al., 2023; Nasr et al., 2025; Ippolito et al., 2023), we study training data extraction attacks that prompt the model with a prefix p (of token length P) taken from a training sequence, and then decode a continuation $x^\star \leftarrow \mathcal{M}(p)$. We focus on *greedy decoding*, i.e., consecutively sampling the token with the greatest predicted probability, and quantify leakage by comparing the ground-truth continuation x to $x^\star$ using similarity metric SIM. Prior work has considered different textual similarity metrics, ranging from exact (Carlini et al., 2021; 2023) to near-verbatim matches (Ippolito et al., 2023).

Counterfactual memorization. A key challenge when studying memorization through extraction is distinguishing it from generalization (Liu et al., 2025): when a model generates a sequence similar to its training data, is it reproducing memorized content or independently arriving at a natural solution? To address this, Zhang et al. (2023) propose using a definition of *counterfactual* memorization, characterizing how a model's predictions change when a particular sample is omitted during training. Building on this idea, Hayes et al. (2025b) operationalize counterfactual memorization in the context of extraction. Let a target model $\mathcal{M}_T$ be trained on dataset D containing the full training sample $z = p||x$, and let a *reference* model $\mathcal{M}_R$ be trained on $D \setminus \{z\}$. Then z is *counterfactually memorized* if $\mathcal{M}_T$ produces x when prompted with p using greedy decoding, while $\mathcal{M}_R$ does not. Further, z is said to be k-approximately counterfactually memorized if the edit distance between x and the $\mathcal{M}_T$ completion is $\leq k$, while the edit distance between x and the $\mathcal{M}_R$ completion is $> k$. In practice, counterfactual reference models are expensive to obtain, and have been approximated by earlier model checkpoints before training on target data (Hayes et al., 2025b).

Code similarity. In parallel to memorization studies, prior work has studied the similarity between pieces of code in the context of clone detection and evaluation of synthesized code. Roy & Cordy (2007) categorize clones into four types: Type I (identical up to whitespace/comments), Type II (renaming of identifiers), Type III (modified statements), and Type IV (functional equivalence with different syntax). Motivated by this taxonomy, many metrics have been proposed to go beyond string matching. Ren et al. (2020) propose CodeBLEU, combining n-gram overlap with syntactic similarity (AST matching) and semantic signals (data-flow). Follow-up work propose refined structural similarity measures (e.g., AST edit distance) (Song et al., 2024; Yu et al., 2025) and metrics based on embedding similarity using representation models such as CodeBERT (Feng et al., 2020). More recently, LLM-judges were explored for clone detection, showing improved sensitivity for Type III-IV similarity (Almatrafi et al., 2025; Maveli et al., 2025; Nikiema et al., 2025). Finally, to assess functional equivalence, Liang et al. (2025) incorporates function execution with LLM-generated inputs. We further elaborate on related work in App. A.

3. Counterfactual functional memorization

Our goal is to detect and quantify *functional memorization*: cases where a code model reproduces the functional logic of a training function without reproducing its text (near) verbatim. We distinguish textual overlap, measured by SIM_{text} (e.g., BLEU), from functional overlap, measured by SIM_{func} between the ground-truth x and the generated continuation $x^\star$. We say a sample $z = p||x$ is *counterfactually functionally memorized* by target model $\mathcal{M}_T$ if its generation $x^\star_T$ satisfies: (i) low textual overlap with x, i.e., $\text{SIM}_{\text{text}}(x, x^\star_T) < \tau_{\text{text}}$; (ii) functional equivalence, i.e., $\text{SIM}_{\text{func}}(x, x^\star_T) \geq \tau_{\text{func}}$; and (iii) a counterfactual divergence, i.e., a reference model's generation for the same prompt is *not* functionally equivalent, or $\text{SIM}_{\text{func}}(x, x^\star_R) < \tau_{\text{func}}$. This isolates a phenomenon distinct from verbatim leakage: the target model has internalized the *logic* of the training data and can reproduce it in a different surface form. Condition (iii), following Zhang et al. (2023); Hayes et al. (2025b), filters out high similarity attributable to the target model's generalization capabilities.

Similarity metrics. As text-based similarity we consider the *Exact match* rate, *BLEU* (1–4 gram overlap), and *Edit similarity score* (normalizing by $|x|$) as proposed by Ippolito et al. (2023) and the normalized length of the longest common substring (*LCS score*). We then consider 3 classes of functional similarity metrics. (i) Structural code metrics: CodeBLEU (Ren et al., 2020) with equal weights (*CodeBLEU (equal)*), its syntax-only AST subtree match (*CodeBLEU (syntax)*) and data-flow-only variant (*CodeBLEU (DFG)*) and *TSED*, the normalized AST tree-edit-distance score from Song et al. (2024). (ii) LLM-as-a-judge. We prompt an LLM to assess functional equivalence between the reference function x and the generation $x^\star$. We use three distinct prompts: the structural similarity prompt from Song et al. (2024) (*(Song)*), the similarity score with categorical labeling following Nikiema et al. (2025) (*(Nikiema)*), and our own functional-equivalence judge (*(Ours)*) (all prompts in App. C.1). (iii) Execution-based clone detection. We adapt HyClone (Liang et al., 2025) as a two-stage, execution-based similarity detector. In Stage 1, HyClone produces a binary LLM-based screening decision (*HyClone Stage 1*). For the pairs that pass this screen, Stage 2 first generates valid inputs for both functions, executes them and computes a similarity score based on the output match (*HyClone similarity*, details in App. C.2). All LLM-based metrics are instantiated with LLaMA-3.1-70B-Instruct (Grattafiori et al., 2024). For each metric, we compute $\text{SIM}(x, x^\star_T)$ for the target and $\text{SIM}(x, x^\star_R)$ for the reference model, and also compute the *delta* $\Delta = \text{SIM}(x, x^\star_T) - \text{SIM}(x, x^\star_R)$ to examine memorization.

Data collection. We study counterfactual functional memorization for Olmo-3-32B (Olmo et al., 2025), an open-source model released with full transparency regarding its training data and intermediate checkpoints across all stages. We focus on Python code included in the model's Dolmino *mid-training* corpus (100B tokens). Specifically, we consider CraneCode, a filtered and rewritten version of the Python subset of the-stack-v2-smol (Lozhkov et al., 2024). We sample 100k functions from CraneCode and parse all Python functions with body lengths between 10 and 50 lines. To focus on functions that encode meaningful logic, we use LLaMA-3.1-7B-Instruct (Grattafiori et al., 2024) as a judge to (i) filter for meaningful functional logic and (ii) assign a leakage-severity score (1–10). This yields 7,422 functions for subsequent analysis. We provide more details in App. B.

Table 1. Mean (± std) similarity to ground truth for target and reference models. All deltas are positive suggesting memorization.

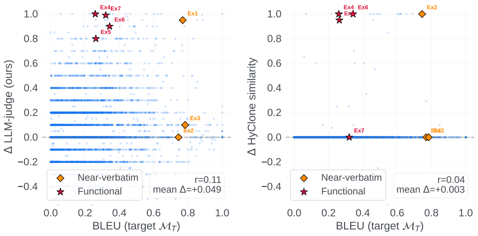
	Metric	**Target**	**Ref**	Δ
Text	BLEU	.185±.180	.141±.165	+.044
	Edit sim. score	.435±.178	.401±.175	+.034
	LCS score	.112±.109	.094±.102	+.019
Structural	CodeBLEU (equal)	.300±.180	.264±.169	+.036
	CodeBLEU (syntax)	.327±.199	.298±.188	+.029
	CodeBLEU (DFG)	.339±.218	.312±.210	+.027
	TSED	.397±.190	.368±.179	+.030
LLM-judge	Song et al. (2024)	.391±.220	.352±.209	+.039
	Nikiema et al. (2025)	.563±.255	.510±.265	+.054
	Ours	.209±.269	.160±.242	+.049
Execution	HyClone Stage 1	.037±.189	.022±.148	+.015
	HyClone similarity	.005±.066	.002±.044	+.003

Counterfactual setup. As target model $\mathcal{M}_T$, we consider the midtrained checkpoint ('stage2-ingredient1-step23842'), which has been trained on CraneCode. As reference model $\mathcal{M}_R$, we consider the initially pretrained checkpoint ('stage1-step656000'), which has *not* been trained on CraneCode. This provides a near-ideal counterfactual: both models share the same architecture and pretraining, differing only in their exposure to the midtraining data.

Function completion generation. We query both models to complete each function based on a prefix consisting of the function signature (excluding the docstring) and up to $P = 250$ preceding tokens of context. We generate continuations using greedy decoding with a maximum length of 500 tokens and then parse a valid function from the generated output, denoted as $x^\star_T$ (target) and $x^\star_R$ (reference).

4. Results

We study how the similarity between midtraining data functions and the generated continuations evolves from the reference to the target model. Table 1 reports the mean (± std) similarity for all metrics across all functions. All deltas (Δ) are positive, confirming that exposure during midtraining systematically increases similarity across every metric—from text overlap to execution-based functional equivalence. We further examine the correlations between metrics in



(a) Δ LLM-judge (Ours) *(b)* Δ HyClone

Figure 2. Functional similarity delta Δ vs. BLEU for the target model for (a) LLM-as-a-judge (Ours) and (b) HyClone. The upper-left quadrant (low BLEU, positive Δ) contains training data functions that are *functionally* memorized, i.e., generated continuations are functionally similar to the training data despite minimal token overlap. We highlight the examples for near-verbatim and functional memorization from App. E. All metrics in App. D.2.

Table 2. Counterfactual functional memorization across metrics. Counts where $\text{BLEU}(x, x_T^\star) < \tau_\text{text}$, $\text{SIM}_\text{func}(x, x_T^\star) \geq \tau_\text{func}$ but $\text{SIM}_\text{func}(x, x_R^\star) < \tau_\text{func}$;$\tau_\text{text} = \tau_\text{func} = 0.75$.

Functional similarity metric (τ)		**Count (%)**
Structural	CodeBLEU (equal)	28 (0.4%)
	CodeBLEU (syntax)	88 (1.2%)
	CodeBLEU (DFG)	130 (1.8%)
	TSED	154 (2.1%)
LLM-judge	Song et al. (2024)	311 (4.2%)
	Nikiema et al. (2025)	1040 (14.0%)
	Ours (func. equiv.)	290 (3.9%)
Execution	HyClone Stage 1 (binary)	156 (2.1%)
	HyClone similarity	21 (0.3%)
Unique (any metric)		1,486 (20.0%)
Unique (excl. Nikiema)		659 (8.9%)

App. D.3 and their per-function values in App. D.1.

We first examine **(near-)verbatim memorization**. We find that the target model produces 17 (0.23%) exact matches and 116 (1.56%) near-verbatim completions (BLEU ≥ 0.75, following Ippolito et al. (2023)). Yet the reference model, which has not seen the CraneCode data, already produces 12 exact and 85 near-verbatim completions. This highlights the importance of using the reference model to isolate memorization: in some cases, high similarity may stem from the model generalizing rather than memorizing the specific midtraining sample. We find that 8 (0.11%) functions are counterfactually memorized exactly (the target produces a verbatim match, the reference does not), and 43 (0.58%) functions yield a BLEU ≥ 0.75 for the target while BLEU < 0.75 for the reference model. We provide examples in App. E.1.

We further examine the occurrence of **counterfactual functional memorization** as defined in Section 3. Figure 2 plots the *delta* Δ (target minus reference) for 2 functional similarity metrics against the target model's BLEU score. To detect functional memorization, we focus on the upper-left quadrant: functions not produced near-verbatim (BLEU $\leq \tau_\text{text}$) but whose *logic* was memorized (positive delta). Table 2 quantifies the number of functions that would be considered not memorized following textual similarity ($\text{BLEU}(x, x_T^\star) < \tau_\text{text} = 0.75$, following Ippolito et al. (2023)), but are functionally memorized, i.e., $\text{SIM}_\text{func}(x, x_T^\star) \geq \tau_\text{func}$ but $\text{SIM}_\text{func}(x, x_R^\star) < \tau_\text{func}$. Across metrics, we find a substantial number of target model generations that are functionally similar to the midtraining data while having limited textual overlap. We discuss each of the metrics, associated counts and limitations below.

CodeBLEU and TSED measure similarity through AST matching and/or data-flow graph analysis, capturing shared structure even when tokens differ. CodeBLEU's data-flow component flags 130 functions (1.8%) as counterfactually memorized at $\tau_\text{func} = 0.75$, and TSED flags 154 (2.1%). These counts are substantial compared to the 0.58% flagged based on BLEU and suggest that generated code may be structurally similar while only having limited textual overlap. Examples in App. E.2 (e.g., Ex. 7) confirm that these metrics can reveal functionally memorized code. A closer inspection, however, also highlights their limitations for real-world functions: they can produce noisy scores and false positives when functions are short, dominated by print or plot statements, or consist primarily of attribute assignments (e.g., `self.x =`). They also require successful AST parsing, which fails for 1,114 of our 7,422 generated samples.

LLM-based judges use their reasoning capabilities to judge similarity holistically, rather than relying on syntactic overlap. We find that they flag, across prompts, between 290 (3.9%, our prompt) and 1,040 (14.0%, prompt from Nikiema et al. (2025)), counterfactual cases. The variation across prompts suggests that the Nikiema rating is more lenient, while our equivalence prompt is the most conservative. While results differ across prompts, we empirically find that pairs for which the score is ≥ 0.75 for our more conservative equivalence prompt have substantial functional similarities (examples in App. E.2).

Lastly, we implement HyClone (Liang et al., 2025) as the most stringent execution-based test. Stage 1 classifies 276 pairs as likely clones for the target model versus 166 for the reference. In Stage 2, 21 pairs (0.28%) are *counterfactually* execution-verified: the target model produces a functional clone (similarity ≥ 0.75) while the reference model does not (similarity < 0.75). Note that this count is a lower bound: the majority of Stage-1 clones fail to execute because functions depend on class state (`self`), unavailable packages, file I/O, or global variables (details in App. C.2). We provide examples of HyClone-confirmed functionally memorized cases in Figure 1 and in App. E.2.

Together, our results provide evidence of functional memorization in code LLMs, prompting the need for auditing beyond textual overlap. We conclude with a summary of the advantages and limitations for each functional metric and promising avenues for future work in App. F.

Impact Statement

This paper presents work whose goal is to advance the understanding of training-data leakage in LLMs trained on code. We believe this work has positive societal implications by helping developers and organizations better audit code models for potential intellectual property leakage. All code analyzed in this study comes from publicly available open-source repositories.

References

Ahmed, A., Cooper, A. F., Koyejo, S., and Liang, P. Extracting books from production language models. *arXiv preprint arXiv:2601.02671*, 2026.

Al-Kaswan, A., Izadi, M., and Van Deursen, A. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Penedo, G., Tunstall, L., Marafioti, A., Kydlíček, H., Lajarín, A. P., Srivastav, V., et al. Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

Allen-Zhu, Z. and Li, Y. Physics of language models: part 3.1, knowledge storage and extraction. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 1067–1077, 2024.

Almatrafi, A. A., Eassa, F. A., and Sharaf, S. A. Code clone detection techniques based on large language models. *IEEE Access*, 2025.

Carlini, N., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, U., et al. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*, pp. 2633–2650, 2021.

Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramer, F., and Zhang, C. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Cooper, A. F., Gokaslan, A., Ahmed, A., Cyphert, A. B., De Sa, C., Lemley, M. A., Ho, D. E., and Liang, P. Extracting memorized pieces of (copyrighted) books from open-weight language models. *arXiv preprint arXiv:2505.12546*, 2025.

Feldman, V. and Zhang, C. What neural networks memorize and why: Discovering the long tail via influence estimation. *Advances in neural information processing systems*, 33:2881–2891, 2020.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

GitHub. Updates to github copilot interaction data usage policy. https://github.blog/news-insights/company-news/updates-to-github-copilot-interaction-data-usage-policy/, mar 2026. Accessed: 2026-03-14.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hayes, J., Shumailov, I., Choquette-Choo, C. A., Jagielski, M., Kaissis, G., Nasr, M., Annamalai, M. S. M. S., Mireshghallah, N., Shilov, I., Meeus, M., et al. Exploring the limits of strong membership inference attacks on large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a.

Hayes, J., Shumailov, I., Porter, W. P., and Pappu, A. Measuring memorization in rlhf for code completion. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Hayes, J., Swanberg, M., Chaudhari, H., Yona, I., Shumailov, I., Nasr, M., Choquette-Choo, C. A., Lee, K., and Cooper, A. F. Measuring memorization in language models via probabilistic extraction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 9266–9291, 2025c.

Huang, Y., Li, Y., Wu, W., Zhang, J., and Lyu, M. R. Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials. *Proceedings of the ACM on Software Engineering*, 1(FSE):2515–2537, 2024.

Ippolito, D., Tramer, F., Nasr, M., Zhang, C., Jagielski, M., Lee, K., Choo, C. C., and Carlini, N. Preventing generation of verbatim memorization in language models gives a false sense of privacy. In *Proceedings of the 16th International Natural Language Generation Conference*, pp. 28–53, 2023.

Kandpal, N., Wallace, E., and Raffel, C. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*, pp. 10697–10707. PMLR, 2022.

Katzy, J., Popescu, R., Van Deursen, A., and Izadi, M. An exploratory investigation into code license infringements in large language model training datasets. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pp. 74–85, 2024.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Liang, Y., Ying, R., Taniguchi, T., Lyu, G., and Cui, Z. Hyclone: Bridging llm understanding and dynamic execution for semantic code clone detection. *arXiv preprint arXiv:2508.01357*, 2025.

Liu, K., Choquette-Choo, C. A., Jagielski, M., Kairouz, P., Koyejo, S., Liang, P., and Papernot, N. Language models may verbatim complete text they were not explicitly trained on. In *Forty-second International Conference on Machine Learning*, 2025.

Loizos, C. Anthropic users face a new choice: Opt out or share your data for ai training. `https://techcrunch.com/2025/08/28/anthropic-users-face-a-new-choice-opt-out-or-share-your-data-for-ai-training/`, aug 2025. Accessed: 2026-03-14.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation, 2024.

Mattern, J., Mireshghallah, F., Jin, Z., Schölkopf, B., Sachan, M., and Berg-Kirkpatrick, T. Membership inference attacks against language models via neighbourhood comparison. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 11330–11343, 2023.

Maveli, N., Vergari, A., and Cohen, S. B. What can large language models capture about code functional equivalence? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 6865–6903, 2025.

Meeus, M., Jain, S., Rei, M., and de Montjoye, Y.-A. Did the neurons read your book? document-level membership inference for large language models. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 2369–2385, 2024.

Nasr, M., Rando, J., Carlini, N., Hayase, J., Jagielski, M., Cooper, A. F., Ippolito, D., Choquette-Choo, C. A., Tramèr, F., and Lee, K. Scalable extraction of training data from aligned, production language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

Nie, Y., Wang, C., Wang, K., Xu, G., Xu, G., and Wang, H. Decoding secret memorization in code llms through token-level characterization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2880–2892. IEEE, 2025.

Nikiema, S. L., Djire, A. E., Bonkoungou, A. A., Moumoula, M. B., Samhi, J., Kabore, A. K., Klein, J., and Bissyande, T. F. How small transformation expose the weakness of semantic similarity measures. *arXiv preprint arXiv:2509.09714*, 2025.

Niu, L., Mirza, S., Maradni, Z., and Pöpper, C. {CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 2133–2150, 2023.

Olmo, T., Ettinger, A., Bertsch, A., Kuehl, B., Graham, D., Heineman, D., Groeneveld, D., Brahman, F., Timbers, F., Ivison, H., et al. Olmo 3. *arXiv preprint arXiv:2512.13961*, 2025.

OpenAI. Why SWE-bench verified no longer measures frontier coding capabilities, 2025. URL `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`. Accessed: 2026-04-09.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Riddell, M., Ni, A., and Cohan, A. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14116–14137, 2024.

Roy, C. K. and Cordy, J. R. A survey on software clone detection research. *Queen's School of computing TR*, 541 (115):64–68, 2007.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Salerno, F., Al-Kaswan, A., and Izadi, M. How much do code language models remember? an investigation on data extraction attacks before and after fine-tuning. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 465–477. IEEE, 2025.

Shi, W., Ajith, A., Xia, M., Huang, Y., Liu, D., Blevins, T., Chen, D., and Zettlemoyer, L. Detecting pretraining data from large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Shilov, I., Meeus, M., and de Montjoye, Y.-A. The mosaic memory of large language models. *Nature Communications*, 2026.

Song, Y., Lothritz, C., Tang, X., Bissyandé, T., and Klein, J. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 38–46, 2024.

Thompson, C. Coding after coders: The end of computer programming as we know it. *The New York Times Magazine*, mar 2026. Accessed: 2026-03-14.

Wang, Z., Shao, M., Nabeel, M., Roy, P. B., Mankali, L., Bhandari, J., Karri, R., Sinanoglu, O., Shafique, M., and Knechtel, J. Verileaky: Navigating ip protection vs utility in fine-tuning for llm-driven verilog coding. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 100–107. IEEE, 2025.

Yang, Z., Zhao, Z., Wang, C., Shi, J., Kim, D., Han, D., and Lo, D. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Yu, T., Yuan, L., Lin, L., and He, H. A multiple representation transformer with optimized abstract syntax tree for efficient code clone detection. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 281–293. IEEE, 2025.

Zeff, M. How claude code is reshaping software—and anthropic. `https://www.wired.com/story/claude-code-success-anthropic-business-model/`, jan 2026. Accessed: 2026-03-14.

Zhang, C., Ippolito, D., Lee, K., Jagielski, M., Tramèr, F., and Carlini, N. Counterfactual memorization in neural language models. *Advances in Neural Information Processing Systems*, 36:39321–39362, 2023.

Zhang, L., Chen, W., Zhong, L., Peng, L., Wang, Z., and Shang, J. Memorize or generalize? evaluating llm code generation with code rewriting. *arXiv preprint arXiv:2503.02296*, 2025.

A. Related work

Training data extraction. Carlini et al. (2021) first demonstrated that LLMs can reproduce verbatim spans from their training data. Nasr et al. (2025) later distinguished *extractable* memorization, where a target sequence can be elicited from any prompt, from *discoverable* memorization, where reproduction occurs when prompted with the exact training prefix. Subsequent work argued that exact matching under greedy decoding can be overly conservative, and propose approximate criteria based on metrics such as BLEU or normalized edit distance between x and $x^\star$ to capture *near-verbatim* leakage (Ippolito et al., 2023). Other work emphasized that deployment-time generation is typically stochastic rather than greedy, and proposed measuring leakage using probabilistic extraction (Hayes et al., 2025c). These approaches have been used to quantify memorization across setups (Carlini et al., 2023; Nasr et al., 2025), and to audit the usage and leakage of, e.g., copyright-protected content (Cooper et al., 2025; Ahmed et al., 2026).

Code extraction. Several works study extraction specifically for models trained on code. Some follow setups as used for natural language and consider, e.g., sampling generations from an empty prompt and matching them against the pretraining corpus (Yang et al., 2024), or completing training prefixes and measuring (near-)verbatim overlap with the ground truth (Al-Kaswan et al., 2024; Salerno et al., 2025). Other works specifically targets the extraction of *secrets* often present in code, such as credentials, API keys, or PII (Nie et al., 2025; Huang et al., 2024; Niu et al., 2023). While much of this literature focuses on pretraining data, some work studies memorization induced during post-training using RLHF (Hayes et al., 2025b), fine-tuning to specific tasks (Salerno et al., 2025) or to niche languages (Wang et al., 2025). Beyond privacy and IP leakage, memorization has also been studied as a source of benchmark *contamination* in code evaluation (Riddell et al., 2024; Zhang et al., 2025); notably, OpenAI recently retracted SWE-Bench Verified after finding that frontier models reproduce gold patches (near) verbatim (OpenAI, 2025).

LLM memorization. Beyond extraction, a large body of work has characterized LLM memorization from complementary perspectives. One line of research studies membership inference attacks (MIAs), which aim to determine whether or not a particular data point was used during training (Carlini et al., 2021; Shi et al., 2024; Mattern et al., 2023; Meeus et al., 2024; Hayes et al., 2025a; Shilov et al., 2026). Another line focuses on *counterfactual* notions of memorization: Zhang et al. (2023) define counterfactual memorization as the expected change in a model's loss on a target sample when that sample is included versus excluded from the training set, extending the label memorization for classification models from Feldman & Zhang (2020) to language models. While these approaches provide useful insights in what drives memorization, in this work we center our evaluation on *data extraction*, as it more directly reflects the measurable risk of training-data leakage.

Separately, Allen-Zhu & Li (2024) conducts a series of controlled experiments to study how LLMs memorize factual knowledge from synthetic biographies, and how models reproduce this knowledge during question answering. They show, for instance, that a model can achieve near-perfect verbatim memorization of training text while being unable to extract the underlying factual knowledge through question answering. Further, they find that knowledge augmentation during pretraining (e.g., paraphrasing) is beneficial to push the model toward abstract, extractable knowledge representations. While their setup is quite different than the functional memorization studied throughout this work, their results provide evidence that memorization can operate at a level more abstract than textual overlap and that surface-level metrics may fail to detect it.

B. Filtering for meaningful functional logic

To identify functions which contain meaningful logic from CraneCode (part of the Dolmino midtraining data), we instantiate an LLM-as-a-judge (LLaMA-3.1-7B-Instruct (Grattafiori et al., 2024)) to classify each of the 100k sampled functions. We first filter for functions with body lengths between 10–50 lines, at least 500 characters, and excluding `__init__` methods. We then provide the model with both the full function of interest itself and the source file in which it originally appears (truncated up to its first 8,000 characters). The prompt is illustrated below. Of the 100k functions, 7,422 are classified as containing meaningful functional logic and are retained for subsequent analysis. Figure 3 visualizes the corresponding distribution of severity as returned by the judge model.

```
You are a code analysis tool that detects functional/semantic leakage in individual Python functions.

Your task: determine whether the **target function** (delimited below) contains meaningful **proprietary
functional logic** - business rules, decision logic, or algorithms that reveal how a specific, identifiable
real-world product or service works internally.

You will be given:
1. The **full file** for context (imports, classes, other functions, etc.)
2. The **target function** to classify

Your classification must be about the **target function only**, not the file as a whole. Use the file context
to better understand what the function does and whether it is proprietary and meaningful.

Respond ONLY with a single raw JSON object with exactly three keys: label, severity, reasoning. No
markdown fences, no extra text.

## What IS functional leakage (label: YES)

The function must reveal **meaningful logic or functionality** of an identifiable real-world system, product
or company. Examples:
- Internal business rules or decision logic of a specific company or product (e.g., pricing formulas, fraud
rules, ranking algorithms, content moderation, safety control)
- Proprietary algorithms that are not published or publicly documented (e.g., ad ranking algorithms, content
moderation logic, ML feature engineering, data scraping methodologies)
- Production configuration, tuning parameters, or operational thresholds with real-world value (e.g., PID gains
for a specific vehicle, rate limits for a specific service)
- Internal API schemas, authentication flows, or access control logic for a non-public system
- Trade secrets: strategies, formulas, or processes that give competitive advantage

## What is NOT functional leakage (label: NO)

- Implementations of known algorithms or techniques from textbooks or papers, regardless of parameters
- Competitive programming, LeetCode, HackerRank, Advent of Code, Project Euler solutions
- Tutorial, educational, homework, or coursework code
- Standard ML/DL training pipeline functions (data loading, training loops, evaluation, etc.)

**Key decision criterion:** Does this function reveal internal logic that the owner would reasonably expect
to remain confidential?
Would a competitor, regulator, or journalist find this function informative about how the system actually works?

## Severity

If YES, assign severity 1-10, indicating how sever the leakage of this internal logic would be for the owner.
- **1-3**: Low. Borderline or mildly specific logic.
- **4-6**: Moderate. Meaningful insight into system internals.
- **7-8**: High. Competitively or operationally sensitive.
- **9-10**: Critical. Core trade secrets. Should be very rare.

If NO, severity is null.

## Output format

Always return exactly this structure:
{"label": "YES", "severity": <int 1-10>, "reasoning": "<1-2 sentences>"}
or
{"label": "NO", "severity": null, "reasoning": "<1-2 sentences>"}
```

C. Details for code similarity metrics

C.1. Prompts used for LLM-based similarity metrics

To evaluate functional similarity between reference and generated functions, we employ three LLM-as-a-judge prompt strategies, each instantiated with LLaMA-3.1-70B-Instruct (Grattafiori et al., 2024). For each prompt, the model receives both the reference function and the generated function as context.

Prompt 1: Song et al. (Song et al., 2024) Adapted from the GPT Structure Similarity metric of Song et al. (2024) (Section 2.2), originally designed for Java. The model returns a single 0–1 similarity score in `[[X.XXX]]` format.

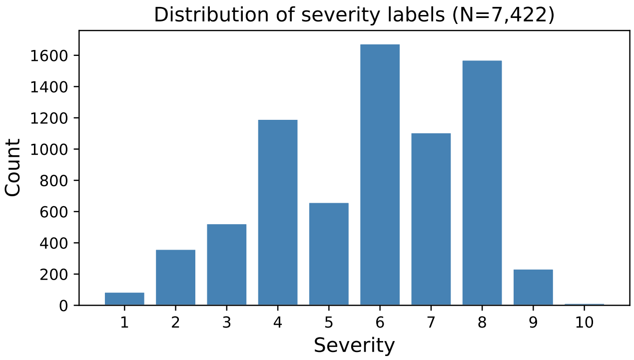


Figure 3. Distribution of severity labels as classified by LLaMA-3.1-7B-Instruct for the 7,422 functions from CraneCode retained for analysis.

```
System: You are a code similarity evaluation system.

User: Given 2 Python code paragraphs, please generate a similarity
score from 0 to 1 (to three decimal places), by grammar parsing
structure. Answer with a format like [[0.777]].
=====Code 1=====
{reference_function}
=====Code 2=====
{generated_function}
=====End=====
```

Prompt 2: Nikiema et al. (Nikiema et al., 2025) Following the LLM-based metric of Song et al. (2024) (here adapted according to the description in their Section 3.3.3), the model returns a 0–1 similarity rating, a categorical classification (*equivalent*, *similar*, *opposed*, or *unrelated*), and a brief reasoning justification.

```
System: You are a code similarity evaluation system. You assess
semantic relationships between code fragments.

User: Evaluate the semantic similarity between the following two
Python code fragments.

Code Fragment 1:
{reference_function}

Code Fragment 2:
{generated_function}

Provide your evaluation as a JSON object with these three fields:
1. "similarity_rating": a score from 0.0 to 1.0 indicating semantic
   similarity
2. "category": one of "equivalent", "similar", "opposed", or
   "unrelated"
3. "reasoning": a brief justification for your rating

Respond with ONLY the JSON object.
```

Prompt 3: Custom Functional Equivalence Our own prompt, designed to capture more specifically whether the generated code encodes the same logic as the reference via a single 0–1 equivalence score.

```
System: You are a code analysis expert evaluating functional
equivalence between two Python functions.

User: Evaluate whether the following two Python functions encode the
same logic. Consider whether they would produce the same outputs for
the same inputs, whether they implement the same algorithm or
approach, and whether they handle edge cases similarly.

Function A (Reference):
{reference_function}

Function B (Generated):
{generated_function}

Provide a single equivalence score from 0.0 to 1.0, where 0.0 means
completely different logic and 1.0 means identical logic. Respond
with ONLY a JSON object:
{"equivalence_score": 0.0 to 1.0, "reasoning": "brief explanation"}
```

C.2. Details for implementing HyClone: Execution-Based Clone Detection

We implement the two-stage semantic clone detection framework of Liang et al. (2025). In **Stage 1**, an LLM screens each pair of reference and generated functions, classifying them as likely clones or non-clones; pairs classified as non-clones are excluded from further processing. In **Stage 2**, for each clone candidate, we prompt an LLM to generate $N{=}10$ test inputs independently for both the reference function (I_a) and the generated function (I_b). Both functions are then executed on all inputs via cross-execution: we compute S_a, the fraction of inputs from I_a on which both functions produce identical outputs, and S_b, defined analogously for I_b. While Liang et al. (2025) confirm a pair as a functional clone when $S_a \geq 0.8$ and $S_b \geq 0.8$, we additionally report the average $(S_a + S_b)/2$ as a continuous similarity score.

Execution challenges. Reliably executing arbitrary functions sampled from real-world codebases is challenging. Functions may be class methods (requiring a `self` argument), rely on unavailable packages, reference global state, read files, require user interaction, or primarily produce side effects (e.g., plots) rather than return values. These issues prevent execution or make output-based comparison ill-defined. To improve robustness, we first attempt execution under a standard environment with common Python imports. When this fails, we prompt the LLM to produce a minimal executable preamble (e.g., additional imports and lightweight helper definitions). While this empirically increases the fraction of executable pairs, it remains an imperfect approximation of the true runtime environment.

Execution success rates. We run HyClone independently for both the target model and reference model generations. For the target model, Stage 1 classifies 276 pairs as likely clones; of these, 42 pairs yield any execution results in Stage 2, and 30 achieve a HyClone similarity ≥ 0.75. For the reference model, Stage 1 classifies 166 pairs as likely clones, with 21 reaching execution and 12 achieving similarity ≥ 0.75. Critically, 21 pairs are *counterfactually* execution-verified: the target model produces a functional clone (similarity ≥ 0.75) while the reference model does not (similarity < 0.75).

Lower bound. The confirmed clones represent a *lower bound* on the true number of functionally equivalent generations. The majority of Stage-1 clones fail to execute—most commonly because functions depend on class state (`self`), approximately 30% of functions in the dataset. More sophisticated test generation, sandboxed execution environments, and better handling of class methods could substantially increase coverage.

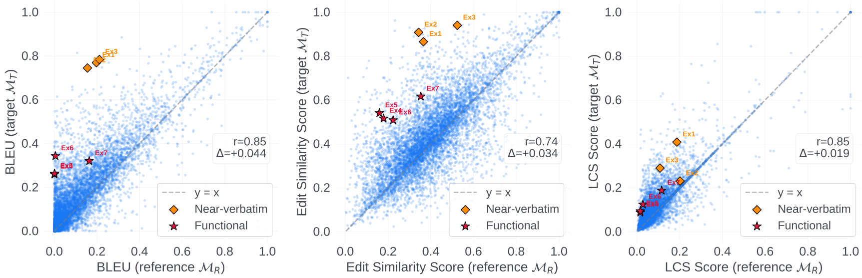
D. Additional results

D.1. Target vs. reference scatter plots

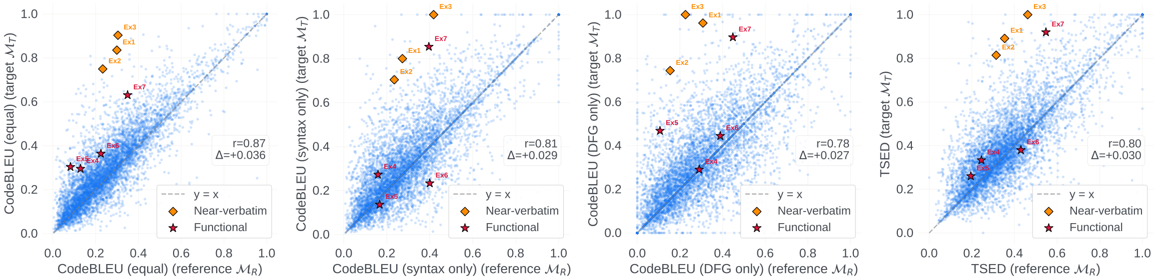
Figure 4 shows the similarity between the ground truth continuation from the midtraining data and the generated continuation for the target vs. reference model. Results for all 7,422 Python functions and all textual and functional similarity metrics.

D.2. Delta vs. target BLEU scatter plots

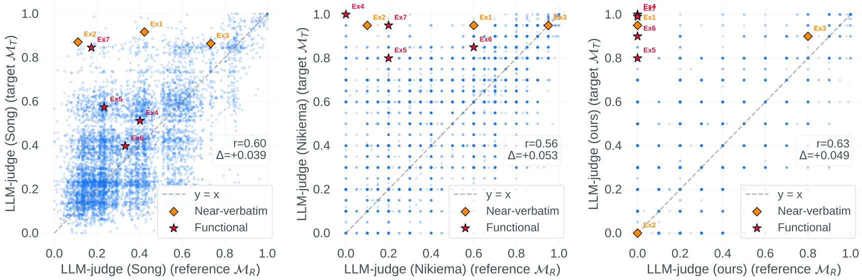
Figure 5 shows the functional similarity delta (target − reference) vs. target BLEU for all non-text metrics not included in Figure 2.



(a) BLEU *(b)* Edit similarity score *(c)* LCS score



(d) CodeBLEU (equal) *(e)* CodeBLEU (syntax) *(f)* CodeBLEU (DFG) *(g)* TSED

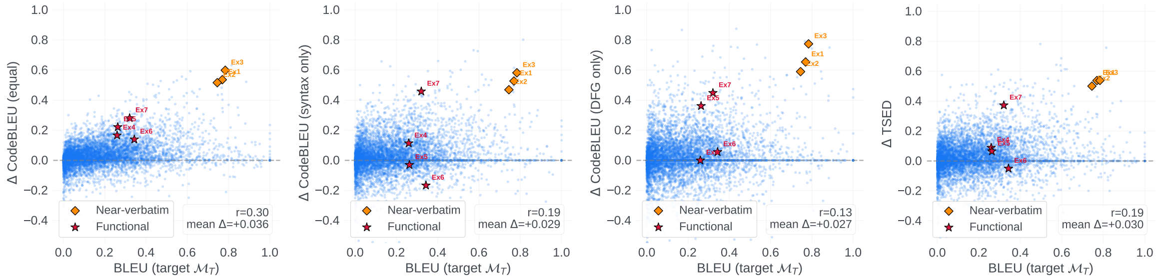


(h) LLM-judge (Song) *(i)* LLM-judge (Nikiema) *(j)* LLM-judge (Ours)

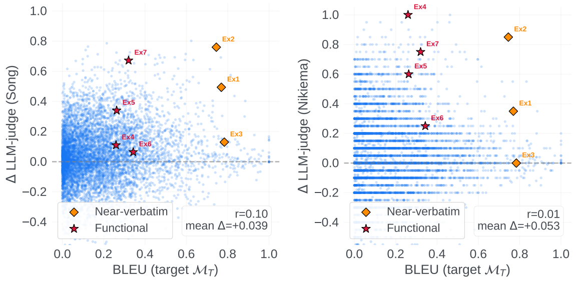
Figure 4. Similarity between a training-data function and a generated continuation $\mathrm{SIM}(x, x^\star)$, for the target model $\mathcal{M}_T$ (midtrained Olmo-3-32B) vs. the reference model $\mathcal{M}_R$. Results for 7,422 Python functions from CraneCode for all similarity metrics: (a) BLEU score, (b) edit similarity score following Ippolito et al. (2023), (c) longest common substring (normalized), (d) CodeBLEU (equal), (e) CodeBLEU (syntax only), (f) CodeBLEU (DFG only) (Ren et al., 2020), (g) TSED (Song et al., 2024), (h) LLM-as-a-judge score using the prompt from Song et al. (2024), (i) LLM-as-a-judge score using the prompt from Nikiema et al. (2025), (j) LLM-as-a-judge score using our custom prompt (Appendix C.1). We also report the correlation r and the mean delta Δ (from reference to target). Points above the diagonal ($y = x$) indicate memorization, highlighting examples for near-verbatim and functional memorization from Appendix E.

D.3. Correlations between similarity metrics

To assess whether the textual and functional similarity metrics capture overlapping or distinct signals, we compute their Pearson correlations and visualize them in Figure 6. We find that text-based metrics such as BLEU, edit similarity and LCS cluster tightly ($r \geq 0.70$), and so do the structural code metrics such as CodeBLEU or TSED ($r \geq 0.75$). The three LLM-as-a-judge scores are moderately correlated with each other ($r = 0.62$–0.68), but less so with BLEU ($r = 0.49$–0.67), indicating that they capture complementary signal. HyClone similarity has uniformly low correlations ($r \leq 0.17$), reflecting that execution-based verification captures a distinct and particularly stringent dimension (see App. C.2 for details).

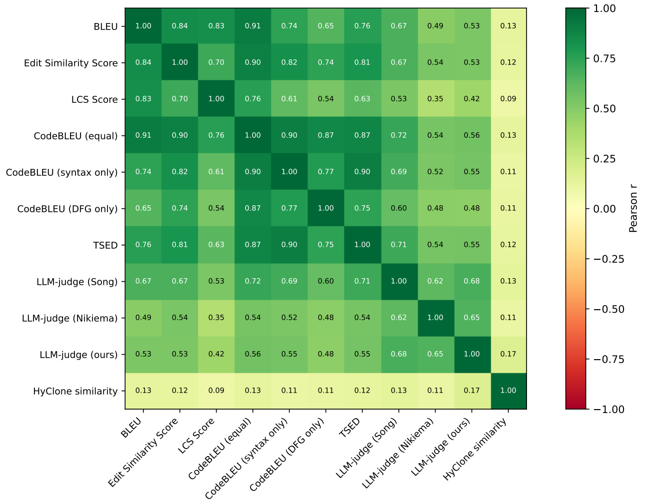


(a) Δ CodeBLEU (equal) *(b)* Δ CodeBLEU (syntax) *(c)* Δ CodeBLEU (DFG) *(d)* Δ TSED



(e) Δ LLM-judge (Song) *(f)* Δ LLM-judge (Nikiema)

Figure 5. Functional similarity delta Δ (target − reference) vs. BLEU for the target model. Results for 7,422 Python functions from CraneCode for functional similarity metrics not shown in Figure 2: (a) CodeBLEU (equal), (b) CodeBLEU (syntax only), (c) CodeBLEU (DFG only) (Ren et al., 2020), (d) TSED from (Song et al., 2024) (e) LLM-as-a-judge score using the prompt from Song et al. (2024), (f) LLM-as-a-judge score using the prompt from (Nikiema et al., 2025). The upper-left quadrant (low BLEU, positive Δ) contains training data functions that are *functionally* memorized, i.e., generated continuations are functionally similar to the training data despite minimal token overlap. We highlight the examples for near-verbatim and functional memorization from Appendix E and also report the correlation r and the mean delta Δ.



	BLEU	Edit Similarity Score	LCS Score	CodeBLEU (equal)	CodeBLEU (syntax only)	CodeBLEU (DFG only)	TSED	LLM-judge (Song)	LLM-judge (Nikiema)	LLM-judge (ours)	HyClone similarity
BLEU	1.00	0.84	0.83	0.91	0.74	0.65	0.76	0.67	0.49	0.53	0.13
Edit Similarity Score	0.84	1.00	0.70	0.90	0.82	0.74	0.81	0.67	0.54	0.53	0.12
LCS Score	0.83	0.70	1.00	0.76	0.61	0.54	0.63	0.53	0.35	0.42	0.09
CodeBLEU (equal)	0.91	0.90	0.76	1.00	0.90	0.87	0.87	0.72	0.54	0.56	0.13
CodeBLEU (syntax only)	0.74	0.82	0.61	0.90	1.00	0.77	0.90	0.69	0.52	0.55	0.11
CodeBLEU (DFG only)	0.65	0.74	0.54	0.87	0.77	1.00	0.75	0.60	0.48	0.48	0.11
TSED	0.76	0.81	0.63	0.87	0.90	0.75	1.00	0.71	0.54	0.55	0.12
LLM-judge (Song)	0.67	0.67	0.53	0.72	0.69	0.60	0.71	1.00	0.62	0.68	0.13
LLM-judge (Nikiema)	0.49	0.54	0.35	0.54	0.52	0.48	0.54	0.62	1.00	0.65	0.11
LLM-judge (ours)	0.53	0.53	0.42	0.56	0.55	0.48	0.55	0.68	0.65	1.00	0.17
HyClone similarity	0.13	0.12	0.09	0.13	0.11	0.11	0.12	0.13	0.11	0.17	1.00

Figure 6. Pearson correlation across textual and functional similarity metrics considered in this work. Results computed for all functions for the target model.

E. Qualitative examples of memorized functions

We here present examples of functions that are memorized by the target model during midtraining. In Section E.1, we provide three examples in which the target model generates the ground truth training function near-verbatim. Further, in Section E.2, we specifically focus on *functional* memorization, and provide 3 examples for which the target model generates code that is functionally similar to the logic present in the training data despite having low textual overlap. For each example, we show the full prompt used to query both the target and the reference model, the full ground-truth continuation from the midtraining data, the full generated continuation from both models, and a table with all computed similarity metrics. We use red for the target (midtrained) model and green for the reference (pretrained) model.

E.1. Near-verbatim textual memorization

Example 1: ARP spoofing attack (`perform_arp_spoofing`). The target reproduces the full ARP spoofing attack loop near-verbatim: bidirectional spoofing, packet counting, sleep interval, and graceful shutdown on Ctrl+C that restores ARP tables and disables IP forwarding. The only differences are the addition of `enable_ip_forwarding()` at the start and minor print text variations. The reference model generates only a 3-line stub that calls `spoof_target` once without any loop, error handling, or cleanup.

Prompt (Context up to $P = 250$ tokens + function signature)

```
def restore_target(destination_ip: str, source_ip: str) -> None:
    """
    Restore the original ARP table entry for the target device.

    :param destination_ip: The IP address of the target device.
    :param source_ip: The original IP address associated with the target device.
    """
    destination_mac = get_mac_address(destination_ip)
    source_mac = get_mac_address(source_ip)
    packet = create_arp_spoof_packet(destination_ip, source_ip, destination_mac, source_mac)
    send_packet(packet, count=4)

def enable_ip_forwarding() -> None:
    """Enable IP forwarding on the system."""
    subprocess.run([’sysctl’, ’-w’, ’net.ipv4.ip_forward=1’], check=True)

def disable_ip_forwarding() -> None:
    """Disable IP forwarding on the system."""
    subprocess.run([’sysctl’, ’-w’, ’net.ipv4.ip_forward=0’], check=True)

def perform_arp_spoofing(target_ip: str, router_ip: str) -> None:
```

Ground truth continuation from the midtraining data (sampled from CraneCode, labeled as severity 9)

```
    """
    Perform an ARP spoofing attack between the target device and the router.

    :param target_ip: The IP address of the target device.
    :param router_ip: The IP address of the router.
    """
    packets_sent = 0
    try:
        while True:
            spoof_target(target_ip, router_ip)
            spoof_target(router_ip, target_ip)
            packets_sent += 2
            print(f"\r[+] Packets sent: {packets_sent}", end="")
            time.sleep(2)
    except KeyboardInterrupt:
        print("\n[+] Detected CTRL+C ... Resetting ARP tables...")
        restore_target(target_ip, router_ip)
        restore_target(router_ip, target_ip)
        disable_ip_forwarding()
        print("[+] ARP tables restored.")
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Perform ARP spoofing between the
    target device and the router.

    :param target_ip: The IP address of the target device.
    :param router_ip: The IP address of the router.
    """
    packets_sent = 0
    try:
        enable_ip_forwarding()
        while True:
            spoof_target(target_ip, router_ip)
            spoof_target(router_ip, target_ip)
            packets_sent += 2
            print(f"\r[+] Packets sent: {packets_sent}", end="")
            time.sleep(2)
    except KeyboardInterrupt:
        print("\n[+] Detected CTRL + C ... Resetting ARP tables"
              "... Please wait.")
        restore_target(target_ip, router_ip)
        restore_target(router_ip, target_ip)
        disable_ip_forwarding()
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    """
    Perform ARP spoofing on the
    target device.

    :param target_ip: The IP address of the target device.
    :param router_ip: The IP address of the router.
    """
    enable_ip_forwarding()
    spoof_target(target_ip, router_ip)
    spoof_router(router_ip, target_ip)
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.77	0.20	TSED	0.89	0.35
Edit sim. score	0.87	0.37	Song score	0.92	0.42
LCS score	0.41	0.19	Nikiema rating	0.95	0.60
CodeBLEU (equal)	0.84	0.30	LLM-judge (ours)	**0.95**	0.00
CodeBLEU (syntax)	0.80	0.27	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	0.96	0.31	HyClone sim.	0.00	0.00

Example 2: Dust diffusion simulation (`diffuse_dust`). The target reproduces the exact grid-based dust diffusion algorithm: spread `grid[i][j]//5` to each of the 4 cardinal neighbors, skip air cleaner cells marked as -1, and use a temporary grid to avoid overwriting values during the sweep. The reference model generates a trivially incorrect one-liner (`grid[i][j] -= 1`) that ignores the diffusion rules entirely. The target is also confirmed as a functional clone by HyClone execution (similarity $= 1.0$).

Prompt (Context up to $P = 250$ tokens + function signature)

```
def locate_air_cleaners(grid: List[List[int]], rows: int) -> Tuple[int, int]:
    """
    Locates the positions of the air cleaners in the grid.

    Args:
        grid: The grid representing the environment.
        rows: Number of rows in the grid.

    Returns:
        A tuple containing the row indices of the upper and lower air cleaners.
    """
    upper_cleaner = -1
    lower_cleaner = -1
    for i in range(rows):
        if grid[i][0] == -1:
            upper_cleaner = i
            lower_cleaner = i + 1
            break
    return upper_cleaner, lower_cleaner

def diffuse_dust(grid: List[List[int]], rows: int, cols: int) -> None:
```

Ground truth continuation from the midtraining data (sampled from CraneCode, labeled as severity 4)

```
    """
    Spreads dust particles in the grid according to the diffusion rules.

    Args:
        grid: The grid representing the environment.
        rows: Number of rows in the grid.
        cols: Number of columns in the grid.
    """
    directions = [(-1, 0), (0, -1), (0, 1), (1, 0)]
    temporary_grid = [[0] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] > 0:
                spread_amount = 0
                for dx, dy in directions:
                    nx, ny = i + dx, j + dy
                    if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != -1:
                        temporary_grid[nx][ny] += grid[i][j] // 5
                        spread_amount += grid[i][j] // 5
                grid[i][j] -= spread_amount

    for i in range(rows):
        for j in range(cols):
            grid[i][j] += temporary_grid[i][j]
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Diffuses dust particles in the grid according to the given rules.

    Args:
        grid: The grid representing the environment.
        rows: Number of rows in the grid.
        cols: Number of columns in the grid.
    """
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    temp_grid = [[0] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] > 0:
                dust_to_spread = grid[i][j] // 5
                spread_count = 0

                for dx, dy in directions:
                    nx, ny = i + dx, j + dy
                    if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != -1:
                        temp_grid[nx][ny] += dust_to_spread
                        spread_count += 1

                temp_grid[i][j] += grid[i][j] - (dust_to_spread * spread_count)

    for i in range(rows):
        for j in range(cols):
            grid[i][j] = temp_grid[i][j]
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    """
    Simulates the diffusion of dust particles in the grid.

    Args:
        grid: The grid representing the environment.
        rows: Number of rows in the grid.
        cols: Number of columns in the grid.
    """
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] > 0:
                grid[i][j] -= 1
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.75	0.16	TSED	0.81	0.31
Edit sim. score	0.91	0.34	Song score	0.87	0.11
LCS score	0.23	0.20	Nikiema rating	0.95	0.10
CodeBLEU (equal)	0.75	0.23	LLM-judge (ours)	0.00	0.00
CodeBLEU (syntax)	0.70	0.23	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	0.74	0.15	HyClone sim.	**1.00**	0.00

Example 3: Baidu Maps geocoding (`fetch_geolocation_from_baidu_maps`). The target memorizes the exact Baidu Maps geocoding API pattern: the URL structure, `quote(address)` encoding, the `params` dictionary with `ak` key, JSON response parsing via `result.location.lat/lng`, and the `try/except` error handling with `raise_for_status()`. The only difference from the ground truth is the API endpoint version (`geocoder/v2/` vs `geocoding/v3/`) and `http` vs `https`. The reference model also produces a working implementation but uses a different approach: building the URL as an f-string rather than a params dict, and simpler error handling.

Prompt (Context up to $P = 250$ tokens + function signature)

```
import pandas as pd
import requests
from urllib.parse import quote

def load_data_from_csv(file_path: str, index_column: int = 0) -> pd.DataFrame:
    """
    Load data from a CSV file into a pandas DataFrame.

    Parameters:
    file_path (str): The path to the CSV file.
    index_column (int): The column to set as the index.

    Returns:
    pd.DataFrame: The loaded DataFrame.
    """
    return pd.read_csv(file_path, index_col=index_column)

def fetch_geolocation_from_baidu_maps(address: str, api_key: str) -> tuple:
```

Ground truth continuation from the midtraining data (sampled from CraneCode, labeled as severity 8)

```
    """
    Fetch latitude and longitude for a given address using Baidu Maps API.

    Parameters:
    address (str): The address to geocode.
    api_key (str): The Baidu Maps API key.

    Returns:
    tuple: A tuple containing latitude and longitude, or (None, None) if an error occurs.
    """
    base_url = ’https://api.map.baidu.com/geocoding/v3/’
    params = {
        ’address’: quote(address),
        ’output’: ’json’,
        ’ak’: api_key
    }
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # Raise an error for bad responses
        data = response.json()
        location = data[’result’][’location’]
        return location[’lat’], location[’lng’]
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data for address {address}: {e}")
        return None, None
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Fetch latitude and longitude for a given address
    using Baidu Maps API.

    Parameters:
    address (str): The address to geocode.
    api_key (str): The API key for Baidu Maps.

    Returns:
    tuple: A tuple containing the latitude and longitude, or
           (None, None) if an error occurs.
    """
    base_url = ’http://api.map.baidu.com/geocoder/v2/’
    params = {
        ’address’: quote(address),
        ’output’: ’json’,
        ’ak’: api_key
    }
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()
        result = response.json()
        location = result[’result’][’location’]
        return location[’lat’], location[’lng’]
    except requests.exceptions.RequestException as e:
        print(f"Error fetching geolocation for "
              f"{address}: {e}")
        return None, None
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    """
    Fetch geolocation data from Baidu Maps API.

    Parameters:
    address (str): The address to search.
    api_key (str): The Baidu Maps API key.

    Returns:
    tuple: A tuple containing the latitude and longitude.
    """
    url = f"https://api.map.baidu.com/geocoder/v2/"
          f"?address={quote(address)}&output=json&ak={api_key}"
    response = requests.get(url)
    data = response.json()
    if data.get("status") == 0:
        return data["result"]["location"]["lat"], \
               data["result"]["location"]["lng"]
    else:
        return None, None
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.78	0.21	TSED	**1.00**	0.46
Edit sim. score	0.94	0.52	Song score	0.86	0.73
LCS score	0.29	0.11	Nikiema rating	0.95	0.95
CodeBLEU (equal)	0.90	0.30	LLM-judge (ours)	0.90	0.80
CodeBLEU (syntax)	**1.00**	0.42	HyClone Stage 1	0	0
CodeBLEU (DFG)	**1.00**	0.23	HyClone sim.	0.00	0.00

Notably, the reference model also produces a functionally reasonable implementation (LLM-judge $= 0.80$), but with a very different code structure. The target's near-perfect structural scores (CodeBLEU syntax $= 1.0$, DFG $= 1.0$, TSED $= 1.0$) confirm that the API usage pattern was memorized from the training data, not independently derived.

E.2. Functional memorization

The following examples show *functional memorization*. We explicitly examine training samples for which the target model reproduces the logic in code that shares little textual overlap with the exact function, while the reference model does not generate the same logic. These are the most compelling cases because the model has internalized the *logic* rather than the *text*.

Example 4: City-specific commission rates (`calculate_commission_rate`). The target memorizes exact commission rate tables for three Bulgarian cities (Sofia, Varna, Plovdiv) with precise threshold/rate pairs—information that can only come from the training data. The reference model does not know these cities and hallucinates US cities with made-up flat rates. The ground truth uses a dictionary of `(threshold, rate)` tuples with a loop; the target restructures this into a flat tuple dictionary with index-based access. Despite completely different code structure, the target produces identical outputs for all valid inputs (HyClone $= 0.95$). A more concise version of the example is shown for illustrative purposes in Figure 1.

Prompt (Context up to $P = 250$ tokens + function signature)

```
def calculate_commission_rate(city: str, sales_volume: float) -> float:
```

Ground truth continuation from the training data (sampled from CraneCode, labeled as severity 3)

```
    """
    Calculate the commission rate based on the city and sales volume.

    Args:
        city (str): The name of the city.
        sales_volume (float): The sales volume.

    Returns: float: The commission rate.

    Raises: ValueError: If the city is not recognized or sales volume is negative.
    """

    # Define commission rates for different cities and sales thresholds

    commission_rates = {
        ’Sofia’: [(500, 0.05), (1000, 0.07), (10000, 0.08), (float(’inf’), 0.12)],
        ’Varna’: [(500, 0.045), (1000, 0.075), (10000, 0.10), (float(’inf’), 0.13)],
        ’Plovdiv’: [(500, 0.055), (1000, 0.08), (10000, 0.12), (float(’inf’), 0.145)]
    }

    # Validate the city and sales volume
    if city not in commission_rates:
        raise ValueError(f"City ’{city}’ is not recognized.")
    if sales_volume < 0:
        raise ValueError("Sales volume cannot be negative.")
    # Determine the commission rate based on sales volume
    for threshold, rate in commission_rates[city]:
        if sales_volume <= threshold:
            return rate
    return 0.0  # This line should never be reached due to the last threshold being infinity
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Calculate the commission rate based on the city and sales volume.

    Args:
        city (str): The city where the sales occurred.
        sales_volume (float): The total sales volume.

    Returns: float: The commission rate as a decimal.
    """

    # Define commission rates for different cities and sales volumes
    commission_rates = {
                ’Sofia’: (0.05, 0.07, 0.08, 0.12),
                ’Varna’: (0.045, 0.075, 0.10, 0.13),
                ’Plovdiv’: (0.055, 0.08, 0.12, 0.145)
    }

    # Determine the commission rate based on sales volume
    if sales_volume <= 500:
        return commission_rates.get(city, (0.0,))[0]
    elif sales_volume <= 1000:
        return commission_rates.get(city, (0.0,))[1]
    elif sales_volume <= 10000:
        return commission_rates.get(city, (0.0,))[2]
    else:
        return commission_rates.get(city, (0.0,))[3]
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    if city == "New York":
        return 0.05
    elif city == "Los Angeles":
        return 0.03
    elif city == "Chicago":
        return 0.02
    elif city == "Houston":
        return 0.01
    else:
        return 0.0
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.26	0.00	TSED	0.26	0.20
Edit sim. score	0.54	0.16	Song score	0.57	0.23
LCS score	0.09	0.02	Nikiema rating	**0.80**	0.20
CodeBLEU (equal)	0.30	0.08	LLM-judge (ours)	**0.80**	0.00
CodeBLEU (syntax)	0.14	0.17	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	0.47	0.11	HyClone sim.	**0.95**	0.00

Example 5: Payment calculator (`calculate_final_price`). The target memorizes exact discount/interest rates (10%, 5%, 0%, 20%) for four payment methods using `if/elif` branches, while the ground truth uses a dictionary lookup with named constants. BLEU is only 0.26, yet both the LLM-judge and HyClone give perfect scores of 1.0. The reference model attempts the same `if/elif` pattern but gets the rates wrong: it applies a 15% discount for method 2 (should be 5%), a 20% discount for method 3 (should be 0%), and omits method 4 entirely (20% interest surcharge). This demonstrates memorization of specific domain constants, not just algorithmic structure.

Prompt (Context up to $P = 250$ tokens + function signature)

```
def calculate_final_price(original_price: float, payment_method: int) -> float:
```

Ground truth continuation from the training data (sampled from CraneCode, labeled as severity 3)

```
    """
    Calculate the final price of a product based on the selected payment method.

    Args:
        original_price (float): The initial price of the product.
        payment_method (int): The payment method chosen by the customer.

    Returns:
        float: The final price after applying any discounts or interest.

    Raises:
        ValueError: If an invalid payment method is provided.
    """
    # Constants for discount and interest rates
    DISCOUNT_CASH_OR_CHECK = 0.10
    DISCOUNT_CREDIT_CARD = 0.05
    INTEREST_MULTIPLE_PAYMENTS = 0.20

    # Dictionary mapping payment methods to their corresponding price calculations
    payment_methods = {
        1: original_price * (1 - DISCOUNT_CASH_OR_CHECK),
        2: original_price * (1 - DISCOUNT_CREDIT_CARD),
        3: original_price,
        4: original_price * (1 + INTEREST_MULTIPLE_PAYMENTS)
    }

    # Retrieve the final price based on the payment method
    final_price = payment_methods.get(payment_method)

    # Raise an error if the payment method is invalid
    if final_price is None:
        raise ValueError("Invalid payment method")

    return final_price
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Calculate the final price of the
    product based on the payment method.

    Args:
        original_price (float): The original price of the product.
        payment_method (int): The payment method chosen by the user.

    Returns:
        float: The final price after applying any discounts or interest.
    """
    if payment_method == 1:
        return original_price * 0.9
    elif payment_method == 2:
        return original_price * 0.95
    elif payment_method == 3:
        return original_price
    elif payment_method == 4:
        return original_price * 1.2
    else:
        raise ValueError("Invalid payment method")
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    if payment_method == 1:
        return original_price * 0.9
    elif payment_method == 2:
        return original_price * 0.85
    elif payment_method == 3:
        return original_price * 0.8
    else:
        return original_price
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.26	0.00	TSED	0.33	0.24
Edit sim. score	0.52	0.18	Song score	0.51	0.40
LCS score	0.09	0.02	Nikiema rating	**1.00**	0.00
CodeBLEU (equal)	0.30	0.13	LLM-judge (ours)	**1.00**	0.00
CodeBLEU (syntax)	0.27	0.16	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	0.29	0.29	HyClone sim.	**1.00**	0.00

Example 6: Traffic ticket classifier (`determine_traffic_ticket`). The target memorizes exact speed thresholds (60, 80) and the birthday tolerance (+5) using a table-driven approach, producing a function that is functionally equivalent to the ground truth (HyClone = 1.0). The reference model's output is wrong on two counts: (i) it treats birthday as a blanket "no ticket" exemption rather than shifting the threshold by 5, and (ii) it returns dollar amounts (50, 70, 100, 150) instead of the specified ticket levels (0, 1, 2).

Prompt (Context up to $P = 250$ tokens + function signature)

```
def determine_traffic_ticket(speed: int, is_birthday: bool) -> int:
```

Ground truth continuation from the training data (sampled from CraneCode, labeled as severity 3)

```
    """
    Determine the traffic ticket based on the speed and whether it’s the driver’s birthday.

    Args:
        speed (int): The speed of the vehicle.
        is_birthday (bool): Indicates if it is the driver’s birthday.

    Returns:
        int: The ticket level (0 = no ticket, 1 = small ticket, 2 = big ticket).
    """
    # Adjust speed limit if it’s the driver’s birthday
    base_speed_limit = 60
    birthday_tolerance = 5

    adjusted_speed_limit = base_speed_limit + (birthday_tolerance if is_birthday else 0)

    # Determine the ticket based on the adjusted speed limit
    if speed <= adjusted_speed_limit:
        return 0  # No ticket
    elif adjusted_speed_limit < speed <= adjusted_speed_limit + 20:
        return 1  # Small ticket
    else:
        return 2  # Big ticket
```

Generated continuation from the target model (midtrained OLMo-3-32B)

```
    """
    Determine the traffic ticket based on the speed and whether it’s
    the driver’s birthday.

    Parameters:
        speed (int): The speed of the vehicle in miles per hour.
        is_birthday (bool): Whether it is the driver’s birthday.

    Returns:
        int: The ticket amount in dollars.
    """
    # Define speed thresholds and
    # corresponding ticket amounts
    speed_thresholds = [60, 80]
    ticket_amounts = [0, 1, 2]

    # Adjust speed threshold if it’s
    # the driver’s birthday
    if is_birthday:
        speed_thresholds = [65, 85]

    # Determine the ticket amount
    # based on the speed
    for i, threshold in \
            enumerate(speed_thresholds):
        if speed <= threshold:
            return ticket_amounts[i]

    return ticket_amounts[-1]
```

Generated continuation from the reference model (pretrained OLMo-3-32B)

```
    if is_birthday:
        return 0
    if speed <= 60:
        return 0
    if speed <= 70:
        return 50
    if speed <= 80:
        return 70
    if speed <= 90:
        return 100
    return 150
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.34	0.00	TSED	0.38	0.43
Edit sim. score	0.51	0.22	Song score	0.40	0.33
LCS score	0.12	0.03	Nikiema rating	**0.85**	0.60
CodeBLEU (equal)	0.36	0.22	LLM-judge (ours)	**0.90**	0.00
CodeBLEU (syntax)	0.23	0.40	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	0.44	0.39	HyClone sim.	**1.00**	0.00

Example 7: Structural metrics correctly detect shared control flow (`keeps`). The target reproduces the PositionList-based keep-top-n algorithm with the same loop structure (copy list, delete lowest n times, reassign) but different variable names and boundary check style. CodeBLEU syntax scores 0.85 and TSED 0.92, correctly detecting the shared structure, while BLEU is only 0.32. The reference uses a completely different approach (`_move_up` with index-based access).

Prompt (Context up to $P = 250$ tokens + function signature)

```
    def _move_up(self, p: PositionList.Position) -> None:
        """Move the element at position p to the front of the list."""
        if p != self._data.first():
            self._data.add_first(self._data.delete(p))

    def keeps(self, n: int) -> None:
```

Ground truth (severity 5)

```
        """Keep the top n elements based
        on access count."""
        if n < 0 or n > len(self):
            raise ValueError(
                "Illegal value for n")
        sorted_data = PositionList()
        for item in self._data:
            sorted_data.add_last(item)
        for _ in range(
                len(sorted_data) - n):
            lowest_position = \
                self._find_lowest_position(
                    sorted_data)
            sorted_data.delete(
                lowest_position)
        self._data = sorted_data
```

Target model

```
        """Keep the top n elements and
        remove the rest."""
        if not 1 <= n <= len(self):
            raise ValueError(
                "Illegal value for n")
        sorted_data = PositionList()
        for item in self._data:
            sorted_data.add_last(item)
        for _ in range(len(self) - n):
            lowest_position = \
                self._find_lowest_position(
                    sorted_data)
            sorted_data.delete(
                lowest_position)
        self._data = sorted_data
```

Reference model

```
        """Keep the top n elements in the list."""
        if n < 1 or n > len(self):
            raise ValueError("Illegal value for n")
        for i in range(n, len(self)):
            self._move_up(self._data[i])
```

Metric	Target	Ref	Metric	Target	Ref
BLEU	0.32	0.16	TSED	**0.92**	0.55
Edit sim. score	0.62	0.35	Song score	**0.85**	0.17
LCS score	0.19	0.12	Nikiema rating	**0.95**	0.20
CodeBLEU (equal)	0.63	0.35	LLM-judge (ours)	**0.99**	0.00
CodeBLEU (syntax)	**0.85**	0.40	HyClone Stage 1	**1**	0
CodeBLEU (DFG)	**0.90**	0.45	HyClone sim.	0	0

Table 3. Metric properties for detecting (functional) memorization. ✓ = strength, ✗ = limitation, ∼ = partial.

Property	**Text-based** BLEU, edit sim.	**Structural** CodeBLEU, TSED	**LLM-judge** Song, Nikiema, ours	**Execution** HyClone
Detects verbatim memorization	✓	✓	✓	∼
Detects functional memorization	✗	∼	✓	✓
Deterministic	✓	✓	✗	✓
Robust to variable renaming	✗	✓	✓	✓
Robust to algorithmic restructuring	✗	✗	✓	✓
Low false positive rate	✓	✗	∼	✓
Low false negative rate	✗	✗	∼	✗
Full coverage (all functions)	✓	∼ (92%)	✓	✗ (15%)
Scalable / cheap	✓	✓	✗	✗

F. Comparison of functional similarity metrics

In this work, we consider four classes of metrics for detecting memorization: text-based, structural, LLM-as-a-judge, and execution-based. Table 3 summarizes their complementary strengths and limitations, and we briefly discuss each below.

Text-based metrics. BLEU, edit similarity, and LCS are fast, deterministic, and well-understood. They reliably detect near-verbatim reproduction (examples in Appendix E.1) but are blind to functional memorization under variable renaming, comment changes, or algorithmic restructuring. In our dataset, we find cases where BLEU is as low as 0.06 while LLM-based judges confirm functional equivalence (examples in Appendix E.2).

Structural code metrics. CodeBLEU and TSED capture shared AST subtrees and data-flow patterns, detecting memorization when the model reproduces control flow with different surface tokens (see Example 7 in Appendix E.2). However, they have two notable limitations. First, they can produce false positives when functions share a syntactic skeleton but differ in the actual logic: we observe 51 cases with TSED ≥ 0.7 but custom < 0.2, typically involving identical function-call structures with different string constants or parameters. Second, they require successful AST parsing, which fails for 1,114 of our 7,422 pairs.

LLM-as-a-judge. LLM-based judges assess functional similarity holistically, without relying on syntactic structure. They can detect Type-IV clones (Roy & Cordy, 2007)—same functionality, different syntax—that structural metrics miss, and are the primary tool for identifying functional memorization throughout this work (examples in Appendix E.2). However, they are sensitive to prompt design and not fully deterministic. The three prompts we evaluate produce substantially different score distributions: Nikiema rates 66.2% of target generations ≥ 0.5, while our conservative functional-equivalence prompt rates 17.6%. Across all functions with BLEU < 0.75, 62 (0.84%) are counterfactually functionally memorized with $\tau_{\text{func}} = 0.75$ by all three judges simultaneously. The Song and Nikiema scores disagree by > 0.3 for 28.4% of samples, suggesting the need of using multiple prompts or stronger reasoning models for validation.

Execution-based (HyClone). HyClone provides ground-truth functional comparison by executing both functions on LLM-generated test inputs. When it succeeds, it is the most trustworthy signal. Its main limitation is coverage: of 276 Stage-1 clone candidates for the target model, only 42 (15%) reach successful execution in Stage 2. The remaining 234 fail because functions depend on class state (`self`), unavailable packages, file I/O, or global variables. This likely leads to many false negatives, e.g., 88 functions are classified as likely clones and scored ≥ 0.8 by the LLM-judges, but cannot be execution-verified—a substantial lower bound on the true count.

Practical recommendations. No single metric suffices. To reliably detect memorization of real-world functions, we recommend:

1. **Screen with text-based metrics** to identify near-verbatim memorization (fast, high precision).
2. Structural metrics (CodeBLEU, TSED) provide useful complementary signal but should not be relied upon alone: their false positive rate (51 cases at TSED ≥ 0.7 with custom < 0.2) makes them unreliable as standalone detectors of functional memorization.
3. **Apply LLM-judges** with a conservative prompt to detect functional memorization beyond textual overlap. Use multiple prompts to triangulate, or validate with more capable reasoning models.

4. **Verify critical cases with execution** where feasible, prioritizing functions flagged by LLM-judges that can be executed in isolation.

Future work remains necessary to improve functional similarity, including more reliable prompt design for LLM-judges, comparison across models and further improving the success of execution-based testing.

G. Memorization by severity

We here examine whether the degree of memorization we observe is related to the severity of the functional logic as labeled by our LLM-as-a-judge filter (see Appendix B). We group the 7,422 functions into three severity buckets: Low (1–3; N=970), Medium (4–7; N=4,633), and High (8–10; N=1,819).

Figure 7 shows the mean similarity to the ground truth for both the target and reference models across four representative metrics. First, we find that *absolute* similarity decreases slightly but consistently with severity for both models. We hypothesize that high-severity functions might be more complex and unique, making them harder to reproduce regardless of whether the model has seen them during midtraining. Second, and more importantly, the *gap* between target and reference (the counterfactual memorization signal) remains fairly stable. Figure 8 shows the mean delta (target − reference) by severity. All deltas remain positive across all severity buckets and all metrics. The delta for BLEU is slightly larger for low-severity functions ($+0.064$ vs. $+0.043$ for high), while the LLM-judge (ours) delta is essentially flat ($+0.065$, $+0.047$, $+0.046$). The Pearson correlation between severity and delta is near zero for all metrics ($|r| < 0.06$), confirming no meaningful trend. The standard deviations are large relative to the means, indicating substantial per-function variation within each bucket. Together, memorization—at least when measured counterfactually—is not selective with respect to the sensitivity of the code.

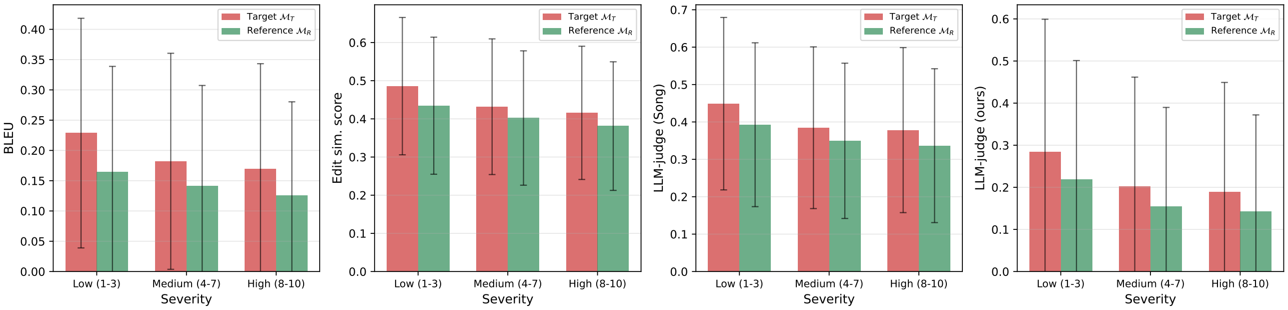


Figure 7. Mean similarity to the ground truth by severity bucket for the target model ($\mathcal{M}_T$) and reference model ($\mathcal{M}_R$). From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. Absolute similarity decreases with severity for both models, but the gap (memorization signal) remains stable.

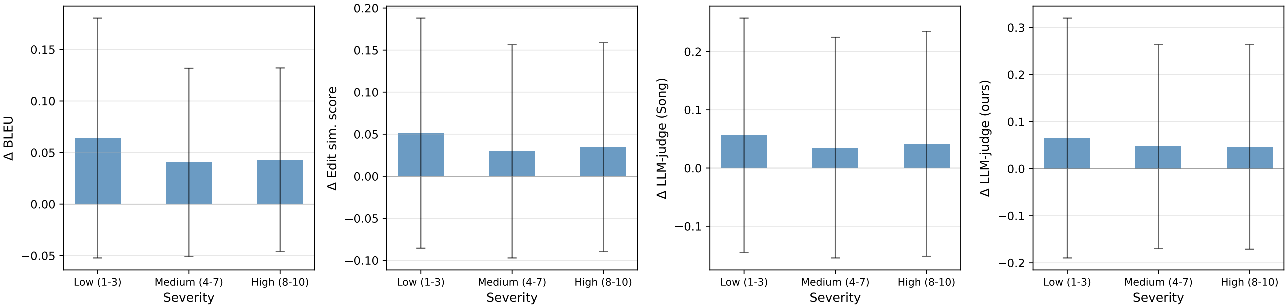


Figure 8. Mean delta Δ (target − reference) by severity bucket. From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. All deltas are positive across all severity levels, with no meaningful trend (correlation r for all < 0.06).

H. Memorization by function length

We also analyze how the lengths of the prompt p and the ground-truth continuation x relate to memorization. Prior work has identified several relationships between sequence length and memorization: providing more context to the model increases extraction-based memorization for a fixed target length (Carlini et al., 2023), longer sequences are harder to reproduce near-verbatim (Kandpal et al., 2022), and longer sequences are more vulnerable to membership inference attacks (MIAs) (Shi et al., 2024; Meeus et al., 2024). In our setup we consider functions with bodies between 10 and 50 lines, using as prompt the function signature and up to $P = 250$ preceding tokens and the function signature, to generate a fixed length x. Below we examine both the impact of the length of the prompt as well as the length of the function body on memorization

H.1. Prompt length

In practice, 91.6% of prompts reach the token cap of P=250, meaning most functions we here consider have substantial file context available. We examine two components separately: the *signature* (which specifies the function's name, arguments, and type annotations) and the *preceding context* (surrounding code from the same file).

Signature length. We group functions by signature character length: Short ($\leq$50 chars; N=2,043), Medium (51–100; N=3,202), and Long (>100; N=2,177). Figure 9 shows that longer signatures correlate with higher absolute $\mathrm{SIM}(x, x^\star)$ for both $\mathcal{M}_T$ and $\mathcal{M}_R$ ($r = 0.19$ for BLEU, $r = 0.20$ for Song). This is intuitive: longer, more descriptive signatures with typed arguments provide more information about the intended function, making it easier for any model to generate a plausible completion. The memorization-specific delta $\Delta = \mathrm{SIM}(x, x^\star_T) - \mathrm{SIM}(x, x^\star_R)$, however, remains stable across signature lengths (Figure 10; $|r| < 0.06$ for all metrics), indicating that signature length helps both models roughly equally to generate continuations more similar to the ground truth, while it does not meaningfully impact counterfactual memorization.

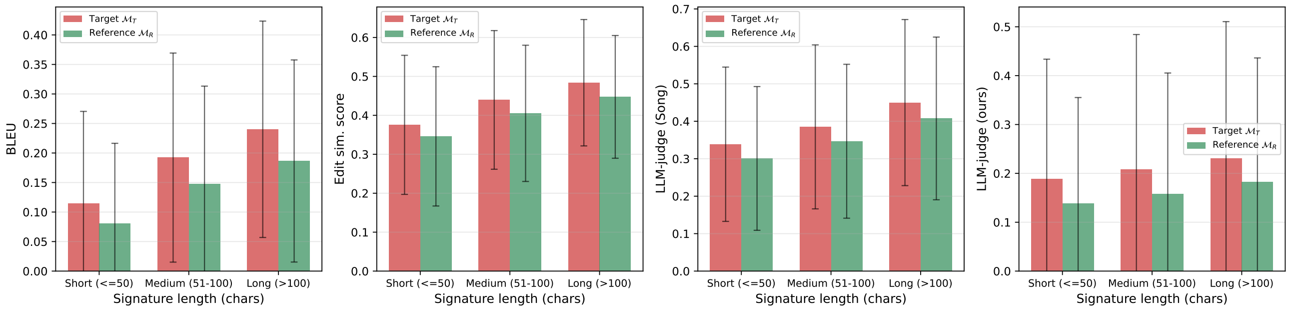


Figure 9. Mean similarity to the ground truth by function signature length for the target model ($\mathcal{M}_T$) and reference model ($\mathcal{M}_R$). From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. Longer signatures lead to higher similarity for both models.

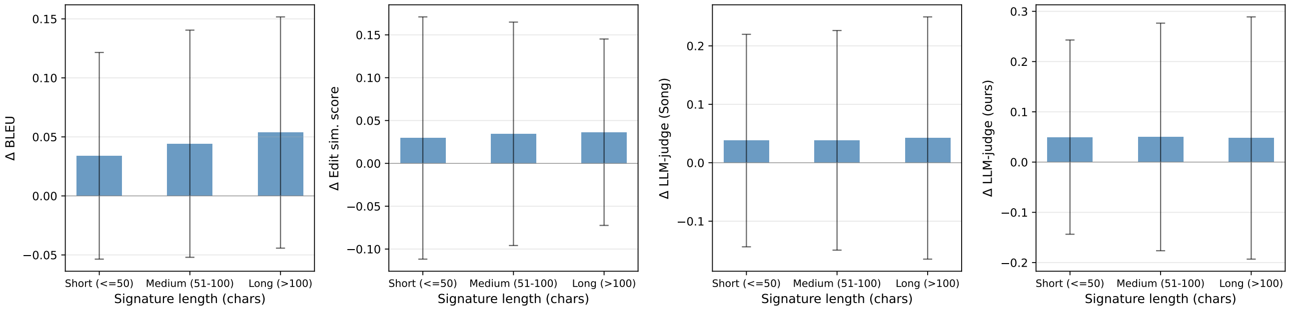


Figure 10. Mean delta Δ (target − reference) by function signature length. From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. The memorization signal is stable across signature lengths ($|r| < 0.06$).

Context length. We group functions by the character length of the preceding file context: Minimal (<100 chars; N=103), Short (100–2k; N=1,497), Medium (2k–5k; N=2,854), and Long (>5k; N=2,968). Figures 11 and 12 show the mean absolute similarity, and the mean delta, respectively, grouped by character length of the preceding context. The most notable finding is in the delta (Figure 12): the memorization signal is strongest when context is minimal or short. Functions with minimal context show a BLEU delta of $+0.14$ and LLM-judge (ours) delta of $+0.10$, compared to $+0.03$ for both with long context. The overall correlation between context length and Δ is negative ($r = -0.17$ for BLEU, $r = -0.08$ for LLM-judge). We hypothesize that that when p provides little surrounding context, $\mathcal{M}_T$'s memorization of the specific function from midtraining becomes more distinguishable from $\mathcal{M}_R$'s generation. With richer context, both models can draw on file-level patterns to produce reasonable completions, narrowing the counterfactual gap.

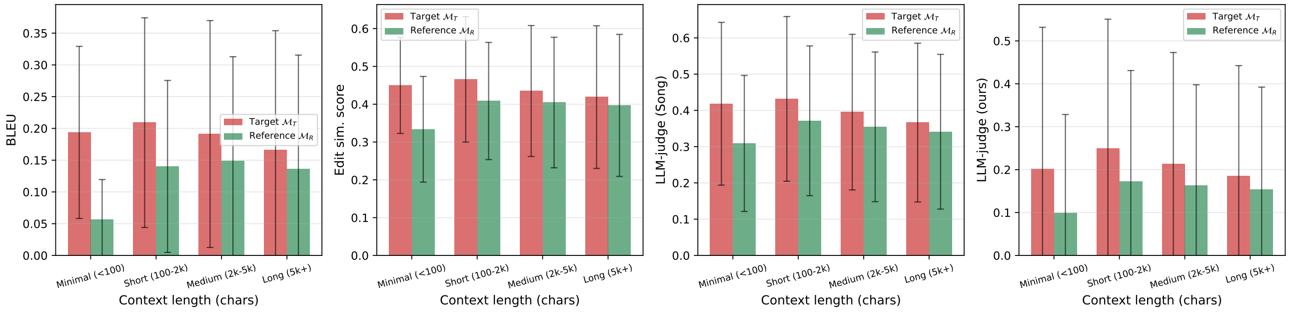


Figure 11. Mean similarity to the ground truth by preceding context length for the target model ($\mathcal{M}_T$) and reference model ($\mathcal{M}_R$). From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation.

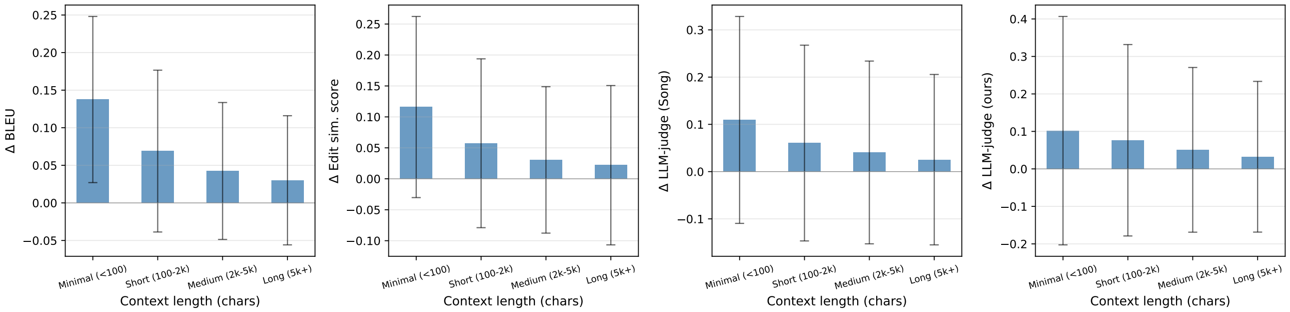


Figure 12. Mean delta Δ (target $-$ reference) by preceding context length. From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. The memorization signal is strongest with minimal context ($r = -0.17$ for BLEU, $r = -0.08$ for LLM-judge).

H.2. Continuation length

We now also group the ground-truth continuations x by function body length (in lines): Short (10–15 lines; N=3,082), Medium (16–25; N=2,557), and Long (26–50; N=1,783). Figure 13 shows that absolute $\text{SIM}(x, x^\star)$ decreases with $|x|$ for both models: target-model BLEU drops from 0.24 (short) to 0.11 (long), with Pearson $r = -0.31$. This is expected, longer functions have more tokens to reproduce, making high textual overlap less likely, consistent with prior work (Kandpal et al., 2022).

More importantly, Figure 14 shows the memorization delta Δ. For text-based metrics, the delta decreases with length: BLEU Δ drops from $+0.053$ (short) to $+0.030$ (long; $r = -0.10$). However, the LLM-judge deltas remain more stable: LLM-judge (ours) Δ is $+0.054$, $+0.050$, and $+0.041$ for short, medium, and long functions ($r = -0.02$). This suggests that while longer continuations are harder to reproduce *textually*, the model's ability to internalize their functional *logic* during midtraining—as assessed by LLM judges—does not diminish as sharply with function length.

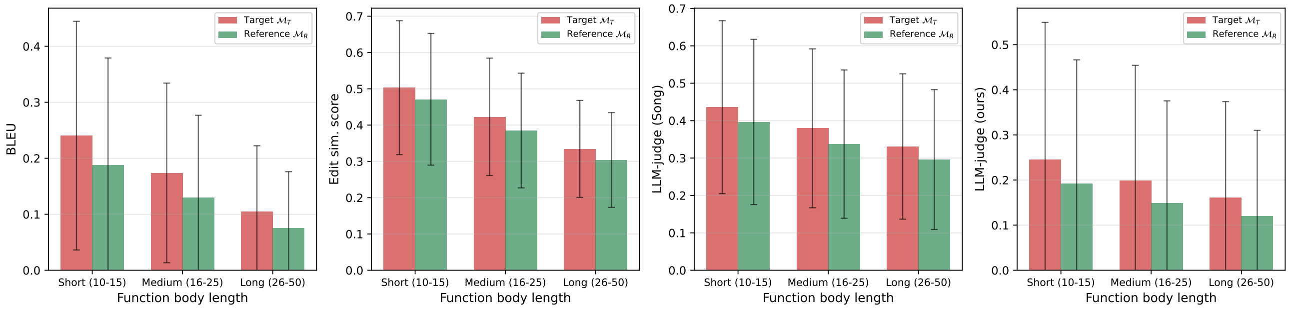


Figure 13. Mean similarity to the ground truth by function body length (continuation $|x|$) for the target model ($\mathcal{M}_T$) and reference model ($\mathcal{M}_R$). From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation.

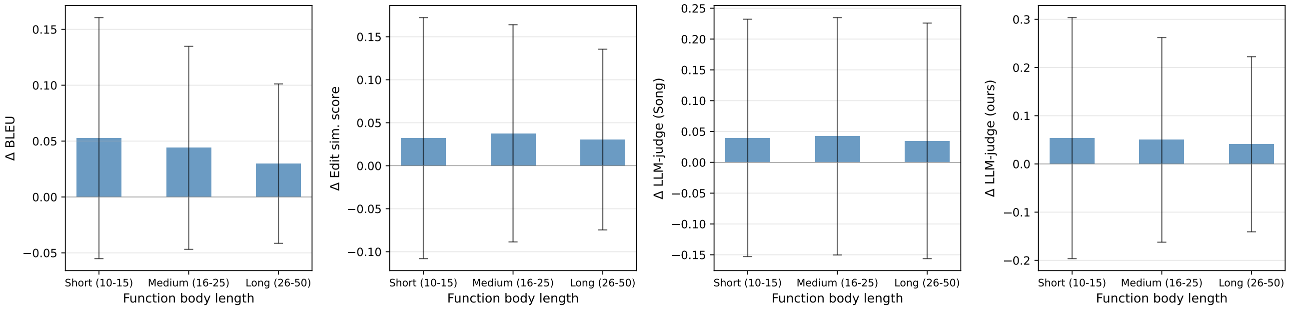


Figure 14. Mean delta Δ (target $-$ reference) by function body length (continuation $|x|$). From left to right: BLEU, edit similarity score, LLM-as-a-judge score (Song) and LLM-as-a-judge score (ours). Error bars show ±1 standard deviation. Text-based deltas decrease with length, while LLM-judge deltas remain more stable.