

# AegisCodeGen: Toward Copyright-Aware Code Generation with Runnability-Guided Fusion

Rui Chu<sup>1</sup>, Nicole Meng<sup>1</sup>, Tanusree Das Tithy<sup>2</sup>,  
Thomas Cao<sup>3</sup>, Faysal Hossain Shezan<sup>2</sup>, Yingjie Lao<sup>1</sup>

<sup>1</sup>Electrical and Computer Engineering Department, Tufts University, Medford, MA, USA

<sup>2</sup>Computer Science Department, The University of Texas at Arlington,  
Arlington, Texas, USA

<sup>3</sup>Fletcher School of Law and Diplomacy, Tufts University, Medford, MA, USA  
rchu02@tufts.edu

## Abstract

Code large language model introduces copyright risks when models reproduce protected implementations from training data. Existing inference-time copyright-protection methods overlook the power of using the original LLM capability for a runnable code generation. In this paper, we propose AEGISCODEGEN (ACG), an inference-time copyright-aware controlled generation framework for code LLMs, effectively extending the text copyright protection towards the code generation domain. We argue that code copyright protection should be formulated as a region-aware generation control problem, where the model first identifies where modification is safe and then decides how strongly copyright-aware fusion should be applied. Based on this insight, ACG mainly consists of two stages. Firstly, ACG performs multiple sampling generations to construct an agreement profile over the target program, indicating which regions are more likely to affect core code logic after modification. Secondly, ACG uses this profile to adaptively control fusion strength, applying stronger copyright protection in low-agreement regions and weaker shifts near executable anchors. Through intense experiments, we demonstrate that ACG reduces protected-content reproduction while preserving runability and generation utility, providing a lightweight copyright-aware module that requires neither model retraining nor access to training data.

## 1 Introduction

Code large language models (code LLMs) (Luo et al., 2024; Liu et al., 2023; Muennighoff et al., 2024) have enabled increasingly strong program generation capability, ranging from function-level synthesis to project-level code completion (Guo et al., 2023; Ding et al., 2023). Such progress has also shown that the performance of code generation is not only determined by model scale, but also

by how the generation process searches over possible solutions (Wang et al., 2025a). Meanwhile, *copyright risk* in language-model generation has received increasing attention (Xu et al., 2026; Peng et al., 2023; Dou et al., 2025).

Existing text-oriented copyright enhancements have been studied from multiple perspectives. One representative direction is fusion-based generation, where the decoder combines the output distributions of multiple LLMs rather than directly changing model parameters, providing a logit-level fusion so that the generation is steered away from protected content while keeping the base generation distribution robust (Abad et al., 2025). These works suggest that copyright protection can be solved easily during inference time, without modifying models and preparing data.

However, *copyright risk* in code generation is an important but under-explored domain. As policy, licensing, and responsible-deployment requirements become more intense (Friedmann, 2024; Chiba-Okabe and Su, 2025; Gaffar and Albarashdi, 2025), LLMs are expected not only to generate executable programs, but also to avoid reproducing protected implementations. In particular, verbatim copying from training data can introduce severe plagiarism and license-compliance risk (Yu et al., 2023; Xu et al., 2025; Lee et al., 2024). Existing code-copyright studies mainly focus on whether generated code infringes existing programs, or on provenance-oriented watermarking for code generation (Yu et al., 2023; Lee et al., 2024). In contrast, works that directly prevent copyright-sensitive code generation remain limited, especially during the inference stage (freezing the backbone model, no access to training data). A preliminary line of work studies duplicate-aware controlled code generation and uses targeted reordering during beam search to reduce direct copying (Fu et al., 2026). Nevertheless, directly perturbing, reordering, or rewriting generated content can hurt coherence, especially

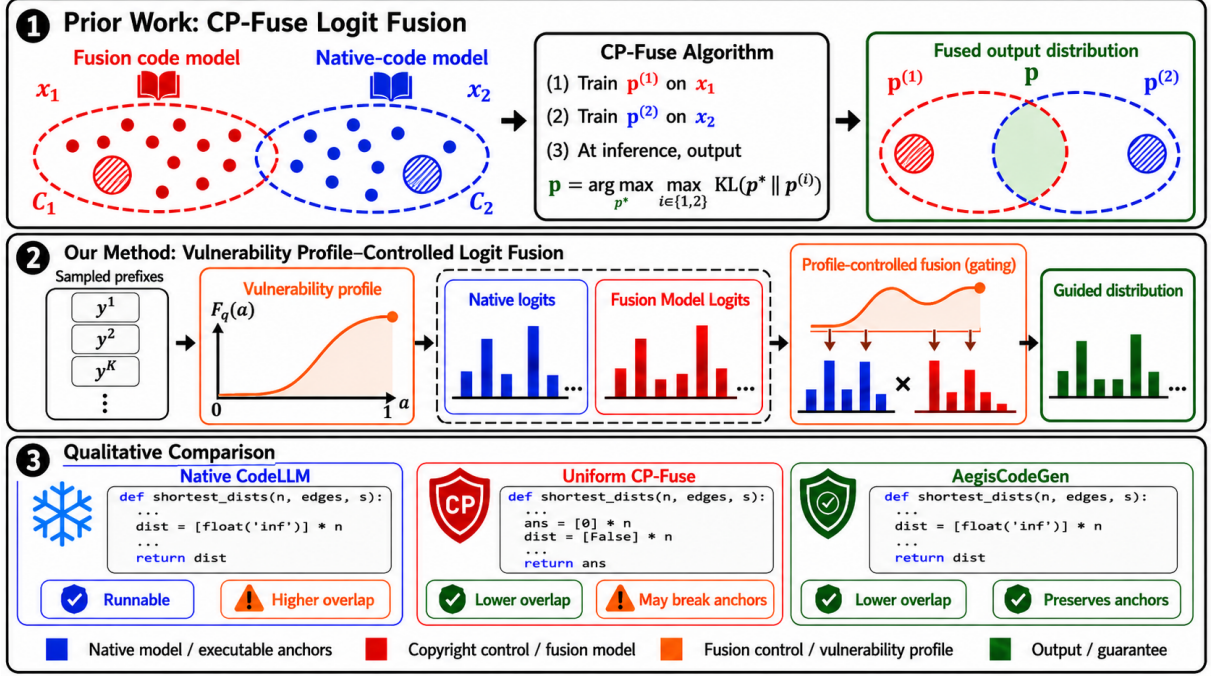


Figure 1: Top: Illustration of prior CP-Fuse logit fusion, which directly fuses the distributions of a fusion code model and a native-code model during inference. Middle: Illustration of our agreement-profile-controlled logit fusion. AegisCodeGen first constructs a query-specific agreement profile from sampled prefixes, and then uses this profile to gate the CP-Fuse output logits before producing the guided output distribution. In this way, copyright-aware fusion is strengthened in safer regions and weakened near executable anchors. Bottom: Qualitative comparison on a representative code-generation prompt. Compared with native CodeLLM decoding and uniform CP-Fuse, AegisCodeGen reduces protected-code overlap while preserving runnable code.

when the protected code target grows from an isolated function to a larger project. More importantly, such methods have ignored the power of the development of code LLMs, which can generate strong executable code.

Thus, we observe that copyright-aware code generation is facing the following challenges: 1) the generated code should remain executable and functionally correct; 2) Identifying where could contain copyright-sensitive content; and 3) the method should directly exploit the strong code-generation ability of the existing LLM while still enforcing copyright protection.

These observations motivate the central research question we seek to address: *can we transfer effective text-copyright enhancement methods (for example fusion-based generation) to the code generation setting while preserving the original LLM’s coding ability?* To approach this, the key goal is to 1) preserve functional correctness and executability (Utility) as much as possible, and 2) reduce the risk of copyright-protected content reproduction. We hypothesize that copyright control should not be applied uniformly to all generated tokens, instead, the

generation process should first identify where modification is safe and where the code contains structural anchors that should be preserved. Our preliminary analysis suggests that high-agreement regions tend to correspond to structurally constrained code, whereas low-agreement regions exhibit greater implementation diversity and are therefore more suitable for copyright-aware intervention.

In this work, we propose AegisCodeGen (ACG), a copyright-aware controlled generation framework for code generation. Our key insight is that solving this problem requires answering two questions: 1) how to find the most editable regions (like the way of adding perturbation), and 2) how to apply copyright enhancement in these regions with minimum impact to executability. ACG treats copyright-aware code generation as a targeted inference-time control problem. It first estimates an agreement profile over the generated program given query, and then uses this profile to guide where and how strongly copyright-aware fusion should be applied. ACG mainly consists of two stages. In the 1st stage, ACG performs multiple plan-search generations, where each plan can be regarded as a high-level

pseudo-code sketch of the target program, inspired by (Wang et al., 2025a), thus constructing a region-level agreement distribution as *Profile*, which indicates where are likely to affect core logic after modification. In the 2nd stage, ACG use the fusion principle (Abad et al., 2025) and adaptively assigns the fusion strength according to the *Profile*. Thus ACG applies stronger copyright protection in low-agreement regions and weaker logit shifts near executable anchors.

Our contributions are summarized as follows:

- To the best of our knowledge, ACG is among the first frameworks to study copyright-aware controlled generation for code LLMs under an inference-time scenario.
- We identify a key gap between text copyright protection and code copyright protection: for code generation, *where* to modify is as important as *how* to modify, and thus designed an agreement-profile-guided copyright control code generation.
- ACG is a lightweight module that can be attached to existing code generation pipelines without retraining the base LLM.
- Through intense experiments, we demonstrate that ACG reduces protected-content reproduction while preserving utility.

## 2 Related Work

**Code Generation Enhancements** One line of work improves code generation through execution feedback and iterative refinement. Chen et al. (2023) selects generated programs with generated tests, Ni et al. (2023) learns an execution-aware verifier, Shinn et al. (2023) repairs failed attempts with verbal feedback, and Zhang et al. (2025a) turns self-generated tests into preference optimization. Another line of work moves from local decoding to plans and repository context. Le et al. (2024) improves modular code through self-revision, Wang et al. (2025a) searches over natural-language plans, Islam et al. (2024) separates recall, planning, coding, and debugging, and Shrivastava et al. (2023); Zhang et al. (2023); Cheng et al. (2024); Wang et al. (2025b); Zhang et al. (2025b) show that repository and retrieval signals improve code completion. These methods show that code LLMs can be strengthened by tests, plans, and context; ACG instead uses these signals to decide where copyright control can be safely applied.

**Code Copyright Enhancement** One line of work first makes code copyright risk measurable. Yu et al. (2023) measures whether code LLMs reproduce licensed programs, and Xu et al. (2025) evaluates license compliance with striking similarity and generated license information. Another line of work studies provenance or structure-aware protection. Lee et al. (2024) adapts watermarking to low-entropy code, Jiang et al. (2021); Zhang et al. (2025c) show that AST-level structure can preserve coherent program units, and Fu et al. (2026) reduces direct copying with targeted beam reordering. These works move from risk assessment to provenance and decoding control; ACG differs by profiling editable regions and applying copyright-aware fusion only where it is less likely to break executability.

**LLM Generation Copyright** One line of work studies copyright risk through memorization. Carlini et al. (2021) shows that language models can emit memorized training examples, Carlini et al. (2023) quantifies how memorization scales with model and data factors, and Lee et al. (2022) reduces memorization through data deduplication. Another line of work formalizes or detects copyright protection. Karamolegkou et al. (2023) studies copyrighted redistribution as verbatim memorization, Meeus et al. (2024) uses copyright traps to detect training-data use, Vyas et al. (2023) gives a provable protection notion, and Abad et al. (2025) implements adaptive model fusion at inference time. These works establish the path from memorization measurement to distribution-level copyright control; ACG transfers this path to code generation with executability-aware, region-level control.

## 3 Preliminaries

**Copyright-Protecting Model Fusion** CP-Fuse (Abad et al., 2025) is an inference-time logit fusion method for reducing copyrighted content reproduction. It fuses the next-token distributions of multiple models during autoregressive decoding.

At decoding step  $t$ , all models are conditioned on the same prefix  $y_{<t}$ . Each model outputs a next-token logit vector over the same vocabulary:  $z_t^{(i)} \in \mathbb{R}^{|V|}$ . CP-Fuse then combines these logits into a fused logit vector:

$$z_t^c = \text{CPFuse}(z_t^{(1)}, z_t^{(2)}, \dots, z_t^{(K)}).$$

The next token is sampled from the fused distribution  $y_t \sim \text{softmax}(z_t^c)$  and the selected token is appended to the shared prefix. Therefore, all models remain synchronized on the same generated sequence. If a copyrighted continuation is strongly preferred by only one model, but not by the others, CP-Fuse weakens this one-sided preference through adaptive fusion. In this way, the generated sequence is less likely to follow a memorized protected path.

However, CP-Fuse applies fusion throughout decoding and does not explicitly control how much the fused distribution may deviate from the native CodeLLM distribution. For code generation, such deviation may affect code runability. This motivates our method: instead of using CP-Fuse with a fixed strength, we pre-compute a query-specific profile that controls how strongly CP-Fuse should guide each decoding step.

ACG take CP-Fuse as a base copyright-enhance method, but the idea can be extended to turn more text-based copyright-aware generation into code-based copyright generation protection.

## 4 Methodology

### 4.1 Problem Statement

We study inference-time copyright-aware code generation with a frozen CodeLLM under CP-Fuse architecture.

Our goal is to find a query-specific guidance profile that controls how strongly Fusion should happen at each decoding step  $t$ . The profile should preserve the native executable-code prior in high-agreement regions, while allowing stronger Fusion guidance in low-agreement regions. We call the Profile  $F$ .

### 4.2 Method Overview

High-level workflow can be found in Fig. 1 Let  $z_t^0$  denote the native CodeLLM logits and  $z_t^c$  denote the CP-Fuse logits obtained by fusing the native CodeLLM and the fusion CodeLLM on the same prefix  $y_{<t}$ . Here,  $z_t^c$  is the fused output of CP-Fuse rather than the logits of the fusion CodeLLM alone. We propose a two-stage framework. First, we pre-run multiple code trajectories (inspired by *PlanSearch* (Wang et al., 2025a)) for the same query and build an agreement profile  $\hat{F}_q$  from the logit agreement between  $z_t^0$  and  $z_t^c$ .

Second, during final decoding,  $\hat{F}_q$  maps the current agreement score to a continuous fusion

---

### Algorithm 1 Query-Specific Agreement Profiling

---

**Require:** query  $q$ ; CodeLLM; Fusion Model

```

1:  $\mathcal{Y} \leftarrow \text{SampleCodeEvidence}(q)$ 
2:  $A_q \leftarrow \emptyset$ 
3: for each  $y \in \mathcal{Y}$  do
4:   for each prefix  $y_{<t}$  in  $y$  do
5:      $z_t^0 \leftarrow \text{NativeLogits}(q, y_{<t})$ 
6:      $z_t^c \leftarrow \text{CPFuseLogits}(q, y_{<t})$ 
7:      $p_t^0 \leftarrow \text{softmax}(z_t^0)$ ,
8:      $p_t^c \leftarrow \text{softmax}(z_t^c)$ 
9:      $a_t \leftarrow 1 - D_{\text{JS}}(p_t^c, p_t^0) / \log 2$ 
10:     $A_q \leftarrow A_q \cup \{a_t\}$ 
11:   end for
12: end for
13:  $\hat{F}_q \leftarrow \text{EmpiricalCDF}(A_q)$ 
14: return  $\hat{F}_q$ 

```

---

strength  $s_t \in [0, 1]$ , and the output Fused logits are computed by

$$z_t^{\text{out}} = z_t^0 + s_t(z_t^c - z_t^0).$$

### 4.3 Query-Specific Agreement Profiling

Algorithm 1 builds the query-specific agreement profile. Line 1 samples multiple candidate code trajectories for the same query (inspired by (Wang et al., 2025a)). Lines 3–10 replay each sampled trajectory prefix by prefix, so native logits and CP-Fuse logits are always computed on the same prefix. (Therefore, the method does not require aligning token positions across different samples.)

Lines 5–8 compute a normalized agreement score: (the normalized score  $a_t = 1 - D_{\text{JS}}(p_t^c, p_t^0) / \log 2$  lies in  $[0, 1]$  because of Jensen-Shannon divergence satisfies  $0 \leq D_{\text{JS}} \leq \log 2$ ) larger values mean that the CP-Fuse output distribution is closer to the native CodeLLM distribution. Lines 10 and 13 aggregate all agreement scores into the empirical distribution profile  $\hat{F}_q$  (by CDF), which calibrates what high or low agreement means for the current query.

### 4.4 Profile-Guided Fusion Decoding

Algorithm 2 applies the agreement profile during final decoding. Lines 3–6 recompute native logits, CP-Fuse logits, and their probability distributions on the live prefix. This is important because the final decoding trajectory may differ from the sampled trajectories used in Algorithm 1. Thus, the profile is not a fixed token-position mask; it is a query-specific calibration function applied online.

---

**Algorithm 2** Profile-Guided Fusion Decoding

---

**Require:** query  $q$ ; agreement profile  $\hat{F}_q$ ; native CodeLLM; Fusion Model

```
1:  $y_{<1} \leftarrow \emptyset$ 
2: for decoding step  $t = 1, 2, \dots$  do
3:    $z_t^0 \leftarrow \text{NativeLogits}(q, y_{<t})$ 
4:    $z_t^c \leftarrow \text{CPFuseLogits}(q, y_{<t})$ 
5:    $p_t^0 \leftarrow \text{softmax}(z_t^0)$ ,  $p_t^c \leftarrow \text{softmax}(z_t^c)$ 
6:    $a_t \leftarrow 1 - D_{\text{JS}}(p_t^c, p_t^0) / \log 2$ 
7:    $s_t \leftarrow (1 - a_t)(1 - \hat{F}_q(a_t))$ 
8:    $z_t^{\text{out}} \leftarrow z_t^0 + s_t(z_t^c - z_t^0)$ 
9:    $y_t \leftarrow \text{Decode}(z_t^{\text{out}})$ 
10:   $y_{<t+1} \leftarrow y_{<t} \circ y_t$ 
11:  if  $y_t$  is EOS then
12:    break
13:  end if
14: end for
15: return  $y$ 
```

---

Line 6 computes the current agreement score, and line 7 converts it into the fusion strength  $s_t$ . The product  $(1 - a_t)(1 - \hat{F}_q(a_t))$  combines absolute and query-relative disagreement, assigning weaker fusion to high-agreement regions and stronger fusion to low-agreement regions. Line 8 then interpolates between native logits and CP-Fuse logits. Lines 9–14 continue standard autoregressive decoding  $y$ .

## 5 Experiments

All Experiments are conducted on *Nvidia L40s GPUs*. Detailed configuration has been put into the code file.

### 5.1 Experiment Setup

In all ACG experiments, we instantiate CP-Fuse with two CodeLLMs: the native CodeLLM and one fusion-partner CodeLLM, where the native CodeLLM is one of the two CP-Fuse inputs.

**Baseline Methods.** We compare ACG with three groups of copyright-aware code generation baselines. For copyright-infringement evaluation, we include Overfit Split 1/2, SystemPrompt, Mem-Free, CP- $\Delta$ , and CP-Fuse, covering overfitted reference models, prompt-only control, memorization-suppression decoding, token-wise distributional control, and adaptive model fusion (Abad et al., 2025; Vyas et al., 2023). For code-IP mitigation, we include the Baseline Model, public-domain

Fine-Tune, and Dynamic Filter under the Code-Parrot and CodeRL settings (Yu et al., 2023). For duplicate-aware decoding, we include Standard Beam Search, Output Regeneration, Training Data Rewrite, TRBS w/o SAP, and full TRBS on HumanEval-X (Fu et al., 2026). For function-level executability, we evaluate pass@1 on APPS, MBPP, and HumanEval (Hendrycks et al., 2021; Austin et al., 2021; Chen et al., 2021).

**Metrics.** For the CP-Fuse copyright benchmark, EM denotes exact-matching length, BLEU measures  $n$ -gram overlap, LEV denotes normalized Levenshtein distance, and JP denotes the JPlag plagiarism score; lower EM, BLEU, and JP and higher LEV indicate lower reproduction risk (Abad et al., 2025; Prechelt et al., 2002; Papineni et al., 2002). For CodeIPPrompt, EM denotes expected maximum similarity, EP denotes the empirical probability of producing at least one high-similarity candidate, and Pass@ $k$  measures code synthesis success over  $k$  generated candidates (Yu et al., 2023). For TRBS, LCS denotes the longest common subsequence length between the generated code and the protected reference, while ACC denotes execution accuracy on HumanEval-X (Fu et al., 2026). For executability, APPS, MBPP, and HumanEval are evaluated by pass@1. We emphasize that EM is overloaded across tables: it means exact-matching length in CP-Fuse but expected maximum similarity in CodeIPPrompt.

### 5.2 Main Results

#### 5.2.1 Reducing Exact Reproduction.

Table 1 shows that ACG consistently achieves lower copyright-infringement risk than CP-Fuse on both splits. In particular, the reduced exact-match length and low plagiarism score indicate that ACG is effective at suppressing long-span verbatim reproduction. Concretely, compared with CP-Fuse, ACG reduces the 95th-percentile EM from 69.58 to 55.37 on Split 1 and from 68.04 to 53.91 on Split 2, while also lowering JP from 0.03 to 0.02 on both splits. These paired improvements suggest that region-aware fusion provides stronger copyright protection than applying fusion uniformly during decoding.

Table 1 shows that ACG further suppresses high-risk exact reproduction while keeping plagiarism signals low. Notably, the absolute EM gap of roughly 14 tokens at the 95th percentile (69.58→55.37 on Split 1; 68.04→53.91 on Split 2)

Table 1: Copyright-infringement metrics averaged at the 95th percentile. EM denotes exact matching length; JP denotes JPlag. Lower is better for EM and JP.

| Method          | Split   | EM↓     | JP↓  |
|-----------------|---------|---------|------|
| Overfit Split 1 | Split 1 | 1469.80 | 1.00 |
|                 | Split 2 | 44.60   | 0.01 |
| Overfit Split 2 | Split 1 | 42.64   | 0.01 |
|                 | Split 2 | 1393.88 | 0.99 |
| SystemPrompt    | Split 1 | 1360.68 | 1.00 |
|                 | Split 2 | 1373.16 | 0.99 |
| MemFree         | Split 1 | 165.48  | 0.99 |
|                 | Split 2 | 157.36  | 0.96 |
| CP- $\Delta$    | Split 1 | 273.20  | 1.00 |
|                 | Split 2 | 284.80  | 0.99 |
| CP-Fuse         | Split 1 | 69.58   | 0.03 |
|                 | Split 2 | 68.04   | 0.03 |
| <b>Ours</b>     | Split 1 | 55.37   | 0.02 |
|                 | Split 2 | 53.91   | 0.02 |

is large enough to move many cases out of the “long-span copy” regime, even though both methods already keep JP close to zero. Equivalently, if we view the 95th percentile as a “worst-case” slice, then ACG removes about 1 out of every 5 tokens from these long exact-match segments compared with CP-Fuse, which directly lowers the chance that a generated function contains a contiguous, license-relevant copied block.

We also note that the reduction is symmetric across splits (20.4% vs. 20.8%), whereas the overfitted reference models swap which split is catastrophic (e.g., 1469.80 vs. 44.60 for Overfit Split 1). This indicates that the region-aware controller is not relying on a split-specific quirk, but rather provides a stable mitigation effect when the test set changes. The gain over CP-Fuse supports the central claim that copyright control should be applied according to editable regions rather than uniformly over the whole decoding path.

**Tail behavior of exact reproduction.** Figure 2 complements the aggregate copyright metrics in Table 1. Compared with CP- $\Delta$ , both CP-Fuse and ACG shift the distribution toward shorter exact matches and substantially reduce the long-tail behavior. ACG is visually better than, CP-Fuse in the high-risk region, suggesting that region-aware copyright control can further suppress long-span

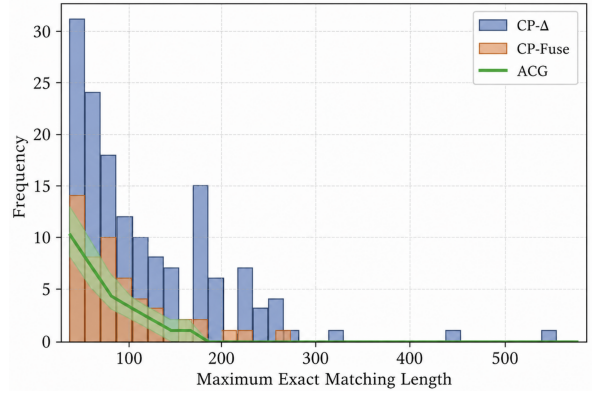


Figure 2: Histogram of exact-matched substring lengths on the Python Instructions setting. We compare the tail behavior of CP- $\Delta$ , CP-Fuse, and ACG. A better method should place more mass on shorter matches and suppress the long-match tail, indicating lower verbatim code reproduction risk.

verbatim code reproduction. This is consistent with the 95th-percentile EM reduction in Table 1, where ACG removes about one-fifth of the longest exact matches relative to CP-Fuse. This result is consistent with our goal: reduce dangerous copying while avoiding unnecessarily strong intervention over the entire generation, i.e., applying stronger control in regions identified as more editable by the profile.

### 5.2.2 Safety–Utility Trade-off.

Table 2 shows that ACG achieves stronger IP-risk mitigation without the severe pass-rate drop caused by aggressive filtering. Numerically, under the CodeParrot setting, ACG reduces EM from 0.31 to 0.22 when moving from the Baseline-Model column to the Dynamic-Filter column (and similarly keeps EP low at 0.018→0.011), while pass@1 only changes from 3.73% to 3.49% (a relative drop of 6.4%). The same stability holds for broader sampling: pass@10 drops from 6.34% to 5.72% and pass@100 drops from 9.27% to 8.66%, indicating that the mitigation does not disproportionately harm diversity-based success.

In contrast, dynamic filtering on the original CodeParrot backbones drives EP to 0.00 but also reduces pass@1 substantially (e.g., 3.84%→1.46% for CodeParrot-1.5B, a 62.0% relative drop). The pattern is similar for CodeParrot-110M (3.41%→1.17%), which highlights the typical failure mode of hard filtering: it eliminates risky candidates by pruning a large portion of the model’s high-probability space, thereby losing solvable trajectories.

Compared with fine-tuning, ACG achieves com-

parable or lower EM (e.g., 0.22 vs. 0.67 for CodeParrot-1.5B fine-tune, and 0.22 vs. 0.40 for CodeParrot-1.5B dynamic filter) while keeping utility close to the baseline. Under CodeRL, ACG maintains the same pattern: EM decreases from 0.27 to 0.22 with a modest pass@1 change from 3.68% to 3.49%. Overall, these numbers support the claim that adaptive fusion can reduce risky similarity while preserving the native code-generation prior, rather than relying on a binary accept/reject mechanism.

### 5.2.3 Structure-Aware Protection.

Table 3 shows that ACG reduces protected-reference overlap while maintaining strong execution accuracy. For example, relative to Standard Beam Search, ACG substantially lowers LCS on Python (65.68→34.76, a 47.1% reduction) while keeping ACC nearly unchanged (80.49→79.27). Similar trends hold across languages: LCS drops on JavaScript from 37.74 to 21.87 with ACC remaining competitive (76.83→72.38), indicating that overlap reduction does not require collapsing to low-quality rewrites.

Beyond these two languages, we observe consistent overlap suppression on Java (103.27→52.63, a 49.0% reduction), Go (55.44→27.92, a 49.6% reduction), and C++ (60.81→31.68, a 47.9% reduction). Meanwhile, ACC remains within a moderate range (74.71/62.84/59.37 respectively), which is notably higher than methods that over-correct by rewriting too aggressively. For instance, Training Data Rewrite indeed reduces LCS substantially (e.g., Python 50.57 and Java 57.29), but this comes with severe accuracy collapse in some languages (e.g., Java ACC 21.95 and C++ ACC 32.32), suggesting that indiscriminate rewriting can break executable anchors.

Compared with TRBS variants, ACG achieves comparable or lower LCS in several languages (e.g., JavaScript 21.87 vs. 24.24–27.53 for TRBS w/o SAP), while avoiding the larger ACC degradation seen at stronger TRBS w/o SAP settings (e.g., Python ACC 79.27 vs. 60.98 at TRBS w/o SAP  $\alpha = 5$ ). This gap suggests that our controller is not simply increasing a global penalty coefficient; instead, it selectively edits the portions most responsible for protected overlap while keeping the execution-critical anchors stable.

Overall, the cross-language pattern suggests that the profile-guided controller protects executable anchors and applies stronger modification only where

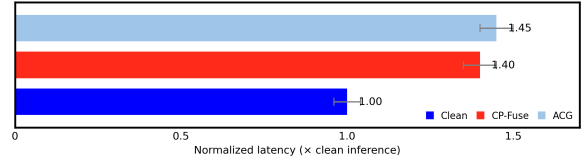


Figure 3: Time complexity comparison. Runtime is normalized by clean CodeLLM inference.

it is safer.

### 5.3 Time Complexity

Figure 3 shows that the runtime overhead of ACG is modest. Compared with clean inference, CP-Fuse introduces the main additional cost from fusion. ACG only adds query-specific profile computing on top of CP-Fuse, increasing normalized latency from 1.40× to 1.45×. Thus, the cost of our agreement-profile stage is small relative to the decoding and fusion cost, while the full pipeline remains well below 2× clean inference. This supports our claim that ACG is a lightweight inference-time copyright-control module rather than a post-training or retraining-based method.

### 5.4 Executability and Utility

**Preserved Code Utility.** Table 4 shows that ACG preserves strong task utility across standard code-generation benchmarks. This confirms that the proposed control does not over-regularize generation, but instead keeps the base CodeLLM capability effective during copyright-aware decoding.

We further provide a preliminary analysis of the relationship between agreement and executability sensitivity in Appendix B.5.

## 6 Conclusion

In this paper, we introduced ACG, an inference-time framework for copyright-aware code generation. Unlike prior copyright-protection methods that apply generation control uniformly, ACG formulates code copyright protection as a region-aware control problem. The key insight is that copyright-aware decoding for code should not only decide how to modify the generation, but also identify where such modification can be safely applied without disrupting executable program structure.

ACG addresses this challenge through a two-stage design. It first constructs a query-specific agreement profile from multiple sampled code trajectories, estimating which regions are more sensitive to functional correctness. It then uses this

Table 2: Performance of mitigation strategies compared to baseline models under the CodeParrot and CodeRL frameworks.

| Baseline Model      |                      |       |          |          |          |
|---------------------|----------------------|-------|----------|----------|----------|
| Model               | EM                   | EP    | Pass@1   | Pass@10  | Pass@100 |
| CodeParrot-110M     | 0.98 <sub>0.04</sub> | 0.99  | 3.41%    | 5.39%    | 7.02%    |
| CodeParrot-1.5B     | 0.99 <sub>0.01</sub> | 0.99  | 3.84%    | 6.77%    | 10.02%   |
| CodeT5-large        | 0.11 <sub>0.21</sub> | 0.04  | 0.00     | 0.00     | 0.00     |
| CodeT5-large-ntp-py | 0.92 <sub>0.12</sub> | 0.99  | 1.40e-05 | 1.30e-04 | 8.00e-04 |
| ACG                 | 0.31 <sub>0.07</sub> | 0.018 | 3.73%    | 6.34%    | 9.27%    |

| Fine-Tune           |                      |       |          |          |          |
|---------------------|----------------------|-------|----------|----------|----------|
| Model               | EM                   | EP    | Pass@1   | Pass@10  | Pass@100 |
| CodeParrot-110M     | 0.71 <sub>0.13</sub> | 0.70  | 3.56%    | 5.17%    | 7.44%    |
| CodeParrot-1.5B     | 0.67 <sub>0.18</sub> | 0.69  | 3.14%    | 5.84%    | 8.28%    |
| CodeT5-large        | 0.71 <sub>0.15</sub> | 0.68  | 1.70e-03 | 7.78e-03 | 0.02     |
| CodeT5-large-ntp-py | 0.82 <sub>0.10</sub> | 0.87  | 2.48e-03 | 8.64e-03 | 0.02     |
| ACG                 | 0.27 <sub>0.06</sub> | 0.014 | 3.68%    | 6.18%    | 9.03%    |

| Dynamic Filter      |                      |       |        |          |          |
|---------------------|----------------------|-------|--------|----------|----------|
| Model               | EM                   | EP    | Pass@1 | Pass@10  | Pass@100 |
| CodeParrot-110M     | 0.46 <sub>0.08</sub> | 0.00  | 1.17%  | 1.69%    | 3.28%    |
| CodeParrot-1.5B     | 0.40 <sub>0.10</sub> | 0.00  | 1.46%  | 2.11%    | 2.78%    |
| CodeT5-large        | 0.10 <sub>0.20</sub> | 0.00  | 0.00   | 0.00     | 0.00     |
| CodeT5-large-ntp-py | 0.47 <sub>0.06</sub> | 0.00  | 0.00   | 2.83e-05 | 1.42e-05 |
| ACG                 | 0.22 <sub>0.04</sub> | 0.011 | 3.49%  | 5.72%    | 8.66%    |

profile to adaptively control the fusion strength between the native CodeLLM logits and copyright-protective CP-Fuse logits. As a result, stronger copyright protection is applied in low-agreement regions, while weaker intervention is used near executable anchors. Extensive experiments demonstrate the effectiveness of ACG across multiple copyright-aware code-generation settings. The results show that ACG reduces protected-content reproduction, suppresses long-span verbatim copying, while preserving runnability and general code-generation utility.

These findings validate our central claim that copyright-aware code generation requires region-aware, executability-preserving control rather than uniform decoding perturbation. Overall, ACG provides a lightweight and practical copyright-aware module, and provides one of the first insights into inference-time copyright-aware code generation.

## Limitations

**No legal guarantee.** Our evaluation metrics measure memorization and protected-content reproduction as engineering proxies; they do not directly establish copyright infringement or license.

**Scope of protection.** Our control primarily targets verbatim or highly overlapping generations and may be less effective when lexical overlap is low but structural similarity remains high.

**Repository-level deployment.** Our evaluation is primarily function-level; repository-scale integration involving APIs, dependencies, and cross-file interactions remains future work.

## Ethical Considerations

ACG is intended as a technical mitigation for reducing reproduction of protected code. It does not determine copyright ownership, infringement, or

Table 3: Averaged comparative results for structure anchor protection on HumanEval-X. LCS denotes longest common subsequence length; ACC denotes execution accuracy. Lower is better for LCS; higher is better for ACC.

| LCS↓                                     |        |        |            |       |       |
|------------------------------------------|--------|--------|------------|-------|-------|
| Method                                   | Python | Java   | JavaScript | Go    | C++   |
| Qwen3-8B – Standard Beam Search          | 65.68  | 103.27 | 37.74      | 55.44 | 60.81 |
| Qwen3-8B – Output Regeneration           | 58.63  | 95.43  | 34.93      | 44.66 | 58.02 |
| Qwen3-8B – Training Data Rewrite         | 50.57  | 57.29  | 27.74      | 38.96 | 22.87 |
| Qwen3-8B – TRBS ( $\alpha = 2$ )         | 55.16  | 70.57  | 26.45      | 39.55 | 39.60 |
| Qwen3-8B – TRBS ( $\alpha = 3$ )         | 43.30  | 63.48  | 22.93      | 33.44 | 35.07 |
| Qwen3-8B – TRBS ( $\alpha = 4$ )         | 41.27  | 55.64  | 22.31      | 30.43 | 33.99 |
| Qwen3-8B – TRBS ( $\alpha = 5$ )         | 38.62  | 53.51  | 22.41      | 29.06 | 33.51 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 2$ ) | 53.91  | 73.70  | 27.53      | 39.76 | 39.09 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 3$ ) | 40.64  | 67.75  | 25.10      | 35.70 | 35.24 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 4$ ) | 38.61  | 60.07  | 24.33      | 33.37 | 34.23 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 5$ ) | 36.08  | 57.89  | 24.24      | 33.90 | 34.59 |
| ACG                                      | 34.76  | 52.63  | 21.87      | 27.92 | 31.68 |

| ACC↑                                     |        |       |            |       |       |
|------------------------------------------|--------|-------|------------|-------|-------|
| Method                                   | Python | Java  | JavaScript | Go    | C++   |
| Qwen3-8B – Standard Beam Search          | 80.49  | 80.49 | 76.83      | 65.85 | 69.51 |
| Qwen3-8B – Output Regeneration           | 75.00  | 63.41 | 70.12      | 54.27 | 61.59 |
| Qwen3-8B – Training Data Rewrite         | 78.66  | 21.95 | 40.98      | 32.93 | 32.32 |
| Qwen3-8B – TRBS ( $\alpha = 2$ )         | 80.49  | 64.63 | 61.59      | 51.22 | 48.78 |
| Qwen3-8B – TRBS ( $\alpha = 3$ )         | 78.66  | 51.83 | 55.49      | 48.17 | 40.85 |
| Qwen3-8B – TRBS ( $\alpha = 4$ )         | 75.61  | 42.68 | 58.54      | 43.90 | 39.63 |
| Qwen3-8B – TRBS ( $\alpha = 5$ )         | 73.78  | 37.80 | 58.54      | 39.63 | 43.90 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 2$ ) | 77.44  | 67.68 | 67.68      | 54.88 | 52.44 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 3$ ) | 69.51  | 60.98 | 67.07      | 56.10 | 52.44 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 4$ ) | 67.07  | 60.37 | 68.90      | 55.49 | 52.44 |
| Qwen3-8B – TRBS w/o SAP ( $\alpha = 5$ ) | 60.98  | 64.02 | 67.07      | 56.10 | 52.44 |
| ACG                                      | 79.27  | 74.71 | 72.38      | 62.84 | 59.37 |

license compliance and should not replace legal or

provenance review in deployment.

Table 4: Utility metrics across datasets. Pass@1 is reported for APPS, MBPP, and HumanEval. Higher is better.

| Method          | APPS | MBPP | HumanEval |
|-----------------|------|------|-----------|
| Overfit Split 1 | 0.43 | 0.44 | 0.29      |
| Overfit Split 2 | 0.42 | 0.44 | 0.28      |
| SystemPrompt    | 0.41 | 0.43 | 0.29      |
| MemFree         | 0.32 | 0.41 | 0.24      |
| CP- $\Delta$    | 0.45 | 0.46 | 0.29      |
| CP-Fuse         | 0.47 | 0.43 | 0.28      |
| ACG             | 0.48 | 0.46 | 0.31      |

ACG is designed to be used as a guardrail within a broader responsible-development workflow. In practice, developers should pair it with dataset documentation, model cards, and clear user-facing disclosures about training data uncertainty and the residual risk of memorization.

While our method aims to suppress long-span verbatim copying, it may still generate code that is substantially similar to protected content at the semantic or structural level.

## Acknowledgments

The work is supported in part by NSF 2532587, 2532588, and 2426299.

## References

- Javier Abad, Konstantin Donhauser, Francesco Pinto, and Fanny Yang. 2025. [Copyright-protected language generation via adaptive model fusion](#). In *The Thirteenth International Conference on Learning Representations*.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *arXiv preprint arXiv:2108.07732*.
- Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2023. [Quantifying memorization across neural language models](#). In *The Eleventh International Conference on Learning Representations*.
- Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. [Extracting training data from large language models](#). In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. [CodeT: Code generation with generated tests](#). In *The Eleventh International Conference on Learning Representations*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *arXiv preprint arXiv:2107.03374*.
- Wei Cheng, Yuhan Wu, and Wei Hu. 2024. [Dataflow-guided retrieval augmentation for repository-level code completion](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand. Association for Computational Linguistics.
- Hiroaki Chiba-Okabe and Weijie J Su. 2025. [Tackling copyright issues in AI image generation through originality estimation and genericization](#). *Scientific Reports*, 15(1):10621.
- Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. [A static evaluation of code completion by large language models](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*.
- Guangyao Dou, Zheyuan Liu, Qing Lyu, Kaize Ding, and Eric Wong. 2025. [Avoiding copyright infringement via large language model unlearning](#). In *Findings of the Association for Computational Linguistics: NAACL 2025*.
- Danny Friedmann. 2024. [Creation and generation copyright standards](#). *NYU J. Intell. Prop. & Ent. L.*, 14:51.
- Junbo Fu, Guoshuai Zhao, Linkang Yang, Yunqi Mi, and Xueming Qian. 2026. [Duplicate-aware controlled code generation: Enhancing copyright protection with targeted reordering beam search in LLMs](#). In *Findings of the Association for Computational Linguistics: ACL 2026*, San Diego, California, United States. Association for Computational Linguistics.
- Hafiz Gaffar and Saleh Albarashdi. 2025. [Copyright protection for AI-generated works: Exploring originality and ownership in a digital landscape](#). *Asian Journal of International Law*, 15(1):23–46.
- Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. [LongCoder: A long-range pre-trained language model for code completion](#). In *International Conference on Machine Learning*. PMLR.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with APPS](#). *arXiv preprint arXiv:2105.09938*.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. [MapCoder: Multi-agent code generation for competitive problem solving](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand. Association for Computational Linguistics.
- Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. [TreeBERT: A tree-based pre-trained model for programming language](#). In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*. PMLR.
- Antonia Karamolegkou, Jiaang Li, Li Zhou, and Anders Søgaard. 2023. [Copyright violations and large language models](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore. Association for Computational Linguistics.
- Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. [CodeChain: Towards modular code generation through chain of self-revisions with representative sub-modules](#). In *The Twelfth International Conference on Learning Representations*.
- Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. [Deduplicating training](#)

- data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Dublin, Ireland. Association for Computational Linguistics.
- Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2024. [Who wrote this code? watermarking for code generation](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. [Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation](#). *Advances in neural information processing systems*, 36.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. [WizardCoder: Empowering code large language models with Evol-Instruct](#). In *The Twelfth International Conference on Learning Representations*.
- Matthieu Meeus, Igor Shilov, Manuel Faysse, and Yves-Alexandre De Montjoye. 2024. [Copyright traps for large language models](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*. PMLR.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. [OctoPack: Instruction tuning code large language models](#). In *The Twelfth International Conference on Learning Representations*.
- Ansong Ni, Sridhar Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. [LEVER: Learning to verify language-to-code generation with execution](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*. PMLR.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. [BLEU: a method for automatic evaluation of machine translation](#). In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*.
- Wenjun Peng, Jingwei Yi, Fangzhao Wu, Shangxi Wu, Bin Bin Zhu, Lingjuan Lyu, Binxing Jiao, Tong Xu, Guangzhong Sun, and Xing Xie. 2023. [Are you copying my model? protecting the copyright of large language models for EaaS via backdoor watermark](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.
- Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. [Finding plagiarisms among a set of programs with JPlag](#). *Journal of Universal Computer Science*, 8(11):1016–1038.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. 2023. [Reflexion: language agents with verbal reinforcement learning](#). In *Advances in Neural Information Processing Systems*, volume 36.
- Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. [Repository-level prompt generation for large language models of code](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*. PMLR.
- Nikhil Vyas, Sham M. Kakade, and Boaz Barak. 2023. [On provable copyright protection for generative models](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*. PMLR.
- Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2025a. [Planning in natural language improves LLM search for code generation](#). In *The Thirteenth International Conference on Learning Representations*.
- Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025b. [CodeRAG-Bench: Can retrieval augment code generation?](#) In *Findings of the Association for Computational Linguistics: NAACL 2025*, Albuquerque, New Mexico. Association for Computational Linguistics.
- Naen Xu, Jinghui Zhang, Changjiang Li, Hengyu An, Chunyi Zhou, Jun Wang, Boyu Xu, Yuyuan Li, Tianyu Du, and Shouling Ji. 2026. [Bridging the copyright gap: Do large vision-language models recognize and respect copyrighted content?](#) In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40.
- Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. 2025. [LiCoEval: Evaluating LLMs on license compliance in code generation](#). In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE.
- Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. 2023. [CodeIPPrompt: Intellectual property infringement assessment of code language models](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*. PMLR.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [RepoCoder: Repository-level code completion through iterative retrieval and generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore. Association for Computational Linguistics.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025a. [CodeDPO: Aligning code models with self generated and verified source code](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vienna, Austria. Association for Computational Linguistics.

Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025b. [CodeRAG: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Suzhou, China. Association for Computational Linguistics.

Yilin Zhang, Xinran Zhao, Zora Zhiruo Wang, Chenyang Yang, Jiayi Wei, and Tongshuang Wu. 2025c. [cAST: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree](#). In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Suzhou, China.

## Appendix

### A AI Usage Disclosure

AI are used for polishing writing and code editing in this paper.

### B More Experiments

#### B.1 Implementation and Profile Robustness

Stage 1 uses top- $p$  0.95, and at most 128 new tokens; experiments use bf16. Profile stability improves as the sampling budget increases: the split-half KS distance decreases from 0.176 at  $N = 2$  to 0.061 at  $N = 16$ .

Unless otherwise stated, reported results are obtained from a single evaluation run.

#### B.2 Scalability

Additional experiments on Qwen3-14B and Qwen3-32B show that ACG remains effective as model scale increases, with comparable reductions in protected-code overlap while preserving execution accuracy.

#### B.3 Broader Threat Model

We focus on black-box elicitation of memorized protected code, where an adversary issues prompts and observes model outputs without access to model weights, logits, or the ACG controller. Accordingly, ACG targets verbatim or near-verbatim continuation along memorized paths; copyright risks that do not arise through this reproduction mechanism remain outside its scope.

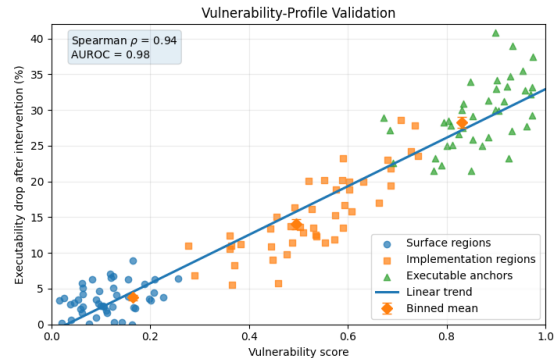


Figure 4: Agreement-profile validation. Each point denotes a code region.

### B.4 Representative Failure Cases

We highlight three representative failure modes.

**HE-01 (semantic drift):** profile-guided fusion changes a loop stopping condition, causing an off-by-one error.

```
seq = [0, 1]
while len(seq) < n - 1:
    seq.append(seq[-1] + seq[-2])
```

**APPS-03 (complexity regression):** a runnable BFS solution uses `list.pop(0)`, preserving the BFS logic but degrading efficiency. **LCE-06 (false-safe structural overlap):** lexical overlap decreases while program structure remains close to the protected pattern.

### B.5 Preliminary Agreement-Profile Validation

Figure 4 supports the core design of ACG: copyright-aware fusion should be region-adaptive rather than uniformly applied.

### B.6 Dataset Statistics.

Following CP-Fuse, we use separate evaluation sets of 500 prompts each for APPS, MBPP, and HumanEval. HumanEval-X contains 820 evaluation samples, with 164 samples for each of Python, Java, JavaScript, Go, and C++. We use the original benchmark splits and introduce no new train/dev/test splits.