COSTA: CODE WATERMARKING VIA STATEMENT-LEVEL ASTS FOR LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models exhibit strong code generation capabilities, accelerating software development and widespread adoption of AI coding assistants. As generated code becomes harder to distinguish from human-written code, legal, ethical, and security concerns motivate watermarking for provenance tracing. However, existing generation-time code watermarking methods rely mainly on token-level biases that can compromise functional correctness, while sentence-level text watermarking remains difficult to transfer directly from natural language to constrained code space. In this paper, we introduce **COSTA**, the first generation-time, statement-level code watermarking framework that encodes provenance in the abstract syntax trees (ASTs) of complete statements. Specifically, a secret key and a locality-sensitive hash of the preceding statement jointly determine the target parity of a selected AST node count. The generator requires no model retraining and embeds watermark signal through three stages: *sample*, *repair*, and *bridge*. The detector tests whether keyed parity matches occur more often than chance, requiring neither the original prompt nor access to the generator. Extensive experiments across models, benchmarks, programming languages, and code transformations demonstrate that COSTA combines strong detectability and robust watermark retention while largely preserving the quality of generated code.

1 INTRODUCTION

Recent advances in large language models (LLMs) have yielded unprecedented capabilities in code generation, matching or even surpassing human experts on a range of programming tasks (Jiang et al., 2026a). This progress has propelled the deployment of specialized code LLMs into widely adopted AI coding assistants, including GitHub Copilot (Chen et al., 2021), Cursor (Cursor, 2026) and Claude Code (Anthropic, 2026). In turn, these automated tools boost researcher and developer productivity while making programming more accessible to beginners (Vaithilingam et al., 2022).

However, as LLM-generated code becomes increasingly sophisticated, the distinction between machine-generated and human-written code continues to blur (Yang et al., 2024). This trend intensifies legal, ethical, and security concerns (Wang et al., 2024; Shi et al., 2025), ranging from code licensing violations and plagiarism to software vulnerabilities and malware generation (Liu et al., 2023; Sandoval et al., 2023; He & Vechev, 2023; Pearce et al., 2025). In response, code watermarking has emerged as a promising approach to tracing code provenance by embedding imperceptible yet detectable signals into generated code (Li et al., 2023b; Sander et al., 2024; Zhao et al., 2025).

Unlike natural language text, code must satisfy strict syntax and remain functionally correct, not merely appear fluent. Existing code watermarking techniques fall into post-hoc and generation-time paradigms grouped by embedding time, as shown in Table 1. Post-hoc watermarking transforms completed programs in a separate stage (Hamilton & Danicic, 2011; Ma et al., 2019; Li et al., 2023a; Yang et al., 2023; Xu et al., 2025). Their limited, language-specific rules are costly to design, and leave watermarks vulnerable once the rules become public. By contrast, generation-time watermarking exploits LLMs' remarkable flexibility to embed signals unobtrusively during generation, inherently compatible with autoregressive decoding.

Despite this flexibility, token-level schemes still dominate generation-time code watermarking (Lee et al., 2024; Kim et al., 2026; Guan et al., 2024; Guo & Cheng, 2025; Guo et al., 2025). Biasing

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison between **COSTA** (ours) and related studies: yes (✔), partial (P), or no (✗).

Watermarking	Detectability	Code Quality	Robustness	Statement-Level	Training-Free
Post-hoc watermarking					
CODEMARK (Li et al., 2023a), SRCMARKER (Yang et al., 2023), ROSEMARY (Zhang et al., 2025)	P	✔	P	P	✗
ACW (Li et al., 2026), MATRIX (Nie et al., 2026), CLASP (Xu et al., 2025), MCW (Choi et al., 2026)	✔	✔	P	P	✔
Generation-time watermarking					
SWEET (Lee et al., 2024), GDW (Yu et al., 2026), STONE (Kim et al., 2026)	✔	P	P	✗	✔
CODEIP (Guan et al., 2024), ACW (Guo & Cheng, 2025), CODETRACER (Guo et al., 2025)	✔	P	P	✗	✗
COSTA (OURS)	✔	✔	✔	✔	✔

individual token choices can compromise syntactic validity or program semantics, while semantics-preserving rewriting can readily erase lexical signals. These limitations motivate embedding the signal in statement structure. Drawing on sentence-level text watermarks (Hou et al., 2024a;b; Dabiriaghdam & Wang, 2025), *we treat statements as code sentences*, using them as syntactically complete signal carriers. Yet the strict grammar and functional requirements of code constrain its generation space, hindering the direct transfer of sentence-level text watermarking methods.[1] This gap calls for a statement-level watermark that accommodates the constraints of code generation.

Consequently, we introduce **COSTA**, a training-free, statement-level framework that encodes provenance within the abstract syntax trees (ASTs) of complete statements. For each statement, a secret key and a locality-sensitive hash (LSH) of its predecessor jointly determine an odd or even target for a selected node count in its AST. Rather than manipulating individual token choices, COSTA realizes this signal through three complementary stages: *sample* selects a naturally matching model candidate, *repair* applies a verified rewrite that preserves semantics, and *bridge* inserts an authenticated statement that leaves execution unchanged, allowing the original candidate to be retained. At detection time, COSTA reparses suspect code and evaluates whether keyed parity matches occur more often than chance, requiring neither the original prompt nor access to the generator. Anchored in canonical program structure, the watermark remains intact under comment removal and identifier renaming, preserving a durable provenance trail even as the code's surface form evolves.

Contributions. Our contributions are as follows:

- We introduce the first generation-time statement-level paradigm tailored to LLM code watermarking, shifting provenance from brittle token choices to complete statements. It maintains syntactic validity while establishing a detectable provenance signal throughout the generated code.
- We instantiate COSTA, a training-free watermarking framework that realizes keyed AST node parity through three stages: *sample*, *repair*, and *bridge*. These stages coordinate candidate selection, local rewriting, and auxiliary insertion to embed signals for model-independent detection.
- We conduct extensive experiments across models, benchmarks, programming languages, and code transformations to evaluate COSTA. The results demonstrate that it combines robust watermark retention with high code quality and strong detectability across the experimental settings studied.

2 RELATED WORK

Code watermarking methods can be broadly categorized into two categories: post-hoc methods that transform completed code and generation-time methods that intervene during decoding (Table 1). Additional LLM watermarking methods are further discussed in Appendix A.

Post-hoc code watermarking. Post-hoc watermarking methods embed ownership information through semantics-preserving edits and thus compatible with black-box deployment. CodeMark (Li et al., 2023a) encodes bits via context-aware variable renaming, while SrcMarker (Yang et al., 2023) extends this approach to natural and formal channels. RoSeMary (Zhang et al., 2025) employs a CodeT5-based encoder-decoder with zero-knowledge verification. Training-free methods instead use predefined transformations: ACW (Li et al., 2026) constructs zero-bit watermarks from idempotent edits; CLASP (Xu et al., 2025) lets an LLM select transformations and recovers bits against

[1] See Appendix H.7 for sentence-level text watermarking results.

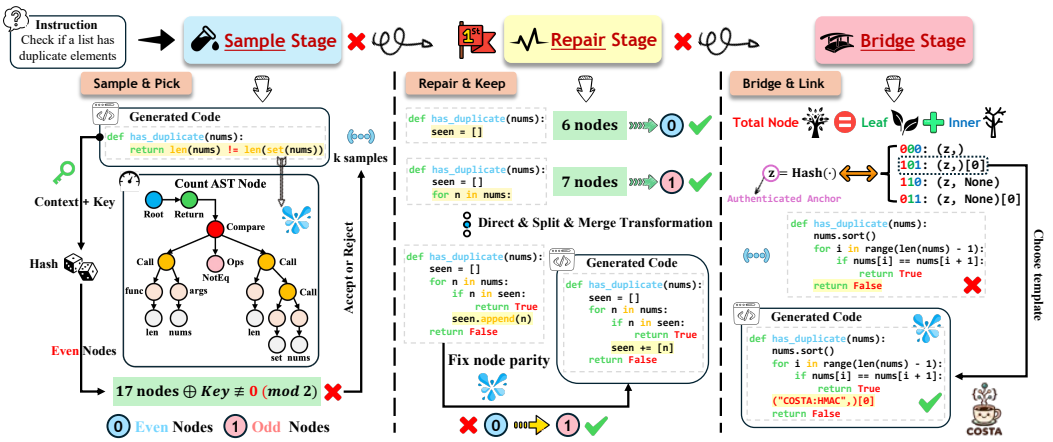


Figure 1: The overview of our COSTA code watermarking framework: *sample*, *repair*, and *bridge*.

inferred original code; MATRIX (Nie et al., 2026) formulates dual-channel embedding as parity constraints solving for hierarchical attribution; and MCW (Choi et al., 2026) incorporates Reed-Solomon codes to provide formal robustness guarantees. Fernandez et al. (2026) further show that rewrite-based watermarking degrades on verifiable outputs such as code. Overall, these methods generally preserve code quality and support multi-bit payloads, but require post-generation edits, sufficient editable locations and a fixed transformation space that may enable evasion once exposed.

Generation-time code watermarking. Generation-time methods inject watermarks by modifying token logits or sampling strategies during decoding. Most adapt text watermarking KGW's red/green-list logit biasing (Kirchenbauer et al., 2023) to code generation, but rigid syntax of code make naive biasing prone to impairing functionality. Therefore, SWEET (Lee et al., 2024) biases only high-entropy tokens, whereas STONE (Kim et al., 2026) restricts watermarking to non-syntactic tokens. GDW (Yu et al., 2026) combines grammar-constrained decoding with role-dependent biasing, while ACW (Guo & Cheng, 2025) learns watermarkable AST positions and constructs context-dependent token partitions. CodeIP (Guan et al., 2024) combines hash-based perturbation with type prediction, while MCGMark (Ning et al., 2025) restricts candidate tokens and avoids mutable regions. CodeTracer (Guo et al., 2025) learns functionality-preserving insertion policies through reinforcement learning and execution feedback. Despite leveraging LLM generation, these token-level methods struggle to capture long-range dependencies required for functional correctness and remain vulnerable to variable renaming, comment removal, and rewriting attacks.

3 Methodology

We introduce **COSTA**, a statement-level code watermarking framework that couples AST node parity with semantic context, as illustrated in Figure 1. The ***generator*** embeds the signal through three successive stages: ***Sample*** stage (Section 3.2) picks candidates satisfying the keyed parity target through stochastic beam search, ***Repair*** stage (Section 3.3) searches for a locally equivalent rewrite when no candidate agrees with the target, and ***Bridge*** stage (Section 3.4) inserts a bounded auxiliary statement when repair fails. The ***detector*** (Section 3.5) reconstructs the same structural observations and context states, then aggregates their parity matches. Algorithms 1 and 2 summarize generation and detection, respectively. More details can be found in Appendix G.

3.1 Problem Setup

Notations. Let $\mathcal{M}$ be an LLM capable of generating code, x a programming prompt given as either a natural language instruction or a code prefix, ℓ the target programming language, and $k \in \mathcal{K}$ a secret key held by the model owner. A statement-level watermarking system for $\mathcal{M}$ comprises a generator $\mathcal{G}$ and a detector $\mathcal{D}$. For any program y, let $\mathsf{C}_\ell(y)$ denote the ordered sequence of carrier statements extracted deterministically in both generation and detection (Appendix G.1).

Prompt Setup. During generation, for each carrier statement s_i, $\mathcal{G}$ makes a watermarking decision based on k, conditioned on x and the generated code prefix $y_{<i} = (s_1, \ldots, s_{i-1})$ preceding s_i. It produces a watermarked program satisfying

$$\mathcal{G}(x, \mathcal{M}, \ell, k) \to y_w, \qquad \mathsf{C}_\ell(y_w) = (s_1, \ldots, s_n). \tag{1}$$

During detection, $\mathcal{D}$ applies the same extraction procedure to a suspect program $\hat{y}$, obtaining $\mathsf{C}_\ell(\hat{y}) = (\hat{s}_1, \ldots, \hat{s}_N)$. It then performs a hypothesis test on the extracted sequence and outputs

$$\mathcal{D}(\hat{y}, \ell, k) \to \delta \in \{0, 1\}, \tag{2}$$

where $\delta = 1$ indicates that the watermark is detected. We assume that $\mathcal{G}$, $\mathcal{D}$, and C_ℓ are publicly known. An adversary may inspect and modify watermarked code but has no knowledge of k.

3.2 COSTA: Sample Stage

Definition 1 (AST Node Parity). *For a carrier statement s, let $\mathcal{A}_\ell(s) = (V_s, E_s)$ be its abstract syntax tree (AST), with node and edge sets V_s and E_s. Let $L_s \subseteq V_s$ be the leaf set and $I_s := V_s \setminus L_s$ the inner-node set, so $V_s = L_s \dot{\cup} I_s$. The tuple $\mathcal{P} = (V, L, I)$, indexed from zero, specifies the total-, leaf-, and inner-node scopes. Defining $n_X(s) := |X_s|$ for $X \in \mathcal{P}$ gives*

$$n_V(s) = |V_s| = |L_s| + |I_s| = n_L(s) + n_I(s). \tag{3}$$

The AST node parity is

$$p_X(s) := n_X(s) \bmod 2 \in \{0, 1\}, \tag{4}$$

where 0 and 1 denote even and odd node counts, respectively.

Definition 2 (Locality-Sensitive Hash). *Let $s_0 := \emptyset$ be the null predecessor. Given a secret key k and fixed code encoder E[2], define the locality-sensitive hash (LSH) with integer output for $i \geq 1$ as*

$$r_i = h(s_{i-1}, k) := \mathrm{LSH}(E(s_{i-1}), k)\,. \tag{5}$$

LSH makes semantically similar statements more likely to share a hash value (Appendix G.2).

Definition 3 (Target Parity). *The scope $d_i := d(s_{i-1}, k)$ and target $b_i := b(s_{i-1}, k)$ are given by*

$$(d_i, b_i) := \begin{cases} (V,\ (r_i \oplus k) \bmod 2)\,, & \textit{fixed-scope mode}, \\ (\mathcal{P}[r_i \bmod 3],\ (r_i \oplus k) \bmod 2)\,, & \textit{adaptive-scope mode}, \end{cases} \tag{6}$$

Here $\oplus$ denotes bitwise XOR with the integer key k, and $b_i \in \{0, 1\}$ selects the target parity.

Intuitively, adaptive scope selection distributes the constraint across three structural statistics, so an attacker cannot deterministically reverse the parity in every scope using a fixed transformation.

For any predecessor c, let $(d(c, k), b(c, k))$ denote the scope and target parity pair derived from $h(c, k)$ by Definition 3. For a candidate v, define the keyed match indicator

$$m(c, v, k) := \mathbb{I}\big[p_{d(c,k)}(v) = b(c, k)\big]\,. \tag{7}$$

Given a budget A, $\mathcal{G}$ uses stochastic beam search to generate up to A candidate branches from the same prefix $y_{<i}$, accepting statements according to

$$\mathrm{Accept}\Big(\tilde{s}_i^{(a)} \sim \mathcal{M}(\cdot \mid x, y_{<i})\Big) \iff m\Big(s_{i-1}, \tilde{s}_i^{(a)}, k\Big) = 1. \tag{8}$$

$\mathcal{G}$ selects a parity matching candidate while keeping the target (d_i, b_i) fixed across all branches.

3.3 COSTA: Repair Stage

When a sampled candidate $\tilde{s}_i$ violates the parity constraint, Repair asks $\mathcal{M}$ for a semantically equivalent rewrite of a local statement or statement pair. We consider three operations.

Direct rewriting. The current statement is replaced by one alternative u, keeping its predecessor and target unchanged. The replacement is accepted if it reverses the mismatched AST node parity:

$$T_{\text{direct}} : \tilde{s}_i \mapsto u, \qquad p_{d_i}(u) = 1 - p_{d_i}(\tilde{s}_i) = b_i. \tag{9}$$

[2]We use `Qwen3-Embedding-0.6B`, which offers support for over 100 languages.

Algorithm 1 COSTA watermark generation

Require: x, $\mathcal{M}$, ℓ, k, A, and ρ
Ensure: watermarked program y_w
$y_w \leftarrow \emptyset$; $c \leftarrow s_0$
while generation is incomplete **do**
  $\tilde{s} \leftarrow \textsc{Sample}(\mathcal{M}, x, y_w, c, k, A)$
  $y' \leftarrow y_w \parallel \tilde{s}$
  if $m(c, \tilde{s}, k) = 0$ **then**
    $y' \leftarrow \textsc{Repair}(\mathcal{M}, y_w, \tilde{s}, c, k)$
    if $y' = \perp$ **then**
      $\alpha \leftarrow \textsc{Bridge}(y_w, \tilde{s}, c, k, \rho)$
      $y' \leftarrow y_w \parallel (\alpha, \tilde{s})$; omit α if $\alpha = \perp$
    end if
  end if
  $y_w \leftarrow y'$; $c \leftarrow \text{last}(\mathsf{C}_\ell(y_w))$
end while
return y_w

Algorithm 2 COSTA watermark detection

Require: suspect program $\hat{y}$, language ℓ, key k, threshold τ
Ensure: decision δ and score $Z(\hat{y})$
$\hat{y} \leftarrow \textsc{Normalize}(\hat{y}, \ell, k)$
$(\hat{s}_1, \ldots, \hat{s}_N) \leftarrow \mathsf{C}_\ell(\hat{y})$
if $N = 0$ **then**
  return $(0, 0)$
end if
$\hat{s}_0 \leftarrow s_0$
for $i = 1, \ldots, N$ **do**
  $(d_i, b_i) \leftarrow (d(\hat{s}_{i-1}, k), b(\hat{s}_{i-1}, k))$
  $m_i \leftarrow \mathbb{I}[p_{d_i}(\hat{s}_i) = b_i]$
end for
$X \leftarrow \sum_{i=1}^{N} m_i$
$Z(\hat{y}) \leftarrow N^{-1/2}(2X - N)$
$\delta \leftarrow \mathbb{I}[Z(\hat{y}) > \tau]$
return $(\delta, Z(\hat{y}))$

Statement splitting. The current statement is split into (u, v) by extracting a subexpression into a fresh temporary used by v. Both must match the target derived from their immediate predecessor:

$$T_{\text{split}} : \tilde{s}_i \mapsto (u, v), \qquad m(s_{i-1}, u, k) = m(u, v, k) = 1. \tag{10}$$

Statement merging. The pair $(s_{i-1}, \tilde{s}_i)$ is merged into one statement u by inlining their shared intermediate value. For $i \geq 2$, the target is derived from s_{i-2}, the predecessor of the replaced pair:

$$T_{\text{merge}} : (s_{i-1}, \tilde{s}_i) \mapsto u, \qquad m(s_{i-2}, u, k) = 1. \tag{11}$$

These operations vary statement syntax and boundaries while preserving the original computation and satisfying sequential parity constraints. Generation and detection derive targets from the same rewritten sequence, allowing LSH values to change without requiring a repair history for detection.

3.4 COSTA: Bridge Stage

Definition 4 (Bridge Connection)**.** *Given a predecessor c and a pending statement s, a bridge is a semantically inert statement α containing an authenticated constant and satisfying*

$$m(c, \alpha, k) = m(\alpha, s, k) = 1. \tag{12}$$

The bridge supplies a temporary context under which the unchanged statement s matches.

When Repair fails, we choose an identity statement α satisfying $m(c, \alpha, k) = 1$, with a constant z authenticating the normalized prefix using k and a random nonce, so retained bridges can detect prefix changes. We search over nonces for $m(\alpha, s, k) = 1$ without changing AST node counts, with a bounded number of trials before continuing without a bridge (Appendix G.4).

For the current prefix of N_t ordinary carriers and B_t bridges, a new bridge requires

$$B_t + 1 \leq \lfloor \rho N_t \rfloor. \tag{13}$$

Theorem 1 in Section 3.5 bounds ρ to preserve detection after all bridges are removed.

3.5 COSTA: Watermark Detection

Given a suspect program $\hat{y}$, the detector removes comments and applies deterministic, syntax-based carrier filtering (Appendix G.1) to obtain $\mathsf{C}_\ell(\hat{y}) = (\hat{s}_1, \ldots, \hat{s}_N)$. With $\hat{s}_0 = s_0$, it reconstructs each carrier's scope and target under k and records

$$m_i := m(\hat{s}_{i-1}, \hat{s}_i, k). \tag{14}$$

This procedure requires neither the generating model $\mathcal{M}$ nor the original prompt x.

Let $X := \sum_{i=1}^{N} m_i$ be the number of matches and $p := \Pr(m_i = 1)$ the match probability. Using $p_0 = 1/2$ as the reference rate for unwatermarked code, we test the null hypothesis $H_0 : p \leq 1/2$

against the alternative hypothesis $H_1 : p > 1/2$. Under the standard independence approximation, the least favorable boundary $p = p_0$ gives $X \sim \text{Binomial}(N, 1/2)$. We therefore normalize the excess number of matches as

$$Z(\hat{y}) := N^{-1/2}(2X - N), \tag{15}$$

where larger scores indicate stronger evidence, with detection at threshold τ defined by $Z(\hat{y}) > \tau$.

Theorem 1 (Detection under bridge removal)**.** *Consider a generated prefix with $N_t > 0$ ordinary carriers, of which M_t match, and $B_t \leq \rho N_t$ bridges. Deleting all bridges without altering the ordinary carrier sequence preserves $Z > \tau$ whenever*

$$0 \leq \rho < \frac{2M_t - N_t - \tau\sqrt{N_t}}{2N_t}. \tag{16}$$

Proof. Let Z^{del} be the score after removing all bridges. Each deletion affects only its successor's match, leaving at least $M_t - B_t$ matches. Thus $Z^{\text{del}} \geq [2(M_t - B_t) - N_t]/\sqrt{N_t} > \tau$ by (16). □

4 Experiments

Overview. Section 4.1 details the datasets, baselines, models, metrics, and implementation. Section 4.2 evaluates watermark detectability, code quality, and cross-language generalization. Section 4.3 examines resilience to semantics-preserving code transformations, while Section 4.4 studies efficiency, hyperparameter sensitivity, and component ablations. Overall, COSTA achieves a consistently superior balance among watermark detectability, code quality, and robustness.

4.1 Experimental Setup

Datasets. Our primary Python testbeds are APPS (Hendrycks et al., 2021) and the MBPP+ and HumanEval+ benchmarks from EvalPlus (Liu et al., 2023). We evaluate model solutions against benchmark test cases and further assess cross-language generalization on six languages from HumanEvalPack (Muennighoff et al., 2024). We sample up to 500 problems per benchmark or language, use the full set when smaller, and report five-run averages. More details are in Appendix B.

Baselines. As COSTA operates at generation time, we compare it with four representative code watermarking: entropy-aware SWEET (Lee et al., 2024), syntax-aware STONE (Kim et al., 2026), grammar-guided CodeIP (Guan et al., 2024), and training-based ACW (Guo & Cheng, 2025). All methods share prompts, samples, backbones, post-processing, and evaluation pipelines. Paired unwatermarked outputs form the negative detection class. Detailed configuration are in Appendix C.

Models. We evaluate three widely adopted open-source models: the general-purpose Llama-3.1-8B-Instruct (Grattafiori et al., 2024) together with two code-specialized models, OpenCoder-8B-IT (Huang et al., 2025) and Qwen2.5-Coder-14B-Instruct (Hui et al., 2024). They span model families and scales, remain frozen, and retain native chat templates; Appendix E details their architectures, context windows, and runtime configurations.

Metrics. We primarily evaluate functional correctness using pass@k (Chen et al., 2021), estimated from $n \geq k$ candidate solutions per problem, and report CodeBLEU (Ren et al., 2020) as a complementary measure of similarity to reference solutions. For detectability, we use AUROC value and report TPR with FPR constrained below 1%. We also report average output length and per sample latency for watermark generation and detection. Formal definitions are provided in Appendix D.

Implementation details. Experiments are conducted in Python 3.12.0 using PyTorch 2.6.0 with CUDA 12.4 on four NVIDIA H800 GPUs, each with 80 GB of memory. The global random seed is set to 42. Unless otherwise specified, we use temperature 1.0, top-$k = 32$, top-$p = 0.95$, repetition penalty 1.05, and minimum and maximum generation lengths of 50 and 512 tokens, respectively. COSTA uses adaptive scope mode by default, with a maximum of three rejection sampling attempts.

Table 2: Detection and functional results on MBPP+, HumanEval+, and APPS for three backbones.

Model	MBPP+ Dataset				HumanEval+ Dataset				APPS Dataset			
+ Watermark	Pass@1 ↑	T@1%F ↑	AUROC ↑	Avg.Len	Pass@1 ↑	T@1%F ↑	AUROC ↑	Avg.Len	Pass@1 ↑	T@1%F ↑	AUROC ↑	Avg.Len
LLAMA-3.1-8B-IT	0.603 ± 0.022	-	-	54.21 ± 1.839	0.564 ± 0.015	-	-	82.30 ± 2.448	0.182 ± 0.008	-	-	155.0 ± 36.45
+ ACW	0.360 ± 0.011	0.394 ± 0.026	**0.843 ± 0.009**	88.36 ± 2.712	0.267 ± 0.020	0.492 ± 0.048	**0.862 ± 0.022**	112.9 ± 4.219	0.124 ± 0.004	0.512 ± 0.037	0.824 ± 0.007	124.4 ± 2.684
+ SWEET	**0.423 ± 0.037**	0.356 ± 0.057	0.836 ± 0.013	73.81 ± 3.107	**0.409 ± 0.018**	0.388 ± 0.091	0.830 ± 0.022	101.9 ± 2.337	**0.137 ± 0.005**	0.568 ± 0.033	0.865 ± 0.012	137.6 ± 6.147
+ STONE	0.423 ± 0.015	0.284 ± 0.047	0.788 ± 0.009	75.80 ± 3.646	0.384 ± 0.032	0.234 ± 0.114	0.794 ± 0.028	105.9 ± 2.301	0.137 ± 0.006	0.462 ± 0.033	0.869 ± 0.009	137.2 ± 4.311
+ CODEIP	0.344 ± 0.020	0.134 ± 0.031	0.758 ± 0.017	69.95 ± 2.548	0.217 ± 0.025	0.443 ± 0.014	0.814 ± 0.028	84.54 ± 3.900	0.122 ± 0.004	0.403 ± 0.206	**0.911 ± 0.007**	154.2 ± 7.505
+ COSTA	**0.548 ± 0.020**	0.263 ± 0.020	**0.871 ± 0.005**	50.31 ± 1.052	**0.475 ± 0.031**	0.463 ± 0.163	**0.946 ± 0.014**	86.92 ± 3.293	**0.161 ± 0.006**	0.616 ± 0.048	**0.946 ± 0.005**	169.6 ± 4.789
OPENCODER-8B-IT	0.690 ± 0.012	-	-	48.01 ± 1.134	0.712 ± 0.023	-	-	68.65 ± 2.403	0.210 ± 0.008	-	-	112.8 ± 2.863
+ ACW	0.503 ± 0.028	0.358 ± 0.094	**0.872 ± 0.032**	68.25 ± 5.568	0.385 ± 0.071	0.460 ± 0.110	**0.886 ± 0.028**	106.1 ± 18.97	0.156 ± 0.004	0.485 ± 0.008	0.891 ± 0.008	136.2 ± 5.170
+ SWEET	**0.592 ± 0.017**	0.281 ± 0.015	0.830 ± 0.012	59.24 ± 1.355	**0.537 ± 0.024**	0.298 ± 0.064	0.831 ± 0.007	85.60 ± 5.415	**0.166 ± 0.009**	0.584 ± 0.036	**0.911 ± 0.004**	133.9 ± 2.629
+ STONE	0.562 ± 0.022	0.191 ± 0.018	0.787 ± 0.011	60.37 ± 1.892	0.508 ± 0.043	0.211 ± 0.063	0.776 ± 0.020	88.75 ± 3.380	0.159 ± 0.006	0.502 ± 0.053	0.911 ± 0.007	141.4 ± 3.530
+ CODEIP	0.493 ± 0.012	0.208 ± 0.052	0.848 ± 0.004	77.11 ± 3.349	0.387 ± 0.033	0.371 ± 0.034	0.862 ± 0.020	102.3 ± 5.455	0.154 ± 0.011	0.317 ± 0.044	0.889 ± 0.007	164.7 ± 5.745
+ COSTA	**0.607 ± 0.010**	0.273 ± 0.045	**0.887 ± 0.009**	52.36 ± 0.392	**0.610 ± 0.037**	0.440 ± 0.064	**0.941 ± 0.009**	80.89 ± 1.982	**0.186 ± 0.006**	0.486 ± 0.024	**0.925 ± 0.010**	117.1 ± 3.369
QWEN2.5-CODER-14B-IT	0.767 ± 0.011	-	-	47.40 ± 0.654	0.853 ± 0.017	-	-	75.85 ± 1.416	0.226 ± 0.009	-	-	102.7 ± 3.788
+ ACW	0.632 ± 0.015	0.116 ± 0.022	0.663 ± 0.011	54.68 ± 1.124	0.658 ± 0.039	0.085 ± 0.033	0.648 ± 0.029	77.44 ± 4.415	0.153 ± 0.004	0.320 ± 0.022	0.779 ± 0.009	97.59 ± 3.777
+ SWEET	0.681 ± 0.021	0.123 ± 0.039	0.708 ± 0.028	55.88 ± 2.849	**0.744 ± 0.034**	0.160 ± 0.035	**0.754 ± 0.009**	88.66 ± 2.920	**0.202 ± 0.011**	0.393 ± 0.024	**0.834 ± 0.008**	114.6 ± 2.257
+ STONE	**0.682 ± 0.012**	0.083 ± 0.004	0.670 ± 0.012	53.67 ± 0.678	0.719 ± 0.021	0.108 ± 0.032	0.706 ± 0.021	88.03 ± 3.205	0.192 ± 0.008	0.284 ± 0.050	0.803 ± 0.010	116.2 ± 2.185
+ CODEIP	0.641 ± 0.014	0.153 ± 0.039	**0.721 ± 0.009**	57.11 ± 2.398	0.633 ± 0.021	0.135 ± 0.043	0.707 ± 0.016	82.73 ± 5.764	0.150 ± 0.005	0.483 ± 0.019	0.807 ± 0.004	128.7 ± 3.326
+ COSTA	**0.704 ± 0.016**	0.311 ± 0.057	**0.927 ± 0.005**	50.93 ± 0.483	**0.762 ± 0.020**	0.482 ± 0.088	**0.945 ± 0.005**	79.32 ± 3.177	**0.208 ± 0.009**	0.677 ± 0.026	**0.966 ± 0.005**	140.1 ± 4.560

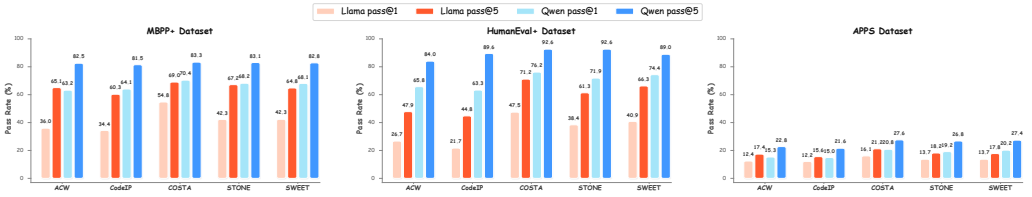


Figure 2: Pass@1 and pass@5 results on MBPP+, HumanEval+, and APPS for two backbones.

4.2 Main Experimental Results

Watermark Detectability. In Table 2, COSTA achieves the highest AUROC across all nine settings, averaging **0.895**, **0.944**, and **0.946** on MBPP+, HumanEval+, and APPS, respectively. The mean relative improvements over the strongest baseline in each setting are **11.2%**, **13.8%**, and **7.1%**. In particular, COSTA averages 0.946 AUROC on Qwen, versus 0.921 on Llama and 0.918 on OpenCoder. This higher AUROC suggests that stronger models offer more diverse code for parity matching. As suggested by the lengths in Appendix H.1, short MBPP+ outputs offer fewer eligible positions for token-level methods, whereas longer APPS solutions provide more observations, consistent with the smaller average AUROC margin on APPS. For practical deployment at 1% FPR, additional calibration data and longer outputs are advisable, as thresholds can be sensitive to unusually high scores from unwatermarked code. Together, these results demonstrate that COSTA's AST parity checks effectively detect watermark signals across the evaluated models and benchmarks.

Code Quality Preservation. These detection gains are accompanied by the highest pass@1 among watermarking methods in all nine settings in Table 2. The mean relative improvements over the strongest baseline in each setting are **11.8%**, **10.7%**, and **10.8%** on MBPP+, HumanEval+, and APPS, respectively. Unlike token-level methods, COSTA selects complete statements and preserves behavior during repair, which can limit disruption to code semantics. Although COSTA's pass@1 remains **6.1** percentage points below unwatermarked generation on average, repeated sampling raises the chance of finding a correct solution. Specifically, Figure 2 and Appendix H.2 show that pass@5 exceeds pass@1 by 5.1 to 24.9 percentage points across nine settings. COSTA leads in eight settings and ties STONE on Qwen HumanEval+. Across methods, lower pass@k on APPS reflects its more demanding problems. Together with the AUROC gains, these results show that COSTA balances watermark detectability and code correctness across the evaluated models and benchmarks.

Cross-language Generalizability. Table 3 shows that COSTA combines high AUROC with strong pass@k across the six HumanEvalPack languages. In particular, COSTA leads nearly all AUROC and pass@k comparisons on Qwen, as detailed in Appendix H.3. On OpenCoder C++ and Rust, Figure 3 shows that CODEIP attains higher AUROC but substantially lower pass@k, whereas COSTA better preserves code quality. On OpenCoder Java, however, COSTA scores 0.490 in CodeBLEU versus SWEET's 0.504, despite higher pass@5 of 87.5% versus 84.7%. CodeBLEU rewards similarity to a single reference and can penalize equally correct alternatives, whereas pass@k evaluates execution against tests and better captures functional correctness. These results further support the extensibility of COSTA's statement-level design across programming languages with varied syntax.

Table 3: Detection and functional results across six HumanEvalPack languages for three backbones.

Model	Python		C++		Java		Rust		JS		Go	
+ Watermark	Pass@1 ↑	AUROC ↑	Pass@1 ↑	AUROC ↑	Pass@1 ↑	AUROC ↑	Pass@1 ↑	AUROC ↑	Pass@1 ↑	AUROC ↑	Pass@1 ↑	AUROC ↑
Llama-3.1-8B-IT	0.674 ± 0.032	-	0.420 ± 0.040	-	0.542 ± 0.053	-	0.183 ± 0.031	-	0.494 ± 0.043	-	0.272 ± 0.044	-
+ ACW	0.493 ± 0.026	0.731 ± 0.023	0.261 ± 0.046	0.934 ± 0.014	0.217 ± 0.021	0.889 ± 0.034	0.029 ± 0.020	0.967 ± 0.013	0.235 ± 0.043	0.823 ± 0.031	0.131 ± 0.024	0.880 ± 0.021
+ SWEET	0.524 ± 0.021	0.754 ± 0.034	0.255 ± 0.026	0.900 ± 0.042	0.314 ± 0.071	0.862 ± 0.049	0.061 ± 0.034	0.975 ± 0.023	0.268 ± 0.062	0.847 ± 0.042	0.089 ± 0.066	0.920 ± 0.016
+ STONE	0.470 ± 0.036	0.774 ± 0.017	0.205 ± 0.021	0.861 ± 0.029	0.206 ± 0.054	0.935 ± 0.012	0.022 ± 0.016	0.988 ± 0.006	0.233 ± 0.056	0.825 ± 0.020	0.052 ± 0.024	0.878 ± 0.007
+ CODEIP	0.452 ± 0.042	0.806 ± 0.031	0.198 ± 0.035	0.943 ± 0.014	0.219 ± 0.039	0.877 ± 0.018	0.029 ± 0.020	0.950 ± 0.010	0.271 ± 0.049	0.848 ± 0.022	0.141 ± 0.031	0.832 ± 0.019
+ COSTA	0.524 ± 0.014	0.914 ± 0.004	0.336 ± 0.044	0.962 ± 0.009	0.431 ± 0.043	0.963 ± 0.007	0.166 ± 0.022	0.911 ± 0.007	0.400 ± 0.044	0.905 ± 0.013	0.185 ± 0.131	0.930 ± 0.012
OpenCoder-8B-IT	0.772 ± 0.028	-	0.545 ± 0.071	-	0.722 ± 0.111	-	0.329 ± 0.032	-	0.647 ± 0.029	-	0.494 ± 0.053	-
+ ACW	0.563 ± 0.026	0.749 ± 0.033	0.189 ± 0.060	0.959 ± 0.015	0.208 ± 0.134	0.928 ± 0.040	0.124 ± 0.022	0.928 ± 0.027	0.405 ± 0.051	0.899 ± 0.031	0.254 ± 0.053	0.847 ± 0.049
+ SWEET	0.626 ± 0.045	0.772 ± 0.020	0.420 ± 0.031	0.924 ± 0.028	0.550 ± 0.055	0.849 ± 0.030	0.193 ± 0.034	0.901 ± 0.027	0.468 ± 0.050	0.917 ± 0.025	0.244 ± 0.034	0.887 ± 0.012
+ STONE	0.589 ± 0.046	0.767 ± 0.018	0.207 ± 0.033	0.906 ± 0.008	0.153 ± 0.010	0.911 ± 0.009	0.088 ± 0.030	0.957 ± 0.013	0.374 ± 0.048	0.881 ± 0.004	0.252 ± 0.052	0.883 ± 0.021
+ CODEIP	0.559 ± 0.059	0.732 ± 0.033	0.139 ± 0.055	0.974 ± 0.001	0.153 ± 0.056	0.918 ± 0.029	0.076 ± 0.016	0.963 ± 0.011	0.311 ± 0.038	0.904 ± 0.020	0.178 ± 0.047	0.864 ± 0.015
+ COSTA	0.693 ± 0.027	0.892 ± 0.022	0.559 ± 0.022	0.935 ± 0.019	0.594 ± 0.036	0.967 ± 0.010	0.300 ± 0.035	0.912 ± 0.007	0.579 ± 0.017	0.913 ± 0.017	0.410 ± 0.034	0.908 ± 0.016
Qwen2.5-Coder-14B-IT	0.913 ± 0.020	-	0.852 ± 0.026	-	0.833 ± 0.182	-	0.707 ± 0.060	-	0.835 ± 0.031	-	0.630 ± 0.066	-
+ ACW	0.761 ± 0.025	0.641 ± 0.007	0.489 ± 0.042	0.778 ± 0.024	0.261 ± 0.021	0.718 ± 0.033	0.285 ± 0.046	0.793 ± 0.021	0.725 ± 0.032	0.731 ± 0.016	0.405 ± 0.014	0.774 ± 0.023
+ SWEET	0.857 ± 0.021	0.681 ± 0.039	0.759 ± 0.032	0.743 ± 0.044	0.733 ± 0.058	0.791 ± 0.033	0.510 ± 0.044	0.748 ± 0.048	0.725 ± 0.038	0.622 ± 0.042	0.514 ± 0.014	0.767 ± 0.034
+ STONE	0.846 ± 0.028	0.662 ± 0.031	0.661 ± 0.030	0.699 ± 0.064	0.331 ± 0.166	0.669 ± 0.008	0.317 ± 0.019	0.816 ± 0.037	0.711 ± 0.038	0.662 ± 0.013	0.459 ± 0.050	0.678 ± 0.050
+ CODEIP	0.683 ± 0.039	0.738 ± 0.021	0.548 ± 0.037	0.848 ± 0.012	0.311 ± 0.045	0.744 ± 0.021	0.324 ± 0.038	0.876 ± 0.026	0.654 ± 0.043	0.762 ± 0.028	0.351 ± 0.037	0.748 ± 0.022
+ COSTA	0.859 ± 0.022	0.946 ± 0.014	0.780 ± 0.046	0.922 ± 0.037	0.803 ± 0.040	0.945 ± 0.038	0.605 ± 0.033	0.881 ± 0.033	0.769 ± 0.024	0.886 ± 0.025	0.610 ± 0.054	0.924 ± 0.018

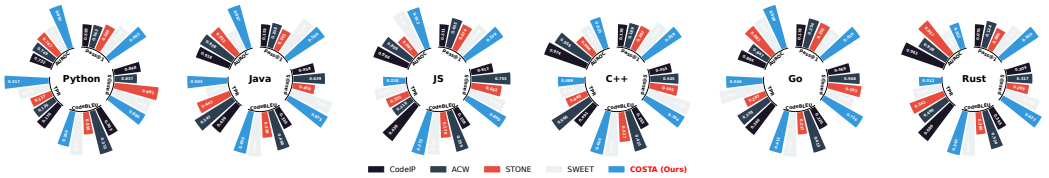


Figure 3: Detection and utility across six HumanEvalPack languages with OpenCoder-8B-IT.

4.3 Robustness Analysis

Attack Settings. Following prior robustness analysis of code watermarks (Suresh et al., 2024), we evaluate five transformations. Insertion adds unreachable dead code, deletion removes comments, modification consistently renames variables, and the mixed attack composes all three operations. Rephrasing prompts GPT-4o-mini to rewrite names, expressions, and formatting while preserving program behavior. Our threat model assumes that an attacker seeks to erase provenance without sacrificing functional correctness. We therefore exclude DIPPER because its natural language paraphraser is not code aware.[3] Attack protocols and implementation details appear in Appendix F.

Attack Results. Figure 4 reports Qwen robustness results, with additional evaluations in Appendix H.4. COSTA achieves the highest AUROC under all five attacks, retaining **0.967** after comment deletion and **0.966** after variable renaming. Comments are not treated as statements, so their removal leaves the carrier sequence unchanged, making COSTA inherently resistant to comment deletion. Renaming preserves AST node counts, while locality-sensitive hashing (LSH) makes semantically similar statements more likely to share a hash, helping preserve the scope and target parity derived from the preceding statement. Under insertion and the mixed attack, COSTA achieves **0.920** and **0.919** AUROC, respectively. Conservative normalization limits interference from dead code before extraction, but remaining insertions can change statement adjacency. LLM rephrasing can alter expressions and statement boundaries, perturbing both AST parity and the context used to determine subsequent targets. COSTA nevertheless retains **0.909** AUROC, exceeding the strongest baseline by **0.174**. These results suggest that aggregating evidence across statements preserves a useful detection margin even when transformations disrupt some of the watermark constraints.

4.4 Efficiency, Hyperparameter, and Ablation Analysis

Generation and Detection Efficiency. Figure 5(a) compares MBPP+ latency on OpenCoder and Qwen, with further results in Appendix H.5. With statement-level rejection sampling, COSTA takes **5.99** and **10.23** s, respectively, consistent with higher inference cost on the larger backbone. Its mean generation time is 3.5 times SWEET's and **5.6%** above CodeIP's. Across these models, Table 2 shows mean MBPP+ gains over CodeIP of **0.123** AUROC and **8.9** percentage points in pass@1, while Figure 4 gives a mean AUROC gain of **0.160** across the Qwen APPS attacks. Detection averages **61.7** ms per program. These gains can justify the added latency when provenance is a priority.

[3] Pilot DIPPER (Krishna et al., 2023) rewrites yielded pass@k near zero.

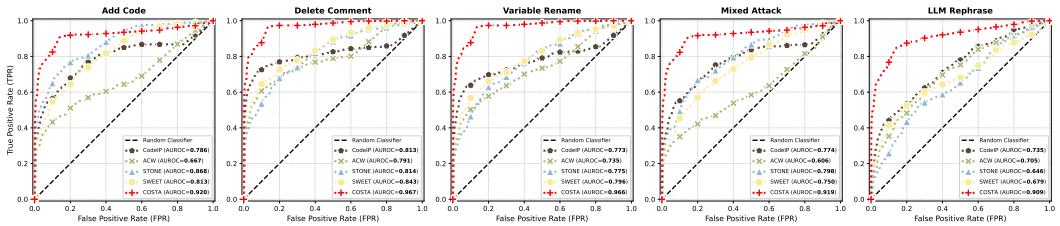


Figure 4: ROC curves under five APPS code transformations with Qwen2.5-Coder-14B-Instruct.

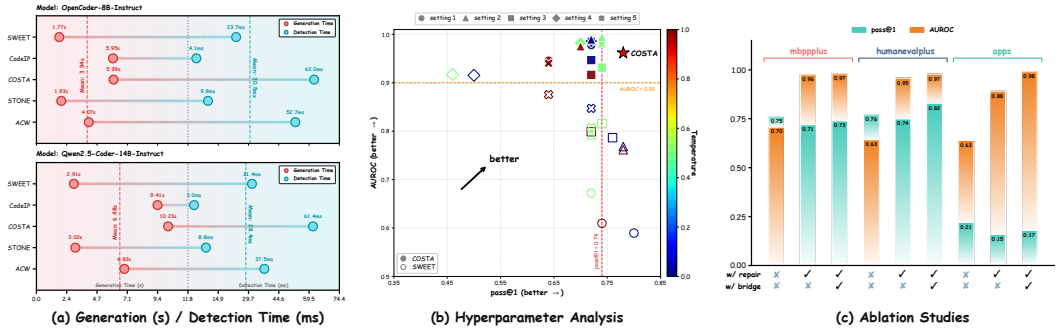


Figure 5: COSTA efficiency, hyperparameter sensitivity, and ablations across three benchmarks.

Since COSTA leaves token generation unchanged, its statement-level operations are compatible in principle with inference acceleration such as speculative decoding (Leviathan et al., 2023).

Hyperparameter Analysis. Figure 5(b) compares temperatures and configurations detailed in Appendix H.6. COSTA retains AUROC above 0.90 throughout the plotted settings, whereas SWEET sometimes achieves higher pass@1 at the expense of detection. Thus, strong detectability does not depend on a single configuration. In a separate sampling study without Repair or Bridge, Table 10 shows that increasing the budget from one to three attempts improves AUROC by 0.146 to 0.211. Increasing it to five adds 0.013 to 0.097 but does not consistently improve pass@1, while Figure 13 shows continued latency growth. We therefore allow up to three attempts by default. Fixed scope has a mean AUROC advantage of 0.010 in Table 11. We retain adaptive scope to vary the protected structural statistic across statements while maintaining comparable detection on unmodified code.

Ablation Studies. Figure 5(c) shows that adding Repair raises AUROC from 0.70/0.63/0.63 to **0.96/0.95/0.88** on MBPP+/HumanEval+/APPS, with full COSTA reaching **0.97/0.97/0.98**. Its pass@1 exceeds Sample alone on HumanEval+ but remains lower on MBPP+ and APPS. Bridge adds the most on APPS, at 0.10 AUROC. Table 7 shows that Sample and Direct together complete **82.4%** to **98.9%** of generation steps, with Direct accounting for most accepted repairs in each setting. Split and Merge provide alternatives when a direct rewrite is insufficient. Bridge completes the remaining transitions and primarily adds auxiliary authentication through a keyed constant bound to the normalized prefix. Retained bridges can therefore help identify changes to preceding code, providing intermediate checks for longer outputs such as APPS solutions. Under the budget bound in Theorem 1, removing bridges preserves detection if the ordinary carrier sequence is unchanged.

5 Conclusion

COSTA reframes generation-time code watermarking around complete statements rather than tokens. It couples keyed AST parity with Sample, Repair, and Bridge, then detects canonical carriers without prompt or generator access. Across models, benchmarks, languages, and transformations, COSTA improves detectability, preserves functional correctness, generalizes broadly, and retains evidence under surface edits and rephrasing. Efficiency and ablations confirm lightweight detection and complementary stage contributions. Future work can extend this structural channel to richer attribution signals and broader language ecosystems as code generators grow in scope and capability.

AI USE STATEMENT

The authors used generative AI tools only for language polishing, LaTeX typesetting and formatting, and icon generation. The tools did not originate any research idea or algorithm, determine the methodology or experiments, analyze results, or draw conclusions. All research ideas, algorithmic designs, scientific decisions, experiments, analyses, and claims were independently developed and verified by the authors. The authors reviewed all AI-assisted edits and generated icons, made all final decisions, and take full responsibility for the manuscript, reported findings, implementation, and accompanying research artifacts.

ETHICS STATEMENT

COSTA aims to promote transparency and accountability in the use of LLM-generated code by supporting provenance tracing and responsible reuse. Our experiments use public code benchmarks and released models, without involving human participants or collecting personal information. We document data sources, experimental settings, and evaluation procedures to support transparent and reproducible research. Responsible deployment should respect software licenses, protect user privacy, and disclose the use of watermarking. Attribution decisions should combine watermark evidence with independent verification and human review to avoid unjustified allegations of misconduct.

REPRODUCIBILITY STATEMENT

Section 3 describes the watermarking method, and Algorithms 1 and 2 summarize the generation and detection procedures. Section 4.1 reports decoding settings, hardware, the random seed, and watermark hyperparameters. Appendices B, C, D, E, and F detail dataset preparation, baseline configurations, metric definitions, backbone models, and attack settings, respectively. Appendix G provides selected few-shot repair examples and describes authenticated bridge construction. Appendix H reports additional results and hyperparameter analyses, and Appendix J presents case studies across datasets and programming languages. The supplement contains COSTA's core implementation code.

REFERENCES

Scott Aaronson. Watermarking of large language models. In *Large Language Models and Transformers Workshop at Simons Institute for the Theory of Computing, 2023.*, 2023. URL https://www.youtube.com/watch?v=2Kx9jbSMZqA.

Anthropic. Claude code overview. https://code.claude.com/docs, 2026. Official Claude Code documentation. Accessed April 12, 2026.

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

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

Ruibo Chen, Yihan Wu, Junfeng Guo, and Heng Huang. Improved unbiased watermark for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20587–20601, 2025.

Soohyeon Choi, Debin Gao, and Yue Duan. Multi-channel spread-spectrum code watermarking. *arXiv preprint arXiv:2607.06009*, 2026.

Cursor. Cursor: The best way to code with AI. https://cursor.com/, 2026. Official product website. Accessed April 12, 2026.

Amirhossein Dabiriaghdam and Lele Wang. Simmark: A robust sentence-level similarity-based watermarking algorithm for large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 30773–30794, 2025.

Sumanth Dathathri, Abigail See, Sumedh Ghaisas, Po-Sen Huang, Rob McAdam, Johannes Welbl, Vandana Bachani, Alex Kaskasoli, Robert Stanforth, Tatiana Matejovicova, Jamie Hayes, Nidhi Vyas, Majd Al Merey, Jonah Brown-Cohen, Rudy Bunel, Borja Balle, Taylan Cemgil, Zahra Ahmed, Kitty Stacpoole, Ilia Shumailov, Ciprian Baetu, Sven Gowal, Demis Hassabis, and Pushmeet Kohli. Scalable watermarking for identifying large language model outputs. *Nature*, 634 (8035):818–823, 2024. doi: 10.1038/s41586-024-08025-4.

Xiaoyan Feng, He Zhang, Yanjun Zhang, Leo Yu Zhang, and Shirui Pan. BiMark: Unbiased multilayer watermarking for large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 17049–17067. PMLR, 2025.

Pierre Fernandez, Tom Sander, Hady Elsahar, Hongyan Chang, Tomáš Souček, Valeriu Lacatusu, Tuan A. Tran, Sylvestre-Alvise Rebuffi, and Alexandre Mourachko. How good is post-hoc watermarking with language model rephrasing? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=ZR1GOaadgP.

Jiayi Fu, Xuandong Zhao, Ruihan Yang, Yuansen Zhang, Jiangjie Chen, and Yanghua Xiao. GumbelSoft: Diversified language model watermarking via the GumbelMax-trick. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5791–5808, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Pan Zhou, and Lichao Sun. Codeip: A grammar-guided multi-bit watermark for large language models of code. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9243–9258, 2024.

Zhimeng Guo and Minhao Cheng. Practical and effective code watermarking for large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=RpE4HeuX69.

Zhimeng Guo, Huaisheng Zhu, Siyuan Xu, Hangfan Zhang, Teng Xiao, and Minhao Cheng. Optimizing token choice for code watermarking: An rl approach. *arXiv preprint arXiv:2508.11925*, 2025.

James Hamilton and Sebastian Danicic. A survey of static software watermarking. In *2011 World Congress on Internet Security (WorldCIS-2011)*, pp. 100–107. IEEE, 2011.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Abe Hou, Jingyu Zhang, Tianxing He, Yichen Wang, Yung-Sung Chuang, Hongwei Wang, Lingfeng Shen, Benjamin Van Durme, Daniel Khashabi, and Yulia Tsvetkov. Semstamp: A semantic watermark with paraphrastic robustness for text generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4067–4082, 2024a.

Abe Hou, Jingyu Zhang, Yichen Wang, Daniel Khashabi, and Tianxing He. k-semstamp: A clustering-based semantic watermark for detection of machine-generated text. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 1706–1715, 2024b.

Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. Unbiased watermark for large language models. In *International Conference on Learning Representations*, volume 2024, pp. 45408–45436, 2024.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33167–33193, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Jiahao Huo, Shuliang Liu, Bin Wang, Junyan Zhang, Yibo Yan, Aiwei Liu, Xuming Hu, and Mingxun Zhou. PMark: Towards robust and distortion-free semantic-level watermarking with channel constraints. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=EhDgP69DJG.

Jiahao Huo, Wenjie Qu, Yibo Yan, Kening Zheng, Jiaheng Zhang, Xuming Hu, Philip S Yu, and Mingxun Zhou. Samark: A self-anchored text watermarking with paragraph-level paraphrase robustness. *arXiv preprint arXiv:2605.25796*, 2026b.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026a.

Ya Jiang, Chuxiong Wu, Massieh Kordi Boroujeny, Brian Mark, and Kai Zeng. StealthInk: A multi-bit and stealthy watermark for large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 27685–27709. PMLR, 2025.

Ya Jiang, Massieh Kordi Boroujeny, Surender Suresh Kumar, and Kai Zeng. Mirrormark: Generalizable mirrored sampling for multi-bit llm watermarking. *arXiv preprint arXiv:2601.22246*, 2026b.

Jungin Kim, Shinwoo Park, and Yo-Sub Han. Marking code without breaking it: Code watermarking for detecting llm-generated code. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 3990–4002, 2026.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *International conference on machine learning*, pp. 17061–17084. PMLR, 2023.

Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Wieting, and Mohit Iyyer. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *Advances in neural information processing systems*, 36:27469–27500, 2023.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. Robust distortion-free watermarks for language models. *arXiv preprint arXiv:2307.15593*, 2023.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 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)*, pp. 4890–4911, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Boquan Li, Zirui Fu, Mengdi Zhang, Peixin Zhang, Jun Sun, and Xingmei Wang. Efficient and universal watermarking for llm-generated code detection. *IEEE Transactions on Software Engineering*, 2026.

Wei Li, Borui Yang, Yujie Sun, Suyu Chen, Ziyun Song, Liyao Xiang, Xinbing Wang, and Chenghu Zhou. Towards tracing code provenance with code watermarking. *arXiv preprint arXiv:2305.12461*, 2023a.

Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. Protecting intellectual property of large language model-based code generation apis via watermarks. In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, pp. 2336–2350, 2023b.

Aiwei Liu, Leyi Pan, Xuming Hu, Shiao Meng, and Lijie Wen. A semantic invariant robust watermark for large language models. In *International Conference on Learning Representations*, volume 2024, pp. 6499–6519, 2024a.

Aiwei Liu, Leyi Pan, Yijian Lu, Jingjing Li, Xuming Hu, Xi Zhang, Lijie Wen, Irwin King, Hui Xiong, and Philip Yu. A survey of text watermarking in the era of large language models. *ACM Computing Surveys*, 57(2):1–36, 2024b.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Yijian Lu, Aiwei Liu, Dianzhi Yu, Jingjing Li, and Irwin King. An entropy-based text watermarking detection method. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11724–11735, 2024.

Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. Xmark: dynamic software watermarking using collatz conjecture. *IEEE Transactions on Information Forensics and Security*, 14(11):2859–2874, 2019.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pp. 7604–7623, 2024.

Yuqing Nie, Chong Wang, Guosheng Xu, Guoai Xu, Chenyu Wang, Haoyu Wang, and Kailong Wang. Matrix: Multi-layer code watermarking via dual-channel constrained parity-check encoding. *arXiv preprint arXiv:2604.16001*, 2026.

Kaiwen Ning, Jiachi Chen, Qingyuan Zhong, Tao Zhang, Yanlin Wang, Wei Li, Jingwen Zhang, Jianxing Yu, Yuming Feng, Weizhe Zhang, and Zibin Zheng. MCGMark: An encodable and robust online watermark for tracing LLM-generated malicious code. *arXiv preprint arXiv:2408.01354*, 2025.

Leyi Pan, Aiwei Liu, Zhiwei He, Zitian Gao, Xuandong Zhao, Yijian Lu, Binglin Zhou, Shuliang Liu, Xuming Hu, Lijie Wen, et al. Markllm: An open-source toolkit for llm watermarking. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 61–71, 2024.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105, 2025.

Wenjie Qu, Wengrui Zheng, Tianyang Tao, Dong Yin, Yanze Jiang, Zhihua Tian, Wei Zou, Jinyuan Jia, and Jiaheng Zhang. Provably robust multi-bit watermarking for AI-generated text. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 201–220. USENIX Association, 2025.

Jie Ren, Han Xu, Yiding Liu, Yingqian Cui, Shuaiqiang Wang, Dawei Yin, and Jiliang Tang. A robust semantics-based watermark for large language model against paraphrasing. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 613–625, 2024a.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Yubing Ren, Ping Guo, Yanan Cao, and Wei Ma. Subtle signatures, strong shields: Advancing robust and imperceptible watermarking in large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 5508–5519, 2024b.

Tom Sander, Pierre Fernandez, Alain Durmus, Matthijs Douze, and Teddy Furon. Watermarking makes language models radioactive. *Advances in Neural Information Processing Systems*, 37: 21079–21113, 2024.

Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 2205–2222, 2023.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1628–1639. IEEE, 2025.

Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is watermarking LLM-generated code robust? In *The Second Tiny Papers Track at ICLR 2024*, 2024. URL https://openreview.net/forum?id=8PhI1PzSYY.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.

Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*, 2024.

Yidan Wang, Yubing Ren, Yanan Cao, and Binxing Fang. From trade-off to synergy: A versatile symbiotic watermarking framework for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10306–10322, 2025a.

Yidan Wang, Yubing Ren, Yanan Cao, and Li Guo. WorldCup sampling for multi-bit LLM watermarking. *arXiv preprint arXiv:2602.01752*, 2026. URL https://arxiv.org/abs/2602.01752.

Zongqi Wang, Tianle Gu, Baoyuan Wu, and Yujiu Yang. Morphmark: Flexible adaptive watermarking for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4842–4860, 2025b.

Yihan Wu, Zhengmian Hu, Hongyang Zhang, and Heng Huang. Dipmark: A stealthy, efficient and resilient watermark for large language models. *arXiv preprint arXiv:2310.07710*, 2023.

Rui Xu, Jiawei Chen, Weizhi Liu, Zhaoxia Yin, Cong Kong, and Xinpeng Zhang. Clasp: Training-free llm-assisted source code watermarking via semantic-preserving transformations. *arXiv preprint arXiv:2510.11251*, 2025.

Borui Yang, Wei Li, Liyao Xiang, and Bo Li. Towards code watermarking with dual-channel transformations. *arXiv preprint arXiv:2309.00860*, 2023.

Xianjun Yang, Liangming Pan, Xuandong Zhao, Haifeng Chen, Linda Ruth Petzold, William Yang Wang, and Wei Cheng. A survey on detection of llms-generated content. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9786–9805, 2024.

KiYoon Yoo, Wonhyuk Ahn, and Nojun Kwak. Advancing beyond identification: Multi-bit watermark for large language models via position allocation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4031–4055, 2024.

Licheng Yu, Aiwei Liu, and Songze Li. Toward stronger code watermarking: A grammar-driven approach to optimizing the trade-off between quality and detectability. *arXiv preprint arXiv:2607.10210*, 2026.

Zhuohao Yu, Xingru Jiang, Weizheng Gu, Yidong Wang, Qingsong Wen, Shikun Zhang, and Wei Ye. SAEMark: Steering personalized multilingual LLM watermarks with sparse autoencoders. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=tXnyVPNOfa.

Ruisi Zhang, Neusha Javidnia, Nojan Sheybani, and Farinaz Koushanfar. Robust and secure code watermarking for large language models via ml/crypto codesign. *arXiv preprint arXiv:2502.02068*, 2025.

Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for AI-generated text. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=SsmT8aO45L.

Xuandong Zhao, Sam Gunn, Miranda Christ, Jaiden Fairoze, Andres Fabrega, Nicholas Carlini, Sanjam Garg, Sanghyun Hong, Milad Nasr, Florian Tramer, et al. Sok: Watermarking for ai-generated content. In *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 2621–2639. IEEE, 2025.

A More Related Work

LLM watermarking embeds statistically verifiable signals in model outputs. Existing methods are commonly classified by the granularity at which these signals are imposed (Liu et al., 2024b).

Token-level Watermarking. These schemes alter each autoregressive decision and test for a keyed signal. Representative approaches such as KGW (Kirchenbauer et al., 2023) increase the logits of a green list determined by the preceding context and detect excess green tokens. Later work reduces distributional distortion through unbiased reweighting or reweighting across multiple channels (Hu et al., 2024; Wu et al., 2023; Chen et al., 2025; Wang et al., 2025b). Detection and resilience to edits improve through entropy weighting, token priors, partitions independent of context, or semantic conditioning (Lu et al., 2024; Ren et al., 2024b; Zhao et al., 2024; Liu et al., 2024a; Ren et al., 2024a). Other designs act directly on token selection: EXP (Aaronson, 2023) samples using exponential random variables; successors include sampling with key sequences (Kuditipudi et al., 2023), GumbelSoft (Fu et al., 2024), and SynthID-Text (Dathathri et al., 2024). SymMark (Wang et al., 2025a) adaptively combines logit and sampling signals. These approaches provide many observations at modest generation cost, but typically require control over decoding and remain susceptible to lexical edits, paraphrasing, and loss of synchronization with the key.

Multi-bit watermarking. These schemes encode provenance metadata, such as user or model identifiers. MPAC (Yoo et al., 2024) maps tokens to message positions, supporting long payloads without enumerating messages. Qu et al. (2025) combine balanced segment assignment with Reed-Solomon coding for robust, efficient decoding under bounded edit distance. BiMark (Feng et al., 2025) uses XOR-based position allocation and multilayer unbiased reweighting for extraction across models. StealthInk (Jiang et al., 2025) preserves the marginal distribution through probability intervals under secret vocabulary permutations, resisting distribution-based spoofing. WorldCup (Wang et al., 2026) enables robust multi-bit watermarking through hierarchical tournament sampling and confidence-aware decoding.MirrorMark (Jiang et al., 2026b) uses mirrored sampling to improve generalizability and robustness. Together, these methods extend watermarking from detection to attribution. Longer payloads dilute evidence per bit, while text with low entropy and paraphrasing impair extraction, creating tradeoffs among capacity, quality, detectability, robustness, and security.

Sentence-level watermarking. Semantic approaches encode signals in sentence representations and better withstand lexical rewriting. SemStamp (Hou et al., 2024a) partitions embedding space into keyed regions; k-SemStamp (Hou et al., 2024b) improves locality and sampling efficiency through clustering adapted to the domain. SimMark (Dabiriaghdam & Wang, 2025) marks similarity between sentences and uses soft counts to tolerate embedding perturbations. PMark (Huo et al., 2026a) combines scalar proxy functions with constraints across multiple channels to improve robustness while preserving the output distribution. SAEMark (Yu et al., 2025) uses sparse autoencoder features to encode multiple bits for personalized attribution across models and languages. SAMark (Huo et al., 2026b) defines semantic regions from each sentence itself, removing reliance on preceding sentences and positions and improving resilience to paragraph paraphrasing and sentence reordering. These methods are primarily designed for natural language text rather than source code. Syntax, variable dependencies, and execution semantics constrain the space of valid code variations, making these methods difficult to transfer directly and effectively to code generation.

B DATASETS

We evaluate code generation on APPS (Hendrycks et al., 2021), HumanEval+ and MBPP+ from EvalPlus (Liu et al., 2023), and HumanEvalPack (Muennighoff et al., 2024). Table 4 summarizes the datasets. Using seed 42 for each benchmark, we sample 500 instances without replacement or retain smaller pools in full. We run reference solutions in their benchmark harness (Ben Allal et al., 2022) and exclude instances with failed assertions, exceptions, compilation errors, or timeouts. This avoids counting conversion errors as model failures. We strip prompt echoes, Markdown fences, and surrounding natural language from generated outputs to retain only source code wherever possible.

Table 4: Validated dataset pools and sample sizes. Pools under 500 instances are evaluated in full.

Benchmark	Pool	Evaluated	Languages	Timeout
APPS	1,000	500	Python	10 s
HumanEval+	163	163	Python	20 s
MBPP+	378	378	Python	10 s
HumanEvalPack	977	500	Six languages	By language

APPS. APPS contains introductory, interview, and competition problems using standard input or function calls. The shortest official solution passing all supplied tests serves as the reference, yielding a stratified pool of 1,000. We draw 500 instances without replacement using seed 42. A generated candidate is correct only if it passes all supplied tests within 10 seconds per invocation.

HumanEval+ and MBPP+. EvalPlus strengthens the HumanEval and MBPP suites with additional tests, reducing false acceptance of incorrect programs. Validation retains all 163 HumanEval+ problems, which we evaluate with the complete augmented driver and a timeout of 20 seconds. All 378 MBPP+ problems also pass validation. Their three example assertions appear in the prompt, whereas augmented tests remain hidden and serve only for evaluation with a timeout of 10 seconds.

HumanEvalPack. HumanEvalPack spans function synthesis in Python, C++, Java, JavaScript, Go, and Rust. We retain each task's declaration, imports, setup code, example test, and hidden harness. Validation leaves 977 instances, from which we sample 500 using seed 42: 92 Python, 88 C++, 72 Java, 85 JavaScript, 81 Go, and 82 Rust. Timeouts are 10 seconds for Python, JavaScript, and Java, 60 seconds for C++, 20 seconds for Go, and 300 seconds for Rust to cover compilation overhead. A compilation error, nonzero exit status, failed assertion, or timeout counts as incorrect.

Prompting and execution. All benchmarks use zero-shot prompts built with each backbone's native instruction template. We request source code only and include public examples when available. Hidden tests never enter the prompt. We execute candidates in isolated subprocesses with four parallel workers. For each prompt and method, we conduct five stochastic runs with distinct run identifiers on the same task set selected using seed 42. We report the mean and standard deviation.

C BASELINES

We compare four baselines that modify token sampling: SWEET (Lee et al., 2024), STONE (Kim et al., 2026), CodeIP (Guan et al., 2024), and ACW (Guo & Cheng, 2025). For methods using a green list, let $\boldsymbol{\ell}_t \in \mathbb{R}^{|\mathcal{V}|}$ denote the backbone logits at step t and $G_t \subset \mathcal{V}$ a keyed subset containing a fraction γ of the vocabulary. The watermark increases the logits of all green tokens according to

$$\widetilde{\ell}_{t,v} = \ell_{t,v} + \delta\, \mathbb{I}[v \in G_t], \tag{17}$$

where $\delta > 0$ controls the bias. Methods differ in eligible positions and the construction of G_t. All use the decoding settings in Section 4.1, including temperature 1.0. Individual predictions use each method's threshold, while comparisons use the common 1% FPR calibration in Appendix D.

SWEET. SWEET applies Equation 17 at positions with high predictive entropy. It computes Shannon entropy using the unwatermarked distribution $p_t(v) = \mathrm{softmax}(\boldsymbol{\ell}_t)_v$ for the next token,

$$H_t = -\sum_{v \in \mathcal{V}} p_t(v) \log p_t(v). \tag{18}$$

We use $\gamma = 0.5$, $\delta = 4.0$, hash key 15,485,863, prefix length 1, and a z threshold of 4.0. SWEET embeds only when $H_t > 0.3$, avoiding positions with low entropy. Detection reconstructs eligible positions and tests whether green tokens occur more often than expected under the null hypothesis.

STONE. STONE excludes syntax tokens, including keywords, whitespace, types, delimiters, and operators, and applies the bias to the remaining tokens. We use $\gamma = 0.5$, $\delta = 4.0$, hash key 15,485,863, prefix length 1, `skipping_rule=all_pl`, and `watermark_on_pl=false`. We select language rules automatically and use a z threshold of 4.0. This limits syntactic interference.

CodeIP. CodeIP encodes multiple bits with a soft watermark that combines signals from the backbone, the message, and a grammar predictor before sampling. The grammar guidance favors syntactically compatible token types. We encode message [0] with a message code of length 1, $\delta = 3.0$, $\gamma = 3.0$, $\beta = 0.0$, and encoding ratio 10.0. We use `top_k=1000` as an internal candidate cap, a hash prefix of length 1, seed 42, and a z threshold of 0.75. PDA guidance is enabled with automatic language selection. The internal cap is separate from the shared decoding setting top-$k = 32$.

ACW. ACW uses ASTs to supervise a Watermark Partitioning Network (WPN). At step t, its two output heads map a context of c preceding tokens to partition logits $\boldsymbol{\ell}_t^{\mathrm{WPN}}$ and a switch score g_t,

$$(\boldsymbol{\ell}_t^{\mathrm{WPN}}, g_t) = \mathrm{WPN}(y_{t-c:t}). \tag{19}$$

The switch head identifies substitutable positions. Tokens with similar WPN logits are paired, and a keyed bit assigns the two members to the green and red lists. Equation 17 then biases the green list at these positions. We use $\gamma = 0.5$, $\delta = 4.0$, hash key 15,485,863, context width $c = 1$, raw switch threshold -10, and a z threshold of 4.0. The WPN has six layers, hidden size 512, eight attention heads, feedforward size 2,048, dropout 0.2, and maximum input length 64. Its training objective is

$$\mathcal{L}_{\mathrm{WPN}} = 0.6\mathcal{L}_{\mathrm{distill}} + 0.2\mathcal{L}_{\mathrm{CE}} + 0.2\mathcal{L}_{\mathrm{switch}}. \tag{20}$$

For each backbone model, we automatically select the corresponding pretrained WPN checkpoint.

D METRICS

For each model, dataset, and method, we evaluate M prompts per run. For $i = 1, \ldots, M$, let x_i^+ and x_i^- denote the paired watermarked and unwatermarked outputs, respectively. For any output x, $s(x)$ denotes its detection score, with larger values indicating stronger evidence of a watermark.

Detection metrics and operating point. At threshold τ, we classify output x as watermarked if $s(x) \geq \tau$. Let TP, FP, TN, and FN count true positives, false positives, true negatives, and false negatives, respectively. The true positive rate (TPR), false positive rate (FPR), and F1 score are

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad \mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}, \qquad \mathrm{F1} = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}. \tag{21}$$

We also report accuracy $\mathrm{ACC} = (\mathrm{TP} + \mathrm{TN})/(2M)$, true negative rate $\mathrm{TNR} = 1 - \mathrm{FPR}$, and false negative rate $\mathrm{FNR} = 1 - \mathrm{TPR}$. For each run, let $\mathcal{T}$ denote the midpoints between adjacent sorted scores from both output groups. We select the lowest threshold satisfying the 1% FPR constraint:

$$\tau^* = \min\{\tau \in \mathcal{T} : \mathrm{FPR}(\tau) \leq 0.01\}. \tag{22}$$

TPR@1%FPR denotes TPR evaluated at τ^*. Calibration and evaluation share the same run, so these rates describe performance at the observed operating point. Our primary metric is the area under the receiver operating characteristic curve (AUROC), which does not require a decision threshold:

$$\mathrm{AUROC} = \Pr\big[s(x^+) > s(x^-)\big] + 0.5 \Pr\big[s(x^+) = s(x^-)\big]. \tag{23}$$

Here, x^+ and x^- are independent uniform draws from the watermarked and unwatermarked output groups in each run after excluding empty token sequences. Threshold metrics retain all samples.

For each output x, let N count eligible positions and N_G count the green tokens among them. For a green list that covers a fraction $\gamma \in (0, 1)$ of the vocabulary, the conventional detection score is

$$z = (N_G - \gamma N)/\sqrt{N\gamma(1-\gamma)}, \qquad N > 0. \tag{24}$$

Functional correctness. Pass@k is the probability that at least one of k sampled candidates passes all hidden tests. For prompt i, let n_i be the number of generated candidates and c_i the number that pass. For $1 \le k \le n_i$, the unbiased estimator (Chen et al., 2021) and its mean over M prompts are

$$\widehat{\text{pass@}k}_i = 1 - \binom{n_i - c_i}{k} \Big/ \binom{n_i}{k}, \qquad \text{pass@}k = \frac{1}{M}\sum_{i=1}^{M} \widehat{\text{pass@}k}_i. \tag{25}$$

Each run supplies one candidate per prompt. We compute pass@1 across prompts within each run and report the mean and standard deviation over five runs. For pass@5, we pool the five independent candidates for each prompt, with $n_i = 5$, before applying Equation 25. We assess watermarked and unwatermarked outputs separately and rerun all hidden tests after every robustness transformation.

Code similarity. For candidate x and reference r, CodeBLEU (Ren et al., 2020) measures lexical and structural agreement using the ordinary n-gram matching score S_{ngram}, weighted n-gram score S_{weighted}, AST matching score S_{syntax}, and data flow matching score S_{dataflow} with equal weights:

$$\text{CodeBLEU}(x, r) = \frac{1}{4}\left(S_{\text{ngram}} + S_{\text{weighted}} + S_{\text{syntax}} + S_{\text{dataflow}}\right). \tag{26}$$

We restore both declarations or prefixes, score in the task language, and average across prompts.

Length and runtime. After removing surrounding natural language as in Appendix B, let L_i denote the number of tokens in the retained code for prompt i, counted with the corresponding backbone's tokenizer. We report the mean code length over all M prompts for each output group as

$$\overline{L} = \frac{1}{M}\sum_{i=1}^{M} L_i. \tag{27}$$

Embedding time averages watermarked generation time per output across batches, excluding unwatermarked generation. Let T_i be the time in seconds for scoring x_i^+. Mean scoring time is

$$\overline{T}_{\text{det}} = \frac{1}{M}\sum_{i=1}^{M} T_i. \tag{28}$$

E Backbone Models

We evaluate Llama-3.1-8B-Instruct (Grattafiori et al., 2024) as a general instruction model and OpenCoder-8B-IT (Huang et al., 2025) and Qwen2.5-Coder-14B-Instruct (Hui et al., 2024) as models specialized for code. Table 5 lists the architectures and context limits of these three backbones.

Table 5: Architecture and context limits of the three backbones used in our experiments.

Model	Parameters	Layers	Hidden size	Q/KV heads	Context
Llama-3.1-8B-Instruct	8B	32	4,096	32/8	128K
OpenCoder-8B-IT	8B	32	4,096	32/8	8K
Qwen2.5-Coder-14B-Instruct	14.7B	48	5,120	40/8	128K

Q/KV lists the query and key/value head counts in grouped query attention. Hidden size is the token representation dimension. B denotes billions of parameters, and K denotes context length in thousands of tokens.

We load all three Transformer decoders in half precision (FP16) and keep their weights frozen. Each run uses one NVIDIA H800 GPU with 80 GB of memory and batch size 4, together with the model's native tokenizer and chat template. ACW uses the pretrained Watermark Partitioning Network (WPN) described in Appendix C. All comparisons use the same prompts and generation settings from Section 4.1, including temperature 1.0 and a maximum of 512 generated tokens.

F Attack Settings

We evaluate five attacks on 500 APPS problems, transforming watermarked outputs and retaining paired unwatermarked outputs as negative examples. The first four attacks use random seed 42.

- **Add code.** We sequentially insert one comment between existing lines, add one unreachable block under a false condition before a statement, and enclose one statement in `try`/`except`. Each operation selects a random location from those available in the current version of the program.
- **Delete comment.** We remove all comments and documentation strings except those whose removal would leave a module, function, or class body empty. The remaining layout is unchanged.
- **Variable rename.** Every eligible local variable is consistently replaced with a new name formed by joining two random words with an underscore. We keep function parameters, Python keywords, common built-in names, and names beginning with two underscores unchanged.
- **Mixed attack.** We first add one unreachable block and enclose one statement in `try`/`except`, then rename all eligible local variables and apply comment deletion using the settings above.
- **LLM rephrasing.** We use `gpt-4o-mini` at temperature 0.8 with a maximum of 1,024 output tokens to rephrase each watermarked program. The prompt is shown in the box below.

Rephrasing prompt

You are an expert Python developer. Rewrite the following Python code while preserving its EXACT behavior and semantics. You may:
- rename variables and function arguments to other meaningful names
- replace expressions with semantically equivalent ones
- adjust whitespace and formatting

Rules:
- Do NOT change the code's functionality or its external interface.
- Do NOT add, remove, or modify any comments.
- Output ONLY the rewritten code: no markdown fences, no explanations, no prose.

G Additional COSTA Details

G.1 Carrier Statement Extraction

Generation and detection share the parsing and AST counting rules for $\mathsf{C}_\ell(y)$ (Section 3.1). We use Python's standard `ast` module and Tree-sitter for Rust, Go, Java, JavaScript, and C++. Carriers include simple statements and selected control headers, processed in source order. A selected header precedes any carriers extracted from its body, and nested blocks are visited before subsequent statements. Comments, imports, documentation strings, and enclosing function or class syntax are excluded. After normalization, detection derives each target from the preceding retained carrier.

For Python `if` and `for` headers, including `async for`, a temporary `pass` body makes the fragment parseable. The body is removed before AST nodes are counted. Both stages then count only the control node and header syntax, as do the parsers for other languages. This lets Sample evaluate a header before generating its body and prevents body edits from changing the header's parity.

Detection uses local syntax rules to counter dead code insertion. Within each block, we exclude branches with a known false condition and no `else`, statements after unconditional control transfers, and selected unused constant assignments. Local constant evaluation resolves simple conditions. Unresolved conditions are retained. Empty statements, standalone literals, and recognized output calls are also excluded. The source remains intact for Bridge authentication (Appendix G.4).

G.2 Locality-Sensitive Hashing

Following SemStamp's LSH strategy (Hou et al., 2024a), we instantiate Definition 2 using an 8-bit random hyperplane LSH with standard Gaussian projection vectors deterministically generated from the secret key k and reused in generation and detection. Each inner product with $E(s_{i-1})$ yields one bit (1 if positive, 0 otherwise); the ordered bits encode r_i, with the first bit most significant.

Before encoding, we canonicalize identifiers while retaining literals. Identical canonical inputs yield the same hash, whereas nearby embeddings collide only probabilistically. Exact collisions are not guaranteed, and a bit flip can change the scope or target in Definition 3. Detection aggregates parity matches, each preserved when the statement and its predecessor remain unchanged.

G.3 Repair Prompts, Examples, and Statistics

We guide $\mathcal{M}$ using the instructions below and examples selected for the current operation (Direct, Split, or Merge). Few-shot inputs alternate `user` messages containing context and original statements with `assistant` messages containing rewrites. Table 6 presents partial Python examples.

Table 6: Selected Python repair examples. ↪ denotes a line break.

Local context	Code to replace	Demonstrated response
Direct rewriting		
`a = 1↪b = 2`	`c = a + b`	`c = sum((a, b))`
`def f(a):`	`return not a`	`return False if a else True`
`a = []↪b = 1`	`a.append(b)`	`a.extend([b])`
`a = 1↪b = 2↪c = 3`	`if a <= b and b < c:`	`if a <= b < c:`
`a = [1, 2]`	`for b in a:`	`for b in iter(a):`
Statement merging		
`a = ' X '`	`b = a.strip()↪c = b.lower()`	`c = a.strip().lower()`
`def f(a):`	`b = a + 1↪return abs(b)`	`return abs(a + 1)`
`a = []↪b = ' x '`	`c = b.strip()↪a.append(c)`	`a.append(b.strip())`
`a = [1, 2]`	`b = len(a)↪assert b == 2`	`assert len(a) == 2`
`def f(a):`	`b = str(a)↪raise ValueError(b)`	`raise ValueError(str(a))`
Statement splitting		
`a = 1`	`c = abs(a + 1)`	`b = a + 1↪c = abs(b)`
`def f(a):`	`return abs(a + 1)`	`b = a + 1↪return abs(b)`
`a = []↪b = ' x '`	`a.append(b.strip())`	`c = b.strip()↪a.append(c)`
`a = [1, 2]`	`assert len(a) == 2`	`b = len(a)↪assert b == 2`
`a = 1↪b = 3`	`if a + 1 < b:`	`c = a + 1↪if c < b:`

Repair Instruction

Direct: Rewrite this statement as one equivalent statement using a different expression or syntax.
Split: Split this statement into two or more statements using fresh temporaries and related intermediate computations.
Merge: Merge these two adjacent statements into one by inlining their shared intermediate value.
Shared constraints: Require known runtime types for type-specific rewrites; annotations are insufficient. Preserve values, object identity, evaluation order, side effects, and needed names. Keep an if/for header as a header without its body. Output only replacement code, without context, explanations, or Markdown.

Generation proportions. Table 7 reports the stage that completes each generation step, separating Repair into Direct, Split, and Merge. Proportions sum to one within each dataset and model.

Table 7: Generation outcome proportions across datasets and models.

Model	MBPP+					HumanEval+					APPS					HumanEvalPack				
	Sample	Direct	Split	Merge	Bridge	Sample	Direct	Split	Merge	Bridge	Sample	Direct	Split	Merge	Bridge	Sample	Direct	Split	Merge	Bridge
Llama-3.1-8B-IT	0.671	0.284	0.015	0.030	0.000	0.682	0.277	0.017	0.022	0.002	0.781	0.177	0.003	0.027	0.012	0.791	0.127	0.020	0.004	0.058
OpenCoder-8B-IT	0.718	0.271	0.010	0.000	0.001	0.704	0.258	0.009	0.026	0.003	0.788	0.175	0.005	0.024	0.008	0.778	0.147	0.009	0.007	0.059
Qwen2.5-Coder-14B-IT	0.640	0.222	0.082	0.055	0.001	0.633	0.191	0.078	0.097	0.001	0.691	0.191	0.049	0.061	0.008	0.731	0.166	0.035	0.015	0.053

Sample completes 63.3% to 79.1% of steps, and Direct supplies most accepted repairs. Together they resolve 82.4% to 98.9% of steps without changing statement boundaries. Qwen uses Split and Merge more often, with combined shares of 13.7% on MBPP+ and 17.5% on HumanEval+, compared with

at most 4.5% for Llama and OpenCoder on these benchmarks. Bridge accounts for at most 1.2% on MBPP+, HumanEval+, and APPS, compared with 5.3% to 5.9% on HumanEvalPack.

G.4 Authenticated Bridge Construction

We choose a bridge α from Table 8 satisfying $m(c, \alpha, k) = 1$. Its quoted literal z contains a signature of the normalized preceding code, generated using k and a random nonce as in Section 3.4.

The value of the string literal z is preserved under identifier canonicalization. Changing its nonce and authentication tag can thus alter the semantic embedding and LSH output of the bridge statement while preserving AST node counts and program behavior. We seek $m(\alpha, s, k) = 1$ within a finite number of trials, retaining s without a bridge on failure (Algorithm 1).

Since $p_V = p_L \oplus p_I$, only four parity tuples are possible. Table 8 covers all four in each language, so either parity can be selected for each scope before the nonce is adjusted.

During detection, marked statements matching bridge templates are retained only if their authentication tags verify against the preceding code. Other marked statements remain ordinary carriers.

Table 8: Authenticated bridge templates across six languages.

Language	Bridge $\alpha(z)$	(p_V, p_L, p_I)	Language	Bridge $\alpha(z)$	(p_V, p_L, p_I)
Python	`(z,)`	$(0,0,0)$	Java	`assert z != null;`	$(0,1,1)$
	`(z, None)`	$(1,1,0)$		`assert (z != null);`	$(1,1,0)$
	`(z,)[0]`	$(1,0,1)$		`assert !(z == null);`	$(1,0,1)$
	`(z, None)[0]`	$(0,1,1)$		`assert (!(z == null));`	$(0,0,0)$
Rust	`let _ = z;`	$(0,0,0)$	JS	`void z;`	$(0,1,1)$
	`let _ = (z,);`	$(0,1,1)$		`void (z);`	$(1,1,0)$
	`let _ = (z);`	$(1,0,1)$		`void !z;`	$(0,0,0)$
	`let _ = ((z,));`	$(1,1,0)$		`void (!z);`	$(1,0,1)$
Go	`_ = z`	$(0,0,0)$	C++	`(void)z;`	$(1,1,0)$
	`_ = (z)`	$(1,0,1)$		`(void)(z);`	$(0,1,1)$
	`_ = z == ""`	$(1,1,0)$		`(void)sizeof(z);`	$(0,0,0)$
	`_ = (z == "")`	$(0,1,1)$		`static_cast<void>(z);`	$(1,0,1)$

A retained bridge's signature helps detect changes to the normalized preceding code, providing intermediate checks in long programs. Under the budget bound in Theorem 1, deleting all bridges preserves a detection score above τ provided the ordinary statement sequence is unchanged.

H Additional Results

H.1 Completion Length across Benchmarks

Figure 6 compares code token lengths after removing surrounding natural language, using each backbone's tokenizer as defined in Appendix D. APPS and HumanEvalPack generally yield longer outputs than MBPP+ and HumanEval+. COSTA produces code of comparable length on MBPP+ and HumanEval+ and is among the shortest on HumanEvalPack. On APPS, its mean code length is lowest with OpenCoder but highest with Qwen. Code length thus varies with both the benchmark and backbone, with no consistent increase under COSTA across the eight evaluated combinations. Together with Section 4.2, these results show that high AUROC can coexist with compact outputs. COSTA accumulates evidence over statements, so token length alone does not determine its score.

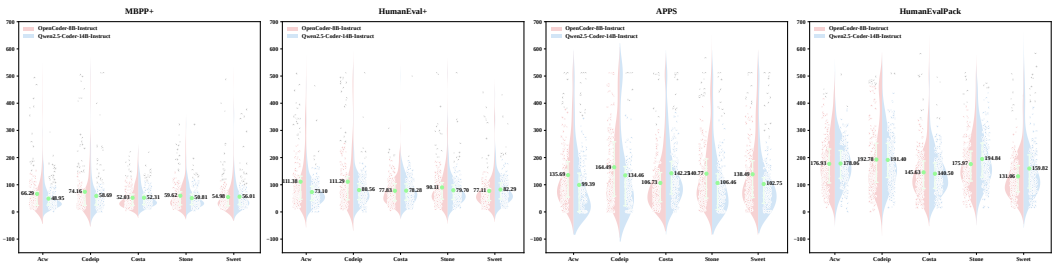


Figure 6: Code token lengths for five methods across four benchmarks and two backbones.

H.2 Code Quality with OpenCoder and Qwen

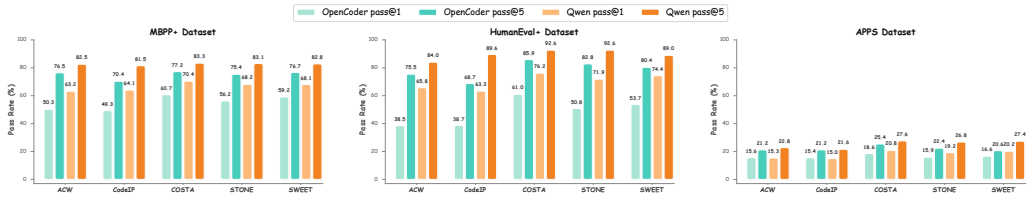


Figure 7: Pass@1 and pass@5 on MBPP+, HumanEval+, and APPS with OpenCoder and Qwen.

Figure 7 extends the comparison to OpenCoder, with Qwen retained for reference. COSTA achieves the highest pass@1 in all six settings and leads pass@5 in five, tying STONE at 92.6% on Qwen HumanEval+. Specifically, its OpenCoder pass@5 reaches 77.2%, 85.9%, and 25.4% on MBPP+, HumanEval+, and APPS, exceeding the strongest baseline by 0.5, 3.1, and 3.0 percentage points, respectively. This agreement with the main results shows that COSTA retains its advantage when multiple candidates are sampled, extending the evidence for functional correctness to OpenCoder.

H.3 Performance across Languages on HumanEvalPack

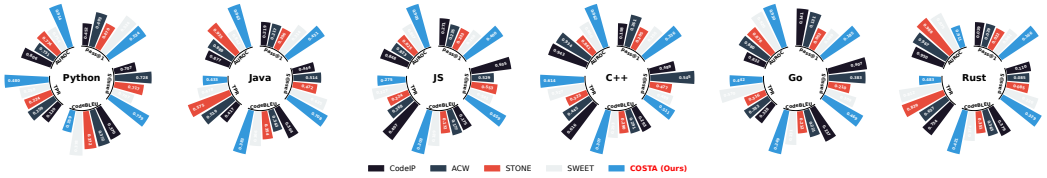


Figure 8: Detection and utility across six HumanEvalPack languages with Llama-3.1-8B-Instruct.

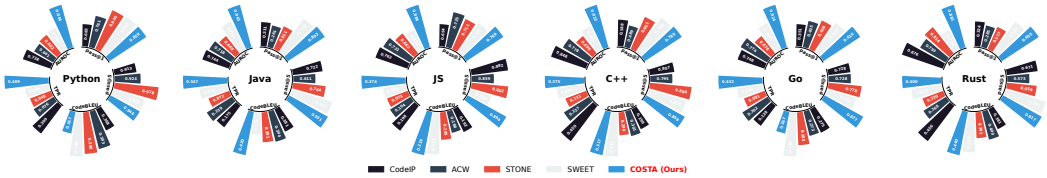


Figure 9: Detection and utility across six HumanEvalPack languages with Qwen2.5-Coder-14B-IT.

Figures 8 and 9 complement the OpenCoder comparison with Llama and Qwen. COSTA generally leads in pass@5 on both models. On Llama Rust, STONE attains 0.988 AUROC versus COSTA's 0.911, but COSTA raises pass@1 from 2.2% to 16.6% and pass@5 from 8.5% to 32.9%. On Qwen, COSTA also achieves higher AUROC across languages, with a mean relative improvement of 15.4% over the strongest baseline in each language. Its CodeBLEU on Go is lower than SWEET's (0.267 versus 0.310), whereas pass@5 is higher (82.7% versus 76.5%). These results illustrate why reference similarity should be considered alongside execution tests when assessing overall code quality.

H.4 Robustness under Code Transformations

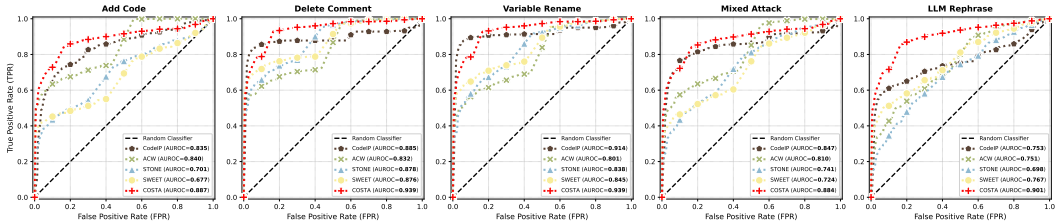


Figure 10: ROC curves under five code transformation attacks on APPS with Llama-3.1-8B-Instruct.

For Llama-3.1-8B-Instruct, as illustrated in Figure 10, COSTA achieves an AUROC of 0.939 under both comment deletion and variable renaming, compared with 0.887 under code insertion and 0.884 under the mixed attack. Code insertion can alter statement adjacency, while LLM rephrasing can modify expressions and statement boundaries. Despite these changes, COSTA retains an AUROC of 0.901 under rephrasing. Across the five attacks, its AUROC ranges from 0.884 to 0.939, with a mean of 0.910, outperforming all baselines in every setting.

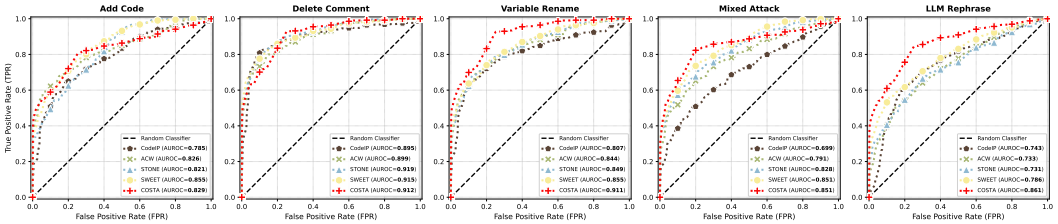
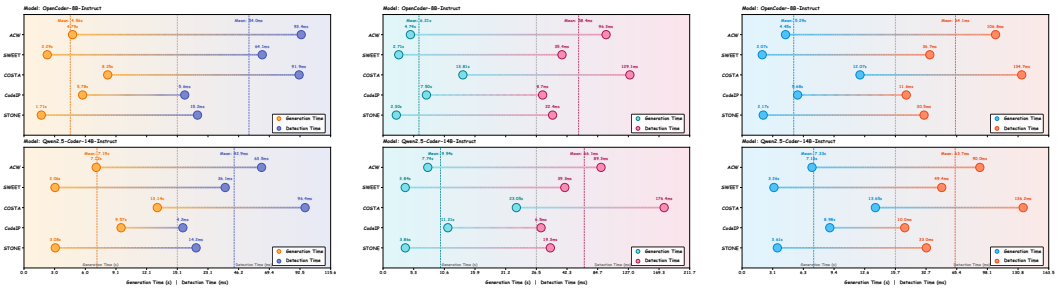


Figure 11: ROC curves under five code transformation attacks on APPS with OpenCoder-8B-IT.

H.5 Generation and Detection Latency across Benchmarks



(a) HumanEval+ (b) APPS (c) HumanEvalPack

Figure 12: Generation (s) and detection (ms) latency on OpenCoder-8B-IT and Qwen2.5-Coder-14B-IT.

Figure 12 extends the MBPP+ comparison to HumanEval+, APPS, and HumanEvalPack with OpenCoder-8B-IT and Qwen2.5-Coder-14B-IT. Averaging the six plotted settings equally, COSTA requires approximately 14.00 s per generation versus 8.12 s for CodeIP, a 72.4% increase. Each setting overhead ranges from 37.3% on Qwen HumanEval+ to 112.5% on OpenCoder HumanEvalPack. COSTA generation takes 1.13 to 1.67 times as long on Qwen as on OpenCoder. Detection requires 91.9 to 176.4 ms per program, averaging 127.5 ms across the six settings. The detector uses a fixed code encoder and statement ASTs without invoking the generation backbone, keeping detection below 0.18 s per program in every setting.

H.6 Hyperparameter Analysis

H.6.1 Configurations

Table 9 lists the five paired settings in Figure 5(b). At each shared temperature, SWEET varies entropy threshold and logit bias, whereas COSTA varies scope policy and rejection budget $\mathcal{B}$.

Table 9: Hyperparameter settings for COSTA and SWEET in the performance curves of Figure 5(b).

Watermarking	Configuration	Setting 1	Setting 2	Setting 3	Setting 4	Setting 5
SWEET	logit bias δ	2.0	4.0	5.0	6.0	8.0
	entropy threshold	0.3	0.4	0.5	0.0	0.3
COSTA	rejection budget $\mathcal{B}$	1	2	3	4	5
	scope mode (Fixed/Adaptive)	F/A	F/A	F/A	F/A	F/A

Unless noted otherwise, all ablations report means $\pm$ standard deviations over five runs; T@1%F is TPR at 1% FPR, and boldface marks the best entry for each combination of model, dataset, and metric. All entries follow the same evaluation protocol and aggregation procedure throughout.

H.6.2 Rejection Sampling Budget

To isolate sampling, we disable repair and the insertion of dead code and vary $\mathcal{B}$ from one to five. Raising $\mathcal{B}$ from one to three yields AUROC gains from 0.146 to 0.211 across all six settings in Table 10, with no monotonic pass@1 loss. Further gains range from 0.013 to 0.097, while Figure 13 shows rising encoding latency; we therefore use $\mathcal{B} = 3$ to balance detectability and generation cost.

H.6.3 Carrier Scope Policy

Table 11 compares a fixed scope over all AST nodes with a keyed adaptive scope. Fixed scope gives higher AUROC in five of six settings, but its mean advantage is only 0.01 (0.962 versus 0.952), the

Table 10: COSTA utility and detectability under five rejection budgets across three code benchmarks.

Model	MBPP+ Dataset				HumanEval+ Dataset				APPS Dataset			
+ Watermark	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑
Llama-3.1-8B-IT	-	-	-	-	-	-	-	-	-	-	-	-
+ COSTA ($\mathcal{B}=1$)	0.536 ± 0.079	0.720 ± 0.000	0.044 ± 0.030	0.514 ± 0.030	0.548 ± 0.078	**0.820 ± 0.000**	0.012 ± 0.018	0.491 ± 0.049	0.140 ± 0.020	**0.180 ± 0.000**	0.024 ± 0.026	0.533 ± 0.028
+ COSTA ($\mathcal{B}=2$)	0.564 ± 0.065	0.740 ± 0.000	0.052 ± 0.041	0.602 ± 0.017	**0.548 ± 0.046**	**0.820 ± 0.000**	0.024 ± 0.033	0.604 ± 0.050	0.136 ± 0.017	0.160 ± 0.000	0.048 ± 0.039	0.609 ± 0.044
+ COSTA ($\mathcal{B}=3$)	0.552 ± 0.036	0.740 ± 0.000	0.096 ± 0.050	0.690 ± 0.083	0.504 ± 0.038	0.780 ± 0.000	0.036 ± 0.038	0.637 ± 0.044	0.144 ± 0.009	0.160 ± 0.000	0.056 ± 0.059	0.689 ± 0.084
+ COSTA ($\mathcal{B}=4$)	**0.572 ± 0.030**	**0.780 ± 0.000**	0.080 ± 0.077	0.713 ± 0.045	0.492 ± 0.048	0.760 ± 0.000	0.032 ± 0.041	0.711 ± 0.038	0.144 ± 0.017	**0.180 ± 0.000**	0.020 ± 0.028	**0.721 ± 0.043**
+ COSTA ($\mathcal{B}=5$)	0.544 ± 0.048	0.720 ± 0.000	**0.152 ± 0.091**	**0.730 ± 0.041**	0.504 ± 0.038	0.800 ± 0.000	**0.044 ± 0.022**	**0.734 ± 0.053**	**0.148 ± 0.011**	**0.180 ± 0.000**	**0.060 ± 0.082**	0.702 ± 0.055
Qwen2.5-Coder-14B-IT	-	-	-	-	-	-	-	-	-	-	-	-
+ COSTA ($\mathcal{B}=1$)	**0.724 ± 0.079**	**0.880 ± 0.000**	0.040 ± 0.024	0.451 ± 0.049	0.800 ± 0.032	0.940 ± 0.000	0.004 ± 0.009	0.416 ± 0.030	0.200 ± 0.037	0.260 ± 0.000	0.016 ± 0.017	0.479 ± 0.027
+ COSTA ($\mathcal{B}=2$)	0.696 ± 0.038	0.840 ± 0.000	0.048 ± 0.023	0.564 ± 0.050	**0.860 ± 0.045**	**0.960 ± 0.000**	0.036 ± 0.059	0.541 ± 0.017	0.188 ± 0.023	0.240 ± 0.000	0.036 ± 0.026	0.647 ± 0.034
+ COSTA ($\mathcal{B}=3$)	0.704 ± 0.026	0.860 ± 0.000	0.080 ± 0.028	0.652 ± 0.024	0.844 ± 0.026	**0.960 ± 0.000**	0.036 ± 0.026	0.596 ± 0.020	0.196 ± 0.017	0.260 ± 0.000	**0.060 ± 0.055**	0.690 ± 0.047
+ COSTA ($\mathcal{B}=4$)	0.708 ± 0.023	0.840 ± 0.000	0.076 ± 0.038	0.678 ± 0.017	0.832 ± 0.058	**0.960 ± 0.000**	0.040 ± 0.047	0.645 ± 0.033	**0.212 ± 0.011**	0.260 ± 0.000	0.032 ± 0.030	0.696 ± 0.044
+ COSTA ($\mathcal{B}=5$)	0.716 ± 0.059	**0.880 ± 0.000**	**0.104 ± 0.054**	**0.689 ± 0.038**	0.792 ± 0.050	0.940 ± 0.000	**0.064 ± 0.048**	**0.653 ± 0.039**	0.200 ± 0.014	**0.280 ± 0.000**	0.056 ± 0.030	**0.749 ± 0.020**

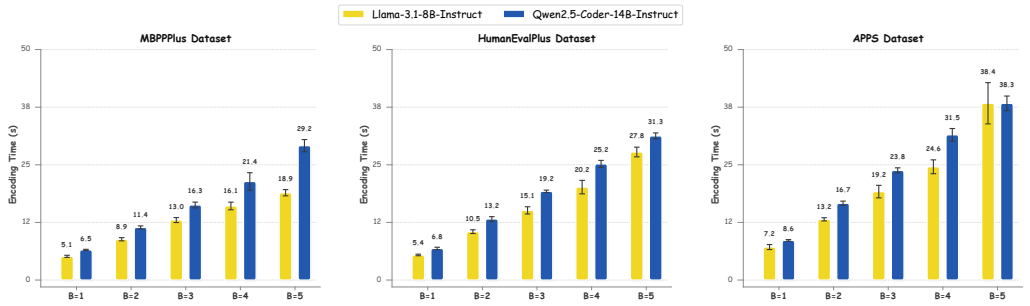


Figure 13: Encoding latency under five rejection sampling budgets across three code benchmarks.

Table 11: Fixed and adaptive COSTA carrier scope policies evaluated across three code benchmarks.

Model	MBPP+ Dataset				HumanEval+ Dataset				APPS Dataset			
+ Watermark	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑	Pass@1 ↑	Pass@5 ↑	T@1%F ↑	AUROC ↑
Llama-3.1-8B-IT	0.538 ± 0.036	0.730 ± 0.000	-	-	0.656 ± 0.043	0.840 ± 0.000	-	-	0.160 ± 0.016	0.190 ± 0.000	-	-
+ COSTA (Fixed)	**0.588 ± 0.046**	**0.740 ± 0.000**	0.348 ± 0.341	0.872 ± 0.016	**0.564 ± 0.055**	0.780 ± 0.000	0.424 ± 0.170	**0.980 ± 0.008**	**0.168 ± 0.027**	**0.220 ± 0.000**	**0.644 ± 0.128**	**0.960 ± 0.009**
+ COSTA (Adaptive)	0.548 ± 0.059	**0.740 ± 0.000**	**0.548 ± 0.218**	**0.876 ± 0.019**	**0.564 ± 0.077**	**0.820 ± 0.000**	**0.504 ± 0.195**	0.969 ± 0.021	0.156 ± 0.009	0.180 ± 0.000	0.468 ± 0.157	0.948 ± 0.011
Qwen2.5-Coder-14B-IT	0.764 ± 0.016	0.840 ± 0.000	-	-	0.874 ± 0.037	0.970 ± 0.000	-	-	0.192 ± 0.014	0.230 ± 0.000	-	-
+ COSTA (Fixed)	0.704 ± 0.038	0.860 ± 0.000	0.452 ± 0.334	**0.977 ± 0.014**	**0.848 ± 0.036**	**0.960 ± 0.000**	**0.820 ± 0.177**	**0.995 ± 0.006**	**0.216 ± 0.030**	**0.280 ± 0.000**	0.684 ± 0.253	**0.990 ± 0.009**
+ COSTA (Adaptive)	**0.724 ± 0.057**	**0.900 ± 0.000**	**0.592 ± 0.077**	0.971 ± 0.006	0.828 ± 0.061	0.940 ± 0.000	0.344 ± 0.151	0.968 ± 0.009	0.212 ± 0.018	0.240 ± 0.000	**0.704 ± 0.106**	0.983 ± 0.012

maximum gap is 0.027, and utility and T@1%F rankings vary across settings. We retain adaptive scope because it conceals each carrier's structural target while preserving comparable detection on clean data. This variability also prevents a fixed structural target from being exposed across carriers.

H.7 More Watermarking Baselines

Table 12: Ten additional text watermarking baselines across three code generation benchmarks, where ● denotes sentence-level text watermarking methods, and ◎ denotes token-level text watermarking methods.

Model	MBPP+ Dataset				HumanEval+ Dataset				APPS Dataset			
+ Watermark	Pass@1 ↑	T@1%F ↑	AUROC ↑	AvgLen	Pass@1 ↑	T@1%F ↑	AUROC ↑	AvgLen	Pass@1 ↑	T@1%F ↑	AUROC ↑	AvgLen
Llama-3.1-8B-IT	0.632 ± 0.046	-	-	40.85 ± 2.395	0.668 ± 0.023	-	-	74.81 ± 2.189	0.164 ± 0.017	-	-	145.2 ± 13.13
+ ◎ KGW (Kirchenbauer et al., 2023)	0.580 ± 0.063	0.044 ± 0.050	0.603 ± 0.028	50.74 ± 2.442	0.560 ± 0.024	0.096 ± 0.068	0.644 ± 0.037	97.10 ± 3.947	0.144 ± 0.017	0.244 ± 0.078	**0.778 ± 0.049**	161.0 ± 12.41
+ ◎ Unigram (Zhao et al., 2024)	0.596 ± 0.026	0.028 ± 0.011	0.627 ± 0.038	47.56 ± 2.804	0.600 ± 0.037	0.080 ± 0.040	0.651 ± 0.017	84.63 ± 1.357	0.144 ± 0.017	0.136 ± 0.098	0.754 ± 0.069	121.8 ± 8.022
+ ◎ DIP (Wu et al., 2023)	0.600 ± 0.024	0.024 ± 0.043	0.507 ± 0.026	41.38 ± 1.060	**0.660 ± 0.028**	0.012 ± 0.011	0.522 ± 0.015	80.62 ± 3.897	0.148 ± 0.011	0.068 ± 0.067	0.615 ± 0.044	150.2 ± 11.87
+ ◎ SynthID (Dathathri et al., 2024)	0.388 ± 0.046	0.184 ± 0.026	0.558 ± 0.031	53.49 ± 4.141	0.560 ± 0.037	0.104 ± 0.030	0.524 ± 0.040	81.02 ± 3.792	0.168 ± 0.011	0.056 ± 0.038	0.514 ± 0.039	148.7 ± 19.91
+ ◎ Unbiased (Hu et al., 2024)	0.600 ± 0.024	0.148 ± 0.039	0.406 ± 0.018	40.01 ± 1.893	0.620 ± 0.037	**0.332 ± 0.086**	0.414 ± 0.041	75.97 ± 5.816	**0.172 ± 0.011**	0.228 ± 0.033	0.742 ± 0.023	124.2 ± 12.84
+ ● SemStamp (Hou et al., 2024a)	0.608 ± 0.044	0.016 ± 0.017	0.526 ± 0.023	38.07 ± 2.982	0.648 ± 0.023	0.056 ± 0.033	0.549 ± 0.060	71.69 ± 3.311	0.160 ± 0.020	0.044 ± 0.036	0.621 ± 0.051	135.0 ± 7.694
+ ● KSemStamp (Hou et al., 2024b)	**0.612 ± 0.052**	0.020 ± 0.014	0.524 ± 0.025	36.29 ± 1.745	0.504 ± 0.059	0.012 ± 0.011	0.557 ± 0.022	53.75 ± 2.563	0.136 ± 0.017	0.084 ± 0.030	0.651 ± 0.027	76.58 ± 18.21
+ ● PMark (Huo et al., 2026a)	0.400 ± 0.057	0.000 ± 0.000	0.476 ± 0.035	32.02 ± 2.249	0.148 ± 0.048	0.008 ± 0.011	0.386 ± 0.064	37.06 ± 1.847	0.128 ± 0.011	0.024 ± 0.033	0.481 ± 0.065	50.51 ± 4.611
+ ● SAEMark (Yu et al., 2025)	0.440 ± 0.028	**0.240 ± 0.072**	**0.662 ± 0.041**	57.12 ± 5.493	0.176 ± 0.017	0.244 ± 0.054	**0.740 ± 0.041**	78.74 ± 14.34	0.128 ± 0.011	**0.296 ± 0.128**	0.746 ± 0.024	103.8 ± 14.27
+ ● SimMark (Dabiriaghdam & Wang, 2025)	0.576 ± 0.036	0.028 ± 0.023	0.548 ± 0.042	42.95 ± 1.969	0.636 ± 0.073	0.056 ± 0.038	0.589 ± 0.036	83.41 ± 7.933	0.144 ± 0.017	0.092 ± 0.061	0.643 ± 0.049	147.5 ± 9.756
Qwen2.5-Coder-14B-IT	0.788 ± 0.011	-	-	37.38 ± 0.584	0.896 ± 0.036	-	-	70.26 ± 1.829	0.204 ± 0.017	-	-	101.2 ± 8.291
+ ◎ KGW (Kirchenbauer et al., 2023)	0.732 ± 0.023	0.024 ± 0.033	0.539 ± 0.023	38.74 ± 2.149	0.800 ± 0.024	0.052 ± 0.039	**0.664 ± 0.020**	69.86 ± 7.137	0.148 ± 0.011	**0.340 ± 0.058**	**0.771 ± 0.025**	97.16 ± 10.89
+ ◎ Unigram (Zhao et al., 2024)	0.764 ± 0.026	0.024 ± 0.033	**0.575 ± 0.029**	38.39 ± 2.146	0.864 ± 0.026	0.040 ± 0.032	0.581 ± 0.019	72.46 ± 2.201	**0.208 ± 0.023**	0.052 ± 0.074	0.629 ± 0.049	119.4 ± 19.85
+ ◎ DIP (Wu et al., 2023)	0.768 ± 0.023	0.032 ± 0.039	0.533 ± 0.026	36.84 ± 1.293	0.864 ± 0.026	0.020 ± 0.020	0.533 ± 0.033	72.06 ± 4.372	0.196 ± 0.026	0.028 ± 0.030	0.508 ± 0.034	97.36 ± 12.50
+ ◎ SynthID (Dathathri et al., 2024)	0.620 ± 0.032	**0.320 ± 0.393**	0.467 ± 0.014	48.32 ± 4.181	0.704 ± 0.068	0.124 ± 0.200	0.470 ± 0.041	70.06 ± 2.610	0.196 ± 0.017	0.044 ± 0.017	0.493 ± 0.022	100.2 ± 16.92
+ ◎ Unbiased (Hu et al., 2024)	0.764 ± 0.048	0.020 ± 0.024	0.545 ± 0.010	37.34 ± 2.398	0.856 ± 0.030	0.048 ± 0.033	0.528 ± 0.026	68.57 ± 2.595	0.192 ± 0.011	0.060 ± 0.024	0.588 ± 0.027	112.0 ± 3.676
+ ● SemStamp (Hou et al., 2024a)	**0.788 ± 0.023**	0.020 ± 0.014	0.498 ± 0.061	36.50 ± 1.338	0.892 ± 0.027	0.028 ± 0.018	0.498 ± 0.030	69.44 ± 1.253	0.172 ± 0.023	0.064 ± 0.043	0.559 ± 0.040	125.1 ± 11.29
+ ● KSemStamp (Hou et al., 2024b)	0.768 ± 0.011	0.024 ± 0.026	0.508 ± 0.049	37.32 ± 1.180	**0.896 ± 0.026**	0.016 ± 0.022	0.547 ± 0.029	68.38 ± 3.301	0.188 ± 0.011	0.036 ± 0.054	0.581 ± 0.055	87.80 ± 11.34
+ ● PMark (Huo et al., 2026a)	0.160 ± 0.024	0.076 ± 0.050	0.507 ± 0.063	80.29 ± 10.97	0.168 ± 0.039	0.028 ± 0.041	0.487 ± 0.083	54.66 ± 8.103	0.120 ± 0.000	0.064 ± 0.033	0.523 ± 0.061	118.1 ± 14.34
+ ● SAEMark (Yu et al., 2025)	0.376 ± 0.048	0.124 ± 0.026	0.573 ± 0.032	77.14 ± 6.902	0.152 ± 0.054	**0.184 ± 0.068**	0.624 ± 0.037	97.99 ± 11.57	0.124 ± 0.009	0.272 ± 0.073	0.727 ± 0.035	112.4 ± 8.337
+ ● SimMark (Dabiriaghdam & Wang, 2025)	0.776 ± 0.026	0.012 ± 0.011	0.497 ± 0.041	37.78 ± 2.037	**0.896 ± 0.026**	0.004 ± 0.009	0.474 ± 0.034	70.14 ± 1.409	0.196 ± 0.009	0.044 ± 0.061	0.542 ± 0.037	107.9 ± 15.87

Table 12 compares five token-level and five sentence-level text watermarks using settings informed by MarkLLM (Pan et al., 2024) and the protocol in Section 4.1. Since these methods can favor prose over code, we remove natural language explanations and comments from watermarked and unwatermarked outputs to avoid inflating detectability in code. The sentence-level methods SemStamp, KSemStamp, and SimMark achieve AUROCs from 0.474 to 0.651. With few valid code candidates matching their targets, rejection sampling often exhausts its budget without embedding a strong signal. Although some token-level methods attain higher AUROC, pass@k can decline substantially. Both outcomes reflect the constraints that syntax and functional correctness impose on code generation. This difficulty in jointly retaining detectability and functionality motivates COSTA's statement-level design grounded in ASTs and tailored to the structure of code.

I LIMITATIONS

Our evaluation focuses on standalone code generation tasks, as testing complex software environments requires reproducible dependencies and builds, validation across files, and repeated integration tests at greater setup and execution cost. These resource demands pose a broader evaluation challenge for generative code watermarking. We believe that COSTA's integration of AST parity and semantic context at the statement level offers a promising foundation for practical code watermarking. Future work will extend its evaluation to more realistic software benchmarks.

J CASE STUDY

We present 12 examples across four datasets and each pair compares unwatermarked and COSTA outputs for the same model and task, with detection scores, token counts, and test outcomes. Prompts are summarized, while the complete code is retained with long lines wrapped for readability.

J.1 MBPP+

Case 1. Left bit rotation *Llama-3.1-8B-Instruct*

Prompt (summary)

Rotate a 32-bit integer n left by d bits.

Unwatermarked

```
def left_rotate(num, d):
    MAX_BITS = 32
    mask = (1 << d) - 1
    return ((num << d) & ((1 <<
    ↪ MAX_BITS) - 1)) | ((num & mask) >>
    ↪ (MAX_BITS - d))
```

$Z = -0.5774$ | 58 tokens | Test: Pass

COSTA

```
def left_rotate(n, d):
    if not isinstance(n, int) or not
    ↪ isinstance(d, int):
        raise TypeError("Both n and d
    ↪ must be integers.")
    if n < 0 or d < 0 or d > 31:
        raise ValueError("n must be
    ↪ between 0 and 2^32-1, and d must
    ↪ be between 0 and 31")
    mask = (1 << 32) - 1
    return ((n << d) | (n >> (32 - d)))
    ↪ & mask
```

$Z = 2.4495$ | 113 tokens | Test: Pass

Case 2. Jacobsthal numbers *Qwen2.5-Coder-14B-Instruct*

Prompt (summary)

Return J_n for $J_0 = 0$, $J_1 = 1$, and $J_n = J_{n-1} + 2J_{n-2}$.

Unwatermarked

```
def jacobsthal_num(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    j0 = 0
    j1 = 1
    for i in range(2, n + 1):
        jn = j1 + 2 * j0
        j0 = j1
        j1 = jn
    return j1
```

$Z = -0.6325$ | 90 tokens | Test: Pass

COSTA

```
def jacobsthal_num(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    prevprev = 0
    prev = 1
    cur = 0
    for i in range(2, n + 1):
        cur = sum((prev, prevprev * 2))
        prevprev, prev = prev, cur
    return cur
```

$Z = 3.3166$ | 104 tokens | Test: Pass

J.2 HUMANEVAL+

Case 3. Primes below a bound *Qwen2.5-Coder-14B-Instruct*

Prompt (summary)

Return all prime numbers less than the nonnegative integer n.

Unwatermarked

```python
def count_up_to(n):
    if n <= 2:
        return []

    def is_prime(num):
        if num < 2:
            return False
        for i in range(2, int(num ** 0.5) + 1):
            if num % i == 0:
                return False
        return True

    primes = []

    for i in range(2, n):
        if is_prime(i):
            primes.append(i)

    return primes
```

$Z = 1.3868$ | 100 tokens | Test: Pass

COSTA

```python
def count_up_to(n):
    def is_prime(num):
        if num <= 1:
            return False
        elif num <= 3:
            return True
        elif num % 2 == 0 or num % 3 == 0:
            return False
        i = 5
        while i * i <= num:
            if any(num % j == 0 for j in [i, i + 2]):
                return False
            i += 6
        return True
    primes = []
    for i in range(n):
        if is_prime(i):
            primes.extend([i])
    return primes
```

$Z = 3.7417$ | 163 tokens | Test: Pass

Case 4. Extension selection *Qwen2.5-Coder-14B-Instruct*

Prompt (summary)

Select the extension with the largest difference between uppercase and lowercase letter counts, choosing the first in a tie, and return `class.extension`.

Unwatermarked

```python
def Strongest_Extension(class_name, extensions):
    max_strength = float('-inf')
    strongest_extension = None
    for extension in extensions:
        CAP = sum(1 for c in extension if c.isupper())
        SM = sum(1 for c in extension if c.islower())
        strength = CAP - SM
        if strength > max_strength:
            max_strength = strength
            strongest_extension = extension
    return f"{class_name}.{strongest_extension}"
```

$Z = -0.6325$ | 97 tokens | Test: Pass

COSTA

```python
def Strongest_Extension(class_name, extensions):
    def calculate_strength(extension):
        cap = sum(1 for c in extension if c.isupper())
        sm = sum(1 for c in extension if c.islower())
        return cap - sm
    strongest_extension = max(extensions, key=calculate_strength)
    return f"{class_name}.{strongest_extension}"
```

$Z = 2.2361$ | 76 tokens | Test: Pass

J.3 APPS

Case 5. Factorials of Fibonacci numbers

Qwen2.5-Coder-14B-Instruct

Prompt (summary)

Return the sum of the factorials of the first n Fibonacci numbers, starting from $0, 1$, for $2 \leq n \leq 22$.

Unwatermarked

```python
import math
def sum_fib(n):
    def fibonacci(num):
        fibs = [0, 1]
        for i in range(2, num):
            fibs.append(fibs[-1] + fibs[-2])
        return fibs[:num]

    fib_sequence = fibonacci(n)
    factorial_sum = sum(math.factorial(num) for num in fib_sequence)
    return factorial_sum
```

$Z = -1.1339$ | 83 tokens | Test: Pass

COSTA

```python
def factorial(num):
    if num == 0 or num == 1:
        return_value = 1
    else:
        return_value = num * factorial(num-1)
    return return_value

def fibonacci(num):
    if num == 0:
        return 0
    elif num == 1:
        return 1
    else:
        return (fibonacci(num - 1) + fibonacci(num - 2))

def sum_fib(n):
    i = 0
    total_sum = 0
    while i <= n-1:
        fibnum = fibonacci(i)
        total_sum += factorial(fibnum)
        i+=1
    return(total_sum)
```

$Z = 3.7417$ | 147 tokens | Test: Pass

Case 6. Route to a table

OpenCoder-8B-Instruct

Prompt (summary)

Find a route from the cat to the table using U, D, L, and R. Return `NoCat` if the cat is outside the room, or `NoTable` if no table exists.

Unwatermarked

```python
def catPath(cat, room):
    if not cat or not room:
        return 'NoTable'

    rows, cols = len(room), len(room[0])

    if cat[0] >= rows or cat[1] >= cols:
        return 'NoCat'

    if room[cat[0]][cat[1]] == 1:
        return ''

    down = 'D' * (rows - cat[0] - 1)
    right = 'R' * (cols - cat[1] - 1)
    up = 'U' * cat[0]
    left = 'L' * cat[1]

    return up + left + down + right
```

$Z = -0.5774$ | 147 tokens | Test: Pass

COSTA

```python
def catPath(cat, room):
    if room[cat[0]][cat[1]] == 1:
        return ""
    elif cat[0] >= len(room) or cat[1] >= len(room[0]) or cat[0] < 0 or cat[1] < 0:
        return "NoCat"
    else:
        cat_row = cat[0]
        cat_column = cat[1]
        for row in room:
            if 1 in row and cat_row != room.index(row):
                cat_row = room.index(row)
                cat_column = row.index(1)

        route = ""
        while cat_row > 0:
            route += 'U' + ''
            cat_row -= 1

        while cat_column > 0:
            route += 'L' + ''
            cat_column -= 1

        while cat_row < len(room) - 1:
            route += 'D' + ''
            cat_row += 1

        while cat_column < len(room[0]) - 1:
            route += 'R' + ''
            cat_column += 1

        return route
```

$Z = 4.3589$ | 277 tokens | Test: Pass

J.4 HUMANEVALPACK

J.4.1 C++

Case 7. Space replacement *OpenCoder-8B-Instruct*

Prompt (summary)

Replace each space with an underscore, except runs of three or more spaces, which become a single hyphen.

Unwatermarked

```cpp
#include<stdio.h>
#include<algorithm>
#include<math.h>
#include<stdlib.h>
#include<iostream>
#include<string>
using namespace std;

string fix_spaces(string text){
    for(int i = 0; i < text.length(); i++){

        if(i + 2 < text.length() && text[i] == ' ' && text[i+1] == ' ' && text[i+2] == ' '){
            int spaces_found = 0;
            while(i + spaces_found < text.length() && text[i+spaces_found] == ' ') {
                spaces_found++;
            }
            if(spaces_found > 2){
                text.replace(i, spaces_found, "-");
            }
            i += 2;
        }

        else if(text[i] == ' '){
            text[i] = '_';
        }
    }
    return text;
}
```

$Z = -1.5076$ | 183 tokens | Test: Pass

COSTA

```cpp
#include<stdio.h>
using namespace std;
#include<math.h>
#include<stdlib.h>
#include<iostream>
#include<string>
#include<algorithm>

std::string fix_spaces(std::string str){
    std::replace( str.begin(), str.end(), ' ', '_');

    for(int i=0; i+2 < str.size(); i++){
        if(str[i] == '_' && str[i+1] == '_' && str[i+2] == '_'){
            while(str[i+1] == '_' && i+1 < str.size()) {
                str.erase(i+1,1);
            }
            str[i] = '-';
        }
    }
    return str;
}
```

$Z = 2.6458$ | 147 tokens | Test: Pass

J.4.2 Go

Case 8. Characters within words *Llama-3.1-8B-Instruct*

Prompt (summary)

Sort the characters of each word by ASCII value while preserving word order and spacing.

Unwatermarked

```go
package main
import (
    "sort"
    "strings"
)
func AntiShuffle(s string) string {
    words := strings.Fields(s)
    var result string

    for _, word := range words {
        chars := []rune(word)
        sort.Slice(chars, func(i, j int) bool { return chars[i] < chars[j] })
        result += string(chars) + " "
    }
    return strings.TrimSpace(result)
}

func main() {

}
```

$Z = 1.1339$ | 105 tokens | Test: Pass

COSTA

```go
import (
    "sort"
    "strings"
)
func AntiShuffle(s string) string {
    words := strings.Split(s, " ")
    var sortedWords []string
    for _, word := range words {
        sortedWord := ""
        runes := []rune(word)
        sort.Slice(runes, func(i, j int) bool { return runes[i] < runes[j]})
        for _, r := range runes { sortedWord += string(r) }
        sortedWords = append(sortedWords, sortedWord)
    }

    sortedString := strings.Join(sortedWords, " ")
    return sortedString
}
```

$Z = 2.1106$ | 140 tokens | Test: Pass

J.4.3 Python

Case 9. Triples divisible by three *OpenCoder-8B-Instruct*

Prompt (summary)

For $a_i = i^2 - i + 1$ with $1 \le i \le n$, count triples $i < j < k$ whose sum $a_i + a_j + a_k$ is divisible by three.

Unwatermarked

```python
def get_max_triples(n):
    arr = [(i * i - i + 1) for i in range(1, n + 1)]

    count = 0

    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            for k in range(j + 1, len(arr)):
                if (arr[i] + arr[j] + arr[k]) % 3 == 0:
                    count += 1

    return count
```

$Z = -0.9045$ | 158 tokens | Test: Pass

COSTA

```python
def get_max_triples(n):
    def get_sum_of_tripple(i, j, k):
        return i*i - i + 1 + j*j - j + 1 + k*k - k + 1
    triples_count = 0
    a = [i*i - i + 1 for i in range(1, n+1)]
    a.sort()
    for i in range(n):
        for j in range(i+1, n):
            for k in range(j+1, n):
                sum_ = a[i] + a[j] + a[k]
                if sum_%3 == 0:
                    triples_count += 1
    return triples_count
```

$Z = 3.3166$ | 150 tokens | Test: Pass

J.4.4 JAVA

Case 10. Primality of string length *Llama-3.1-8B-Instruct*

Prompt (summary)

Return whether the length of the input string is a prime number.

Unwatermarked

```java
import java.lang.*;
import java.util.*;

public class Solution {
    public boolean primeLength(String string) {
        return isPrime(string.length());
    }

    private boolean isPrime(int num) {

        if (num < 2) {
            return false;
        }


        if (num == 2 || num == 3) {
            return true;
        }


        if (num % 2 == 0 || num % 3 == 0) {
            return false;
        }


        int sqrtNum = (int) Math.sqrt(num);
        for (int i = 5; i <= sqrtNum; i += 6) {
            if (num % i == 0 || num % (i + 2) == 0) {
                return false;
            }
        }

        return true;
    }
}

import org.junit.Test;
import java.util.Arrays;
```

$Z = -1.1547$ | 189 tokens | Test: Pass

COSTA

```java
import java.util.*;
import java.lang.*;
import java.util.Arrays;

public class Solution {









    public boolean primeLength(String string) {

        if (string == null) {
            throw new NullPointerException("Input cannot be null");
        }


int length = (string.length());


        if (length < 2) {
return (false);
}


        if (length == 2) {
return true;
}


        for (int i = 2; i <= Math.sqrt(length); i++) {
            if ((length % i) == 0) {
return false;
            }
        }


return true;
    }
}
```

$Z = 3.3166$ | 159 tokens | Test: Pass

J.4.5 RUST

Case 11. Polynomial derivative *Qwen2.5-Coder-14B-Instruct*

Prompt (summary)

Return the derivative coefficients of a polynomial whose input coefficients are ordered by increasing degree.

Unwatermarked

```rust
fn derivative(xs: Vec<i32>) -> Vec<i32> {
    let mut result = Vec::with_capacity(xs.len() - 1);

    for (i, &coeff) in xs.iter().enumerate().skip(1) {
        result.push(i as i32 * coeff);
    }
    result
}
```

$Z = 0.0000$ | 128 tokens | Test: Pass

COSTA

```rust
fn derivative(xs: Vec<i32>) -> Vec<i32> {

    let mut result = Vec::new();
    for i in 1..xs.len() {

        result.push(xs[i] * i as i32);
    }

let derivative_result = result;
return derivative_result;
}
```

$Z = 2.2361$ | 131 tokens | Test: Pass

J.4.6 JAVASCRIPT

Case 12. Words with a given consonant count *Qwen2.5-Coder-14B-Instruct*

Prompt (summary)

Return the words in s that contain exactly n consonants, preserving their order.

Unwatermarked

```javascript
const selectWords = (s, n) => {

  const countConsonants = (word) => {
    const consonants = 'bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ';
    let count = 0;
    for (let char of word) {
      if (consonants.includes(char)) {
        count++;
      }
    }
    return count;
  };

  const words = s.split(' ');

  const result = words.filter(word => countConsonants(word) === n);

  return result;
};
```

$Z = -0.3333$ | 124 tokens | Test: Pass

COSTA

```javascript
const selectWords = (s, n) => {

const vowels = (new Set(['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']) );

const words = s.split(' ');

const result = [];


for (let word of words) {
let count = (0);

for (let char of word) {
    if (!(vowels.has(char))) {
        count += (1);
    }
}

if (count === n) {
(result.push(word));
}
}

return (result);
}
```

$Z = 3.3166$ | 138 tokens | Test: Pass