THE FUNCTION-STYLE TRADE-OFF: UNDERSTANDING LLM CODE GENERATION FROM A NOVEL PERSPECTIVE

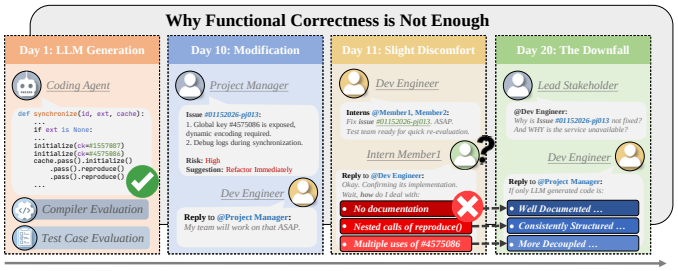
Anonymous authors
Paper under double-blind review

ABSTRACT

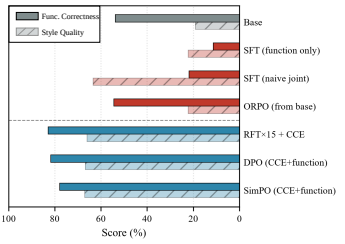
LLMs have made significant progress on code generation, yet their outputs often suffer from poor readability, inconsistent naming, missing documentation, and jumbled control flow. Current benchmarks overemphasize execution correctness while lacking scalable code-style metrics, hindering the development of style-aware code generation. To fill this gap, we construct a *fine-grained code-style evaluation benchmark* and propose a *graph-based framework using Augmented AST* that captures structural style features beyond the token level and enables *node-level attribution* of style decisions. Experiments reveal a *Function-Style Trade-off*: improving style naturalness degrades functional performance. We provide an *information-theoretic interpretation* of this phenomenon, framing fine-tuning as reallocation of representational capacity between natural language and code distributions. Extensive experiments demonstrate that the proposed structural style signal transforms the Function-Style Trade-off from an uncontrollable statistical fluctuation into an analyzable and optimizable engineering problem. This work underscores the necessity of incorporating software quality into LLM code evaluation.

1 INTRODUCTION

Despite reaching high functional pass rates on mainstream code generation benchmarks, the stylistic quality of LLM written code still remains underexplored (Wang et al., 2025; Tihanyi et al., 2026). Functional correctness can be verified via test cases, but *code style*, namely structural preferences in variable naming, nesting depth, documentation density, and exception handling, *lacks scalable and objective evaluation metrics*. This gap creates two practical issues: style review remains mostly manual and unscalable, while existing LLM evaluations focus almost exclusively on executability (Khojah et al., 2025), neglecting that even functionally correct code may still be structurally disorganized and difficult to maintain. As LLMs are increasingly adopted in high-stakes domains such as regulatory compliance and clinical decision support, the development of a comprehensive code-style evaluation framework is becoming urgently necessary.



(a) Why code style is as important as functional correctness.



(b) Fine-tuning performance with or without CCE.

Figure 1: An illustration of why code-style is as important as functional correctness alone in LLM code generation, with the development process of method `synchronization`.

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.

Evaluating code-style, however, is fundamentally difficult. Unlike functional correctness, style has no direct ground-truth label. Traditional statistical methods quantify code-style with hand-crafted features such as comment ratio or indentation regularity, missing high-level structural differences (Buse & Weimer, 2010; Szych & Schwerk, 2026). Other works compulsively defined code-style as human-alikeness and evaluate it merely through binary classification, not only it cannot serve alone as a tenable style evaluation metric, their models missing topological structure and interpretability (Xu et al., 2024; Park et al., 2025; Demirok & Kutlu, 2025). The consequences extend beyond engineering practice: without proper evaluation tools, the trade-off cannot be quantified.

Our experiments make this concrete: function-only SFT collapses HumanEval+ pass rate from 0.538 to 0.114 and lowers CCE-Full style quality, naive joint SFT improves CCE-Full style quality to 0.611 but drops pass rate to 0.218, preference optimization from the base model barely moves either axis. Theoretical analysis shows that this trade-off is intrinsic: functional and stylistic signals compete for finite representational capacity and are bound by Eq. equation 1, so improving one axis costs the other to compensate. What makes the trade-off far more damaging in practice is that no current fine-tuning pipeline uses a style quality signal aligned with the target distribution: style is either ignored during optimization or quantified with methods unaccountable and biased, which leads to severe consequences, i.e. poor readability, excessive nesting or even under-engineering.

To address such hazardous issue, we propose our interpretable code-style quality assessment framework. The framework includes a dual-difficulty evaluation benchmark (CCE-Full/Worse), an Augmented AST with the SlotGCN classifier for node-level attribution, and an information-theoretic account of the trade-off. On CCE-Full (55k+) it achieves **80.8% accuracy**. Crucially, the signal is actionable: with RFT, HumanEval+ pass rate rises from 53.8% to **82.9%** (+29.1 pp), while CCE-Full style quality improves from 0.605 to **0.637**. Experiments without code-style signal remain far from this joint improvement. Here we provide our contributions:

- A **code-style quality evaluation framework** that turns style into a measurable and attributable quantity, with node-level attribution via Augmented AST and SlotGCN.
- A **fine-grained dual-difficulty evaluation benchmark** (CCE), containing 55,669 industrial pairs (CCE-Full) and 10,542 educational/competition pairs (CCE-Worse) under weak functional equivalence. Our framework achieves **80.8% accuracy** on CCE-Full.
- An **information-theoretic formalization** of the Function-Style Trade-off, modeling fine-tuning as competitive allocation of finite representational capacity. It predicts complexity-dependent style detectability and explains the empirical trade-off.
- A **style-aware fine-tuning recipe** that turns the analyzer into a training signal, improving functional correctness and stylistic naturalness together on both HumanEval+ and CCE.

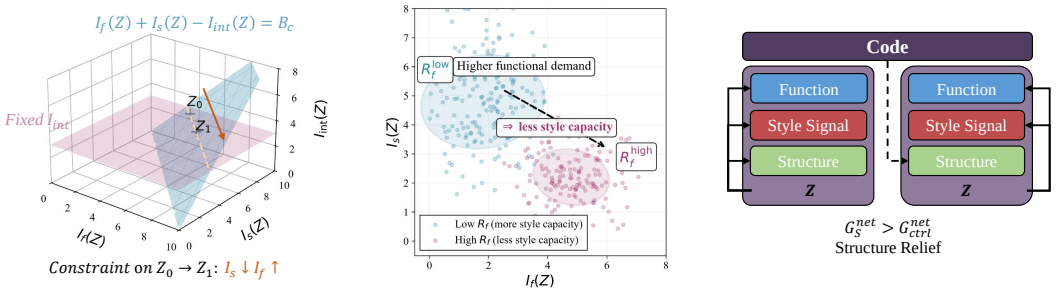
2 Related Works

Code-Style Evaluation. Since code-style is hard to detect, evaluation methods fall in two broad families. Statistical methods rely on developer ratings or hand-crafted features such as comment ratio and indentation regularity, but manual feature selection can be biased and inevitably misses high-level structural differences (Buse & Weimer, 2010; Szych & Schwerk, 2026; Velasco et al., 2025; Ouédraogo et al., 2026; Vitale et al., 2025). More recent methods treat style detection as binary classification between human or LLM written code (Xu et al., 2024; Park et al., 2025; Tahan, 2026; Licorish et al., 2025; Demirok & Kutlu, 2025; Kim et al., 2026), although this proxy is intuitional, it cannot serve as a standalone code-style metric. Moreover, many methods flatten ASTs into token sequences or data-flow graphs, losing topological structure and providing no attribution to specific structural features (Li et al., 2026; Berdichevsky et al., 2025; Dihan & Muhtasim, 2026). Recent work on style inconsistencies (Wang et al., 2025), the gap between correctness and quality (Umre et al., 2026) and attribution (Liu et al., 2025) further underscores the need for interpretable systematic evaluation. Our Augmented AST + SlotGCN explicitly encodes node-level attributes (type, content, depth, control-flow), enabling traceable attribution for code-style evaluation.

Limitations of Existing Benchmarks. Correctness benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and SWE-Bench (Jimenez et al., 2024) focus on test pass rates but neglects maintainability and other quality attributes. Follow-up work expanded language coverage (Athiwaratkun et al., 2023; Zheng et al., 2023; Nijkamp et al., 2023), task complexity (Zhang

et al., 2023; Du et al., 2024; Liu et al., 2024), or even LLM-based automation (Zhou et al., 2025), but remain confined to functional correctness (Zhong et al., 2026). Style-related benchmarks focus on rewrite detection (Park et al., 2025), source classification (Demirok & Kutlu, 2025; Guo et al., 2025), and authorship attribution (Patterson et al., 2026), with cross-scenario baselines (Demirok et al., 2025; Orel et al., 2025; Guo et al., 2026), yet most lack functional equivalence, task difficulty stratification, or is unattributable (Orel et al., 2026). Our CCE benchmark ensures fair and comprehensive code-style evaluation under these principles: weak functional equivalence via code decomposition and LLM regeneration, difficulty stratification via contrasting industrial (CCE-Full) and educational (CCE-Worse) sources, and interpretability via preserved AST structure for perturbation analysis. A detailed comparison is provided in Appendix L.

3 Theoretical Framework



(a) Fixed-level information reallocation.

(b) Functional demand compressing stylistic capacity.

(c) Explicit structural information relieving encoding burden.

Figure 2: Conceptual illustration of the Function-Style Trade-off framework.

We have observed a consistent phenomenon across fine-tuning, that the improvement on functional correctness tends to degrade stylistic naturalness, vice versa. Why should these two measurements that are in principle separable behave this way? To answer this, we propose an information-theoretic framework for the Function-Style Trade-off that organizes the phenomenon and yields testable constraints. Proofs and limitations are in Appendix F.

3.1 Setup and Assumptions

Let X be input, Z a learned representation, and Y_f, Y_s functional and stylistic attributes. Denote their mutual informations by $I_f(Z) = I(Z;Y_f)$ and $I_s(Z) = I(Z;Y_s)$, their interaction information by $I_{\text{int}}(Z) = I(Z;Y_f;Y_s)$, and the joint information by $J_{\mathcal{C}}(Z) = I(Z;Y_f,Y_s) = I_f(Z) + I_s(Z) - I_{\text{int}}(Z)$. Let $B_{\mathcal{C}}$ be the supremum of $J_{\mathcal{C}}(Z)$ over the representations $\mathcal{Z}_{\mathcal{C}}$ accessible under configuration $\mathcal{C}$. For task τ and target error ϵ, the required functional information is $R_f(\tau,\epsilon;\mathcal{C}) = \inf\{I(Z;Y_f) : \mathcal{L}_f(g(Z),\tau) \le \epsilon\}$.

We work under five assumptions, formalized in Appendix F.1: (A1) empirical information over a finite dataset; (A2) $B_{\mathcal{C}}$ as a reference level for $\mathcal{Z}_{\mathcal{C}}$ with ε-optimal representations allowed; (A3) standard metrics and style classifiers as meaningful proxies for I_f and I_s; (A4) matched effective levels, $J_{\mathcal{C}}(Z_0) = J_{\mathcal{C}}(Z_1) = B_{\mathcal{C}}$, relaxed to a δ-match; (A5) ε-functional sufficiency, $I_f(Z) \ge R_f(\tau,\epsilon;\mathcal{C})$ up to the same ε.

3.2 Reallocation, Stylistic Capacity, and Structural Benefit

Proposition 1 (Reallocation and Stylistic Capacity)**.** *Under the notation above and Assumptions A1–A5 where applicable, the following hold.*

(i) For $Z_0, Z_1 \in \mathcal{Z}_{\mathcal{C}}$ with $J_{\mathcal{C}}(Z_0) = J_{\mathcal{C}}(Z_1) = B_{\mathcal{C}}$ (or δ-matched in the relaxed version), define $\Delta I_f = I_f(Z_1) - I_f(Z_0)$, $\Delta I_s = I_s(Z_1) - I_s(Z_0)$, and $\Delta I_{\text{int}} = I_{\text{int}}(Z_1) - I_{\text{int}}(Z_0)$. Then

$$\Delta I_s = -\Delta I_f + \Delta I_{\text{int}}. \tag{1}$$

(ii) For any $Z \in \mathcal{Z}_{\mathcal{C}}$ satisfying ε-functional sufficiency,

$$I_s(Z) \leq B_{\mathcal{C}} - R_f(\tau, \epsilon; \mathcal{C}) + I_{\text{int}}(Z). \tag{2}$$

Complete proof and derivations in Appendices F.7 and F.10.

At a matched effective level, Proposition 1(i) implies that any observed trade-off is attributable to ΔI_{int} rather than to a change in total information: when $\Delta I_{\text{int}} = 0$ the identity reduces to $\Delta I_s = -\Delta I_f$, and otherwise ΔI_{int} can partially or fully offset ΔI_f. The δ-relaxed version is given in Corollary 1. The bound in Proposition 1(ii) is necessary but not sufficient: it does not guarantee that a reachable Z attains it. For any ε-functionally sufficient Z,

$$I_s(Z) \leq B_{\mathcal{C}} - R_f(\tau, \epsilon; \mathcal{C}) + I_{\text{int}}(Z).$$

Hence an increase in R_f tightens the upper bound on I_s, unless offset by I_{int}.

Now, let S be explicit structural information. By the chain rule applied to (S, Z),

$$G_S := I(Z; Y_s \mid Y_f) - I(Z; Y_s \mid S, Y_f) = I(S; Y_s \mid Y_f) - I(S; Y_s \mid Z, Y_f), \tag{3}$$

so $G_S \geq 0$ is an empirical condition, not a universal theorem. A positive G_S alone does not establish a structural inductive bias, because incorporating S may simply increase capacity. We therefore compare with a capacity-matched control S_{ctrl} sharing dimensionality, injection mechanism, trainable parameters, and computational cost but lacking the relevant structure (Appendix F.13), with $G_{\text{ctrl}} = I(Z; Y_s \mid Y_f) - I(Z; Y_s \mid S_{\text{ctrl}}, Y_f)$. With cost $\Omega(X) \geq 0$ and trade-off coefficient $\eta > 0$,

$$\begin{cases} G_S^{\text{net}} = G_S - \eta\Omega(S), \\ G_{\text{ctrl}}^{\text{net}} = G_{\text{ctrl}} - \eta\Omega(S_{\text{ctrl}}). \end{cases} \tag{4}$$

Proposition 2 (Genuine Structural Benefit)**.** *A structural augmentation S yields a genuine structural benefit if*

$$G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}. \tag{5}$$

Complete proof and derivations in Appendices F.12 and F.13.

A positive G_S alone does not imply a structural inductive bias, because incorporating S may increase capacity. We therefore compare against a capacity-matched control S_{ctrl} lacking the relevant structure (Appendix F.13), and write $G_{\text{ctrl}} = I(Z; Y_s \mid Y_f) - I(Z; Y_s \mid S_{\text{ctrl}}, Y_f)$ for its gain. With cost $\Omega(\cdot) \geq 0$ and trade-off coefficient $\eta > 0$, the net gains are $G_S^{\text{net}} = G_S - \eta\Omega(S)$ and $G_{\text{ctrl}}^{\text{net}} = G_{\text{ctrl}} - \eta\Omega(S_{\text{ctrl}})$. The criterion $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$ states that the style information attributable to S exceeds what the matched control achieves, after both are penalized by their costs.

It should be noted that, this criterion is optional rather than a derived theorem: it is conditional on the chosen cost Ω and trade-off coefficient η, which we discuss in Appendix F.17. Removing or perturbing S does not necessarily cause immediate degradation, because representations may reallocate to compensate. The criterion predicts that when $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$, removing S should cause greater degradation or require greater compensation than removing the matched control.

As for preference-based methods such as DPO, the style-specific information in preferences beyond functionality is $I(Y_p; Y_s \mid Y_f)$, where Y_p denotes the preference variable. DPO yields a distinct stylistic effect only when this quantity is substantial, if preferences carry no style signal beyond function, DPO is dominated by its SFT starting point. Details are in Appendix F.11.

3.3 Testable Predictions

The framework yields four empirically examinable implications, each tied to a specific experimental observable. Derivations are in Appendix F.

1. **P1 (Reallocation consistency).** At matched effective levels, observed trade-offs satisfy $\Delta I_s \approx -\Delta I_f + \Delta I_{\text{int}}$ (Proposition 1(i)). Tested via fine-tuning runs at matched effective levels.
2. **P2 (Task-complexity dependence).** Higher functional demand R_f tightens the bound on stylistic capacity (Proposition 1(ii)). Tested via the CCE-Full vs. CCE-Worse accuracy gap.

3. **P3 (Preference regime).** DPO yields a distinct style effect only when $I(Y_p; Y_s \mid Y_f)$ is substantial (Appendix F.11). Tested by comparing DPO on short-code (HumanEval+) versus long-code (CCE) preferences.
4. **P4 (Structure to capacity).** Structural augmentation is beneficial only when Proposition 2 holds. Tested via node-level perturbation and the capacity-matched control (Appendix F.13).

These information-theoretic quantities serve as a conceptual framework to organize our empirical investigation. Operationalization details are in Appendix F.15.

4 METHODOLOGY

4.1 PROBLEM FORMULATION

Let $\mathcal{Q}$ denote the space of programming tasks, where each $q \in \mathcal{Q}$ describes a program intent in natural language. For a given q, a human developer and an LLM produce implementations $c_{\text{human}} \sim P_{\text{human}}(\cdot \mid q)$ and $c_{\text{llm}} \sim P_{\text{llm}}(\cdot \mid q)$. Our goal is to quantify systematic differences in code style between the two populations under functional equivalence.

Since style $\mathcal{S}$ is not directly observable, we adopt a proxy approach: train a binary classifier $f : \mathcal{C} \to [0, 1]$ with source labels $y \in \{0, 1\}$ indicating human or LLM origin. For instance in CCE-Full, human code is sampled from high-quality, well-maintained open-source libraries, representing a *de facto* standard of professional style, thus the classification boundary serves as a proxy for stylistic naturalness $s(c) = 1 - f(c)$. A full justification is in Appendix M. We decompose our work into three sub-tasks: (1) **style classification**: building a classifier that distinguishes human from LLM code; (2) **style evaluation**: using the classifier to assess style naturalness of LLM generated code; (3) **style optimization**: exploring $s(c)$ as a reward signal when fine-tuning.

4.2 DATASET CONSTRUCTION: CCE-FULL AND CCE-WORSE

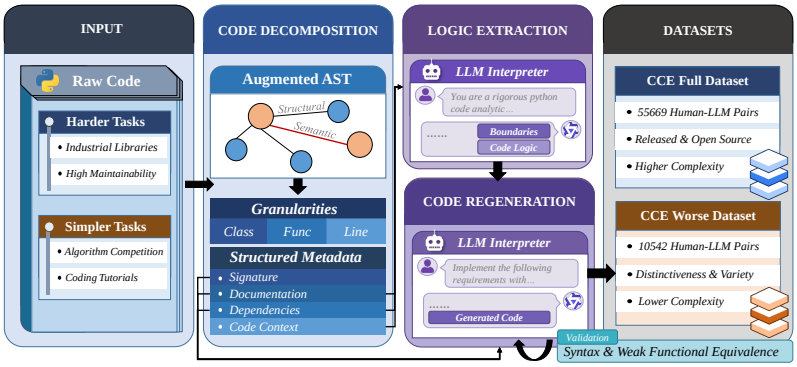


Figure 3: Generation pipeline of the CCE dataset.

CCE-Full draws from high-quality industrial code (open-source Python libraries on GitHub), while CCE-Worse draws from algorithm competition or code tutorials. To obtain functionally equivalent LLM code, we use a three-stage pipeline (Figure 3, details in Appendix C). Collected human code is first parsed into Class, Function, and Line granularities, from which we extract exposable interface information such as signatures, docstrings, and dependencies for later use. Each raw code snippet is then sent to an LLM for logic extraction, which converts it into a natural-language specification of what the code does rather than how it is implemented. Finally, the extracted logic and interface information are sent to a generation model, which produces code that conforms to the given specification while maintaining functional equivalence with the original implementation. The final datasets contain 55,669 and 10,542 code pairs, respectively.

Table 1 confirms our design: CCE-Worse has higher text similarity and lower JS divergence, indicating compressed style space in simpler scenarios.

Table 1: Statistical comparison between CCE-Full and CCE-Worse.

	Overall				Code Length		AST Nodes		Cyc. Comp.		CF Nodes	
Dataset	**Pairs**	**Text Sim.**	**JS Div.**	**ID Jaccard**	**Human**	**LLM**	**Human**	**LLM**	**Human**	**LLM**	**Human**	**LLM**
CCE-Full	55,669	0.62	0.34	0.67	1,462	861	2,377	2,488	3.23	2.65	2.91	2.10
CCE-Worse	10,542	0.71	0.21	0.74	1,336	1,106	1,598	1,621	2.17	2.05	2.01	3.31

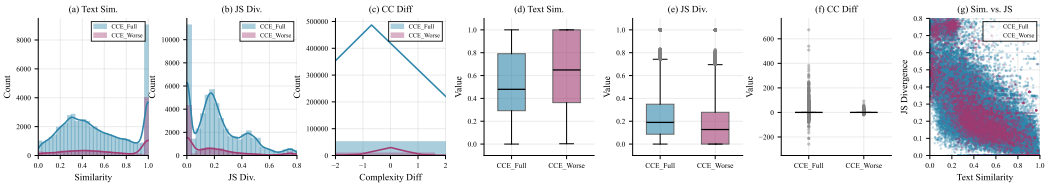


Figure 4: Statistical comparison between CCE-Full and CCE-Worse. Images show distributions of text similarity, JS divergence, and cyclomatic complexity difference, boxplots of the same metrics, scatter plot of text similarity vs JS divergence.

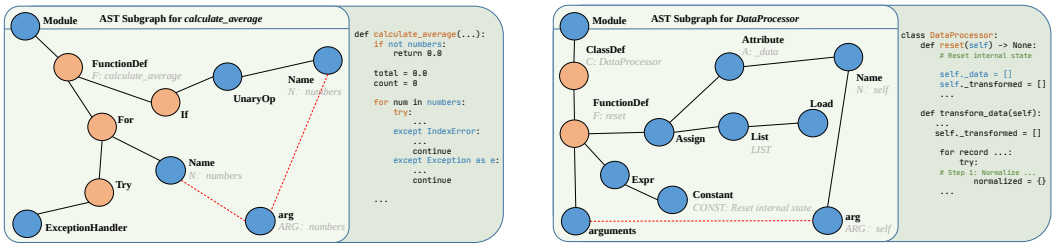
4.3 Evaluation Metrics

To quantify style quality beyond syntax correctness, we define a set of complementary metrics spanning surface, structural, and probabilistic dimensions. Table 2 summarizes these metrics and their roles. The metrics are then aggregated into a normalized **Style Quality Score** in $[0, 1]$, where each dimension is weighted according to task granularity. The weights prioritize the probabilistic style signal which reflects naturalness relative to human code, while penalizing syntactic failure, excessive complexity and more.

Table 2: Evaluation metrics for Code-Style Quality. Each metric captures a distinct dimension, from surface-level syntax to probabilistic style naturalness.

Category	**Metric**	**Abbrev.**	**Description**
Surface	Syntax rate	Syn.	Fraction of generated code that parses successfully
	Doc rate	Doc	Fraction of functions/classes with docstrings
	Doc length	Doc Len.	Average docstring character length
Structural	Cyclomatic complexity	C. Comp.	Average and maximum McCabe complexity
	Nesting depth	Depth	Average AST nesting depth
	Control-flow count	CF Count	Number of control-flow nodes (`if`, `for`, `while`, `try`)
	AST node count	AST Nodes	Total number of AST nodes
Probabilistic	Style probability	P(LLM)	Classifier output $P(\text{LLM} \mid c)$, lower means more human-alike
	Style entropy	Entr.	Shannon entropy of AST node-type distribution

4.4 Augmented AST



(a) `calculate_average` Method (b) `DataProcessor` Class

Figure 5: Augmented AST subgraphs at different scale. Orange nodes are control nodes. Black lines represent structural relations. Red lines are semantic edges.

Section 3 motivates encoding style signals structurally at the input level, where a genuine benefit requires $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$ (Eq. equation 5). For each AST node n, we define:

$$\mathbf{f}(n) = [\text{type_id}(n), \text{content_id}(n), \text{depth}(n), \text{is_cf}(n)] \tag{6}$$

encoding node type, textual content, nesting depth, and control-flow status. We construct two edge types: *structural* (AST parent-child) and *semantic* (variable/function references), yielding a heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}_{\text{struct}}, \mathcal{E}_{\text{sem}})$ that separates the two information channels.

4.5 Model Architecture

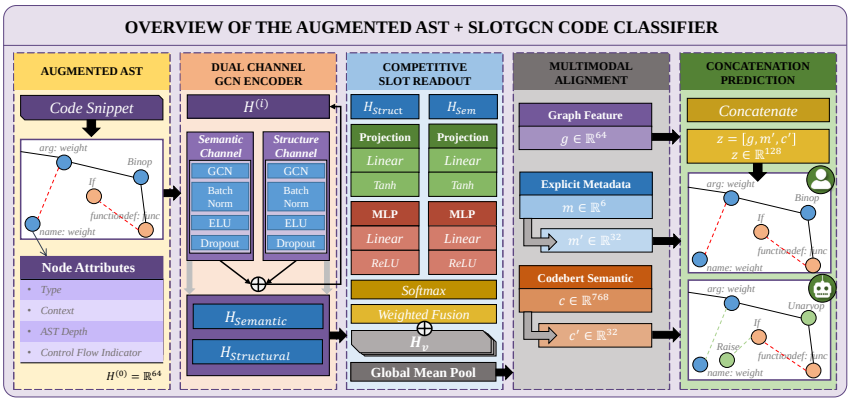


Figure 6: SlotGCN architecture, from code snippet to classification result.

The architecture (Figure 6) comprises four stages.

Node encoding. Type and content identifiers are embedded via lookup tables into two 16-dimensional vectors, concatenated with scalar depth and control-flow indicator into a 34-dimensional raw feature, and linearly projected to 64 dimensions:

$$\mathbf{h}_n^{(0)} = \text{ReLU}(\mathbf{W}_{\text{proj}} \cdot [\mathbf{e}_{\text{type}}; \mathbf{e}_{\text{content}}; \text{depth}; \text{is_cf}] + \mathbf{b}) \,.$$

Dual-channel propagation. Two stacked `SlotGCNLayer` modules process structural and semantic edges independently, each layer consisting of two parallel GCN convolutions with separate parameters, followed by BatchNorm and ELU, and a residual connection with dropout $p = 0.3$.

SlotReadout. For each channel, node-level importance scores are computed through a linear projection with Tanh followed by a shared two-layer MLP attention network ($64 \to 32 \to 1$):

$$\alpha_{\text{struct},n} = \text{MLP}_{\text{attn}}(\tanh(\mathbf{W}_{\text{proj,struct}} \cdot \mathbf{h}_{\text{struct},n})), \quad \alpha_{\text{sem},n} = \text{MLP}_{\text{attn}}(\tanh(\mathbf{W}_{\text{proj,sem}} \cdot \mathbf{h}_{\text{sem},n})).$$

Channel-level logits are obtained by summing these scores over all nodes and normalizing with Softmax, $\boldsymbol{\beta} = \text{Softmax}([\sum_n \alpha_{\text{struct},n}, \sum_n \alpha_{\text{sem},n}])$. The fused node representation is $\mathbf{h}_v = \beta_{\text{struct}}\mathbf{h}_{\text{struct}} + \beta_{\text{sem}}\mathbf{h}_{\text{sem}}$, which is then aggregated by global mean pooling into a 64-dimensional graph feature $\mathbf{g}$.

Multi-modal fusion. We augment $\mathbf{g}$ with a six-dimensional metadata vector $\mathbf{m}$ (docstring presence, control-flow complexity, type hint density, documentation ratio, normalized length, signature-call match) and a projected CodeBERT embedding $\mathbf{c}' \in \mathbb{R}^{32}$, forming a 128-dimensional fused representation $\mathbf{z} = [\mathbf{g}; \text{MLP}_{\text{meta}}(\mathbf{m}); \text{MLP}_{\text{codebert}}(\mathbf{c})]$. A two-layer MLP classifier ($128 \to 64 \to 1$) with ReLU and dropout $p = 0.5$ produces the final logit. Full details in Appendix E.

5 Experiments

5.1 Experiment Setups

We split both CCE-Full and CCE-Worse into training and validation sets at an 8:2 ratio, which is used to train a proxy style classifier, whose output probability involves computation of the **Style Quality Score**. We further conduct perturbation analysis and fine-tuning experiments to examine how the style signal behaves under structural edits and training. The experiments are broadly consistent with the four predictions from Section 3: P1 (reallocation consistency) is tested via fine-tuning runs at matched effective levels; P2 (task-complexity dependence) via the CCE-Full vs. CCE-Worse gap; P3 (preference regime) via DPO on short-code (HumanEval+) vs. long-code (CCE) preferences in Appendix J; and P4 (structure to capacity) via the perturbation analysis in Section 5.4.

5.2 Style Classifier: Selection and Generalization

We evaluate eight architectural configurations on both CCE-Full and CCE-Worse, results are presented in Table 3. Taking into account accuracy, efficiency, and interpretability, we select SlotGCN with A-AST as our default configuration. Our setup outperforms ChatGPT-Detector-RoBERTa by 28.22% on CCE-Full (Appendix G). On HumanEval+ and EvoEval, our classifier achieves 83.12% accuracy, confirming its robustness and reliance on structure-level signals (Appendix K).

Table 3: Full performance of eight configurations on CCE-Full and CCE-Worse. It should be noted that SlotGCN and SlotGAT methods incorporate metadata as well.

Method	Full Acc.	Full Loss	Full Time (s)	Worse Acc.	Worse Loss	Worse Time (s)	AST Info	AST Flattened
Plain AST	72.76%	0.5419	2073.70	65.32%	0.5958	466.59	✓	×
A-AST Only	73.74%	0.4892	2372.58	65.52%	0.5745	580.32	✓	×
CodeBERT Only	78.54%	0.4171	2187.65	69.47%	0.5325	**312.33**	×	×
A-AST with Metadata	75.80%	0.4698	3085.18	66.49%	0.5726	530.85	✓	×
CodeBERT with Metadata	80.15%	0.3819	2632.14	70.27%	0.5180	443.38	×	×
A-AST with CodeBERT	78.89%	0.4098	2285.73	69.71%	0.5143	724.30	✓	×
SlotGCN with A-AST (Ours)	**79.41%**	**0.4133**	**1813.20**	**70.32%**	**0.5273**	**411.47**	✓	✓
SlotGAT with A-AST (Ours)	**80.78%**	**0.3708**	**3005.40**	**71.77%**	**0.4847**	**869.54**	✓	✓

All configurations achieve substantially higher accuracy on CCE-Full than on CCE-Worse, emphasizing that *style classification performance is indeed correlated with task complexity*, a direct empirical validation of Eq. equation 2.

5.3 Cross-Model Style Evaluation

Full evaluation results on CCE-Full and CCE-Worse with **Style Quality Score** are presented in Table 4. From this experiment we find:

Model capability boundaries. Commercial models lead in syntax rate on Full, with Qwen 3.5 Plus reaching 98.1 however, GLM 4.5 Air drops to 28.7% on Full but achieves 99.5% on Worse.

Function-Style Trade-off. DeepSeek V4 Pro has the lowest style probability (0.568±0.201) but only 78.4% syntax rate, while Qwen 3.5 Plus achieves the highest syntax rate (98.1%) yet also the highest style probability (0.759±0.251), consistent with Eq. equation 1.

Quality ranking. Qwen 3.5 Plus (Full: 0.617) and DeepSeek V4 Flash (Worse: 0.635) lead. Full granularity breakdowns are in Appendix G.

Table 4: Cross-model performance on CCE-Full and CCE-Worse, metrics defined in Section 4.3.

	CCE-Full						CCE-Worse					
Model	**Syn.**	**P(LLM)**	**Entr.**	**C. Comp.**	**Doc**	**Q.**	**Syn.**	**P(LLM)**	**Entr.**	**C. Comp.**	**Doc**	**Q.**
DeepSeek V4 Pro	78.4%	**56.8%±20.1%**	1.637	1.167	9.0%	0.571	89.6%	71.2%±21.2%	1.807	1.219	2.0%	0.602
DeepSeek V4 Flash	84.8%	65.8%±18.8%	1.789	1.441	9.5%	0.593	95.3%	69.4%±22.2%	1.954	1.547	**2.5%**	**0.635**
Qwen 3.5 Plus	**98.1%**	75.9%±25.1%	2.078	**2.385**	13.2%	**0.617**	98.9%	68.4%±30.8%	2.036	1.800	1.4%	0.618
GLM 5.2	97.6%	81.2%±23.3%	**2.079**	1.874	**16.1%**	0.614	**99.7%**	74.4%±27.4%	2.121	1.844	0.8%	0.614
GLM 4.5 Air	28.7%	59.1%±17.8%	0.619	0.730	4.7%	0.211	99.5%	88.4%±19.0%	**2.144**	**1.896**	1.4%	0.583
Qwen-2.5-Coder-7B	91.1%	86.3%±20.1%	2.001	1.967	2.4%	0.577	98.9%	**65.0%±29.1%**	2.076	1.633	0.3%	0.622
DeepSeek-Coder-16B	74.7%	68.3%±23.4%	1.655	1.428	9.0%	0.529	89.5%	88.0%±19.7%	1.923	1.478	0.2%	0.585

5.4 Evaluation Analysis and Perturbation

Feature analysis (Appendix H) reveal that LLM docstrings are ∼10× shorter than human-written ones and `pass_count` is 6.98× higher. We further perform node-level perturbation experiments on a short snippet (`calculate_average`, 104 nodes) and a longer one (`DataProcessor`, 1,657 nodes). Results see Table 5.

On a sample of 100 CCE-Full snippets (Appendix H.2.4), `depth_-1` yields the largest average absolute change (0.0309), exceeding `depth_+1` (0.0266) and `is_cf_flip` (0.0235). Single-node depth adjustment (+3.44 pp) more than doubles the effect of global adjustment. Removing control-flow markers is nearly ineffective (+0.24 pp), consistent with $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$ (Eq. equation 5). The `logger→print` replacement triggers an 11.65 pp flip, the strongest single effect we observe. Full comparison logs in Appendix I.

Table 5: Perturbation results on shorter code (left) and longer code (right).

Shorter Code			Longer Code			
Operation	**P(LLM)**	**Diff**	**Operation**	**P(LLM)**	**Diff**	**Judgment**
Baseline	82.46%	—	Baseline	43.55%	—	Human
Depth +1 (all nodes)	84.02%	+1.56%	Depth +1 (all nodes)	48.56%	+5.01%	Human
Single-node depth adj.	85.90%	+3.44%	Remove all CF markers	43.79%	+0.24%	Human
—	—	—	Replace `logger` with `print`	**55.20%**	**+11.65%**	Human→**LLM**

5.5 Style Fine-Tuning

We study whether the style signal can be turned into an actionable training objective. All experiments use Qwen2.5-Coder-1.5B-Instruct as the base model with LoRA ($r = 64$). We report HumanEval+ pass rate (strong functional), CCE-Full syntax rate (weak functional on long code), and style probability $\texttt{prob_llm} = P(\text{LLM} \mid c)$, where *lower is more human-like*. Full configurations, results and analysis are in Appendix J.

Five fine-tuning strategies are compared. SFT trains on canonical solutions, either function-only or mixed with CCE. RFT sample selection are based on HumanEval+, optionally mixed with CCE, both using our style quality as signal. DPO optimizes pairwise preferences built from human or LLM style pairs and passing or failing functional pairs.**ORPO** and **SimPO** are reference-free preference-optimization variants.

Table 6: Representative fine-tuning results on HumanEval+ and CCE. "Corr." and "Style" indicate whether correctness (unit-test) and style-quality signals are used in training data or data selection. Arrows compare each method to its own training start (Base for SFT, E7 for DPO/SimPO).

ID	Method	Corr.	Style	HE Pass	HE P(LLM)↓	CCE-F Syn.	CCE-F P(LLM)↓	CCE-F Q.	Both↑
E0	Base	—	—	53.8%	37.6%	**95.3%**	80.9%	0.605	—
E1	SFT (function only)	×	×	11.4%↓	24.6%↓	81.3%↓	77.8%↓	0.552↓	×
E2	SFT (naive joint)	×	✓	21.8%↓	41.1%↑	83.0%↓	36.6%↓	0.611↑	×
E4	RFT (×8) + CCE	✓	✓	65.9%↑	31.8%↓	84.7%↓	35.0%↓	0.631↑	✓
E6	RFT (×15), no CCE in RFT selection	✓	✓	81.1%↑	19.0%↓	93.7%↓	67.0%↓	**0.643**↑	✓
E7	**RFT (×15) + CCE**	✓	✓	**82.9%**↑	17.4%↓	86.0%↓	34.0%↓	0.637↑	✓
E9	DPO (CCE+function)	✓	✓	81.7%↓	14.9%↓	83.3%↓	33.3%↓	0.621↓	×
E10	ORPO (CCE+function, from base)	✓	✓	54.4%↑	35.8%↓	**95.3%**	77.8%↓	0.617↑	×
E11	SimPO (CCE+function)	✓	✓	77.8%↓	**13.2%**↓	83.0%↓	**33.0%**↓	0.619↓	×
E12	DPO (function only, no CCE)	✓	✓	83.2%↑	17.6%↑	84.7%↓	34.0%	0.631↓	×

From Table 6 we find, plain SFT exhibits the trade-off in the most severe way possible (E1–E3). RFT removes the trade-off only where the style signal is computed: without CCE, E6 improves short-code functionality and style but leaves long-code style nearly unchanged. Incorporating CCE during fine-tuning enables functional correctness and stylistic quality to jointly improve (E7), the slight CCE-Full syntax drop is attributable to truncation of longer outputs due to limited experiment conditions (Appendix J.5). Preference optimization also amplifies style only with CCE: E9 or E11 improve on E7, whereas E10 and E12 stay at their start.

These results show that a reliable code-style analyzer is not only an evaluation tool but a usable training signal: once style is quantified on the relevant task distribution, it can be optimized jointly with functional correctness rather than traded against it.

6 Conclusion

Code-style quality remains an underexplored dimension of AI-generated software quality. We have presented an interpretable framework for code-style quality assessment, built upon a dual-difficulty benchmark (CCE), an Augmented AST representation, and a SlotGCN classifier, together with an information-theoretic account of the Function-Style trade-off. Experiments demonstrate the existence of this trade-off, showing how it degrades fine-tuning in the absence of a code-style signal, and how, under the signal provided by our benchmark, stylistic quality and functional correctness improve jointly during fine-tuning. These results indicate that the trade-off is not an inherent capacity limit but rather a consequence of optimizing without an aligned style signal.

Future work will extend the framework to more programming languages and to ensemble-based style detection. We hope this work encourages further investment in code-style quality as an important research target.

REFERENCES

Ben Athiwaratkun, Sanjay Krishna Gouda, Ziyu Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shi Wang, Qi Sun, Mingyue Shang, Siva Krishna Gonugondla, Han Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Saurabh Jain, Robert Giaquinto, Han Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Shanu Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models, 2021.

Ruslan Berdichevsky, Shai Nahum-Gefen, and Elad Ben-Zaken. Salsa: Single-pass autoregressive llm structured classification, 2025.

Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70.

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, 2021.

Basak Demirok, Mucahid Kutlu, and Selin Mergen. MultiAIGCD: A comprehensive dataset for AI generated code detection covering multiple languages, models, prompts, and scenarios, 2025.

Burak Demirok and Mucahid Kutlu. Aigcodeset: A new annotated dataset for ai generated code detection, 2025.

Mahir Labib Dihan and Abir Muhtasim. Llmsniffer: Detecting llm-generated code via graphcodebert and supervised contrastive learning, 2026.

Xueying Du, Ming Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 5021–5041. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.300.

Haoyu Guo, Shuang Cheng, Kai Zhang, Gang Shen, and Xinyu Zhang. Codemirage: A multilingual benchmark for detecting ai-generated and paraphrased source code from production-level llms, 2025.

Junjie Guo, Zekun Yang, Ming Sun, Hongyi Wang, Jian Lu, and Yuxing Zhang. Code fingerprints: Disentangled attribution of llm-generated code, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024. doi: 10.48550/arXiv.2310.06770.

Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. The impact of prompt programming on function-level code generation. *IEEE Transactions on Software Engineering*, 51(8):2381–2395, 2025. doi: 10.1109/TSE.2025.3587794.

Taekhyun Kim, Hyejoon Jang, Suyoung Kim, Zhenming Jin, Suin Kim, and JinYeong Bak. LiMO: Line-level machine-generated code detection for diverse programming languages. In *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26)*, SAC '26, pp. 917–925, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722943. doi: 10.1145/3748522.3779718. URL https://doi.org/10.1145/3748522.3779718.

Zhengdao Li, Xiuwei Shang, Zhenkan Fu, Shikai Guo, Weiming Zhang, NengHai Yu, and Kejiang Chen. Dualcodedetect: Zero-shot llm-generated code detection via dual-channel perturbation. *Proc. ACM Softw. Eng.*, 3(FSE), 2026. doi: 10.1145/3808166.

Sherlock A Licorish, Ansh Bajpai, Chetan Arora, Fanyu Wang, and Kla Tantithamthavorn. Comparing human and llm generated code: The jury is still out!, 2025.

Rundong Liu, André Frade, Amal Vaidya, Maxime Labonne, Marcus Kaiser, Bismayan Chakrabarti, Jonathan Budd, John Buford, Xiaomin Lin, and Sean Moran. Codequest: Evaluation and improvement of code quality using llms. In *2025 IEEE 36th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 123–130, 2025. doi: 10.1109/ISSREW67781.2025.00060.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, pp. 47832–47850, 2024. doi: 10.48550/arXiv.2306.03091.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations (ICLR)*, 2023. doi: 10.48550/arXiv.2203.13474.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. `Droid`: A resource suite for AI-generated code detection. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 31263–31289, 2025. doi: 10.18653/v1/2025.emnlp-main.1593.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. SemEval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. In Ekaterina Kochmar, Debanjan Ghosh, Kai North, Mamoru Komachi, and Marcos Zampieri (eds.), *Proceedings of the 20th International Workshop on Semantic Evaluation (2026)*, pp. 3640–3658, 2026. doi: 10.18653/v1/2026.semeval-1.445.

Wendkûuni C. Ouédraogo, Yinghua Li, Xueqi Dang, Paweł Borsukiewicz, Xin Zhou, Anıl Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. From metrics to models: Large language models as human-aligned code readability evaluators. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26)*, pp. 409–411, New York, NY, USA, 2026. Association for Computing Machinery. doi: 10.1145/3774748.3795656. URL https://doi.org/10.1145/3774748.3795656.

Shinwoo Park, Hyundong Jin, Jeong-Won Cha, and Yo-Sub Han. Detection of llm-paraphrased code and identification of the responsible llm using coding style features. *Engineering Applications of Artificial Intelligence*, 162:112454, 2025. doi: 10.1016/j.engappai.2025.112454.

Luke Patterson, Li Wang, and Adam Faulkner. Hybridcodeauthorship: A benchmark dataset for line-level code authorship detection. In *Proceedings of the 2026 International Conference on Language Resources and Evaluation (LREC)*, pp. 1520–1532, 2026. doi: 10.63317/4edsbxrqe8na.

Joanna Szych and Anne Schwerk. Evaluating LLM-generated code: A benchmark and developer study. In *Proceedings of the 2026 International Conference on Evaluation and Assessment in Software Engineering (EASE '26)*, Glasgow, United Kingdom, June 2026. ACM. ISBN 979-8-4007-2348-3. doi: 10.1145/3816483.3816621. URL https://doi.org/10.1145/3816483.3816621.

Marzieh Hajizadeh Tahan. A hybrid approach for machine-generated code detection based on statistical, authorship, and semantic feature extraction. *Inf. Softw. Technol.*, 195, 2026. doi: 10.1016/j.infsof.2026.108102.

Norbert Tihanyi, Bilel Cherif, Richard A. Dubniczky, Mohamed Amine Ferrag, Lajos Balkanyi, and Tamas Bisztray. The hidden dna of llm-generated javascript: Structural patterns enable high-accuracy authorship attribution. In *Proceedings of the 3rd International Workshop on Large Language Models For Code*, LLM4Code '26, pp. 42–50, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724121. doi: 10.1145/3786181.3788707. URL https://doi.org/10.1145/3786181.3788707.

Jayesh Umre, Ashish Singh Parihar, and Atul Gupta. From correctness to code quality: Formalizing software engineering metrics for evaluating general llms. In *Proceedings of the 19th Innovations in Software Engineering Conference*, ISEC '26, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400725036. doi: 10.1145/3796563.3796609. URL https://doi.org/10.1145/3796563.3796609.

Alejandro Velasco, Diego Rodriguez-Cardenas, Lamia R. Alif, Daniel N. Palacio, and Denys Poshyvanyk. How propense are large language models at producing code smells? a benchmarking study, 2025.

Antonio Vitale, Emanuela Guglielmi, Rocco Oliveto, and Simone Scalabrino. Personalized code readability assessment: Are we there yet? In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pp. 01–11. IEEE Press, 2025. doi: 10.1109/ICPC66645.2025.00019. URL https://doi.org/10.1109/ICPC66645.2025.00019.

Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. Beyond functional correctness: Investigating coding style inconsistencies in large language models. In *Proceedings of the 33rd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2025. doi: 10.48550/arXiv.2407.00456.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2024. doi: 10.48550/arXiv.2411.04704.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.151.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pp. 5673–5684, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701030. doi: 10.1145/3580305.3599790. URL https://doi.org/10.1145/3580305.3599790.

Ming Zhong, Xiang Zhou, Ting-Yun Chang, Qingze Wang, Nan Xu, Xiance Si, Dan Garrette, Shyam Upadhyay, Jeremiah Liu, Jiawei Han, Benoit Schillings, and Jiao Sun. SWE-IF: Aligning code evaluation with human preference. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2510.07315.

Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luís F. Gomes, Guang Yang, Kui Liu, Xin Xia, and David Lo. SE-Jury: An LLM-as-ensemble-judge metric for narrowing the gap with human evaluation in SE. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2606–2618. IEEE, 2025. doi: 10.1109/ASE63991.2025.00214.

A USE OF LLMS

The use of LLMs was restricted to aiding or polishing of writing, and the help of extracting code logic and generating synthetic datasets as described in Section 4.2, since our main focus of research is inevitably, bound with the topic of LLM code generation. Human authors performed theoretical work and conducted all experiments, and take full responsibility for any possible false-claims or misinformation that might unfortunately appear in this work.

B LIMITATIONS

B.1 VALIDATED ONLY ON PYTHON

Our classifier, benchmark, and fine-tuning experiments are all designed for Python. Code style varies across languages (e.g., Python's indentation conventions vs. Java's brace-style formatting), and although the Augmented AST representation is language-agnostic, the node-type vocabulary, control-flow definitions, and style-feature semantics must be re-adapted for each target language. The dataset generation and evaluation pipeline, by contrast, carries over unchanged.

B.2 HEURISTIC NATURE OF THE QUALITY SCORE

The quality score aggregates syntax rate, cyclomatic complexity, nesting depth, documentation rate, code entropy, and style probabilities, with weights assigned by task level. These choices are based on engineering experience and domain knowledge rather than formal derivation, and syntax rate as a component may couple with the style signal. Future work should calibrate the weights through ablation studies and user studies.

B.3 FUNCTIONAL CORRECTNESS ON INDUSTRIAL CODE

Industrial code in CCE typically lacks standardized test suites, so functional equivalence in CCE is defined at the task level—human and LLM code address the same natural-language task—rather than at the test-case level. Consequently, the CCE syntax rate measures whether code passes Python parsing, not whether it is functionally correct. This is an inherent difficulty of evaluating industrial code, and it limits how strictly our quality score can be interpreted as a functional signal on CCE.

B.4 THE STYLE SIGNAL OPERATES THROUGH DATA SELECTION IN DPO

In our DPO experiments (E8, E9, E12), the style signal enters through the construction of preference pairs rather than through a differentiable objective: style preference pairs are built by pairing human-written and LLM-generated code on the same CCE task, and the resulting pairs are then used as standard DPO training data. This works when the two populations differ systematically in style, but it does not optimize style directly, and the quality of the resulting signal depends on the pairing strategy. Whether a differentiable style objective would achieve the same effect, and whether it would extend to settings where paired data is unreliable, is left for future work.

B.5 STYLE SIGNALS ARE COMPRESSED ON SHORT CODE

On short-code tasks such as HumanEval+, the style signal provides limited traction for optimization: the separation between human and LLM distributions is smaller, so $I_s(Z)$ remains low (Eq. equation 2). This is an inherent property of the task distribution rather than a design flaw of our framework, but it means that style supervision yields smaller benefits on short code than on industrial code.

B.6 QUALITATIVE NATURE OF INTERPRETABILITY ANALYSIS

Our node-level perturbation experiments demonstrate the classifier's causal dependence on specific structural features, but they are currently limited to a small number of representative samples. The observed phenomena are consistent and statistically meaningful, but their quantitative distribution

across larger sample sets remains to be verified. A full large-scale attribution analysis is left for future work.

B.7 Model Scale

The fine-tuning experiments use Qwen2.5-Coder-1.5B-Instruct, a model far smaller than state-of-the-art commercial LLMs. That the joint improvement is already achievable at this scale suggests the result is not an artifact of large capacity: the style signal we provide is effective even under tight parameter budgets, where the Function-Style Trade-off is expected to be most severe. Larger models, with more flexible allocation between functional and stylistic information, would likely relax the effective budget B_C defined in Section 3.1 further, making joint improvement easier rather than harder.

C CCE Dataset Construction Pipeline

This appendix provides a detailed description of the construction pipeline of the CCE-Full and CCE-Worse datasets, illustrated in Figure 7.

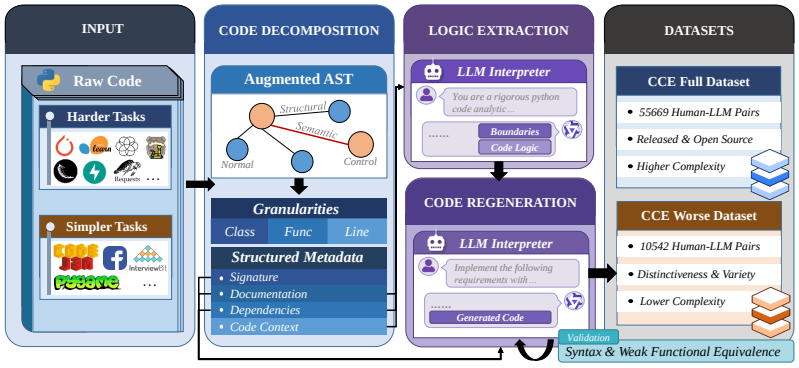


Figure 7: Detailed description of the generation pipeline of the CCE dataset.

Stage 1: Code Decomposition and Structured Information Extraction. We decompose the original human-written code structurally. Using AST analysis tools, we split each code snippet into three granularities: Class, Function, and Line, and extract metadata including dependency libraries, signatures, docstrings, and local context. These serve as input conditions for LLM generation, while the specific implementation details are excluded. For simple snippets of fewer than three lines, we assign a generic logic description directly without invoking the LLM.

Stage 2: Logic Extraction. We feed the docstring and raw code into an LLM, requiring it to output a natural-language *logic specification*: describing *what* the code does, without specifying *how* it is implemented. This forces the LLM to understand the code's functionality and express it as a logical description, rather than reproducing implementation details (which we consider cheating or over-exposure of information). The logic description inevitably retains some structural information from the human code (e.g., problem decomposition strategies), which is acceptable as it ensures the generated LLM code addresses the same requirement. To reduce computational cost, identical code fragments are cached.

Stage 3: LLM Code Generation. We compile the structured information from the previous two stages, including logic specification, signature, documentation, dependencies, and context, into a standardized requirement document, which is fed into the LLM for code generation. The LLM is explicitly instructed to output only code, without Markdown wrappers or additional explanations. Generated outputs are then checked for syntax, failed parses trigger up to two retries, making correctness less strict than first-shot.

Through this pipeline, LLMs generate code based solely on the logic specification rather than the human implementation, thereby preserving their own stylistic characteristics while ensuring functional alignment with the human code. It is worth noting that this approach may cause the LLM to

lose some structural information: under complex logic, the model tends to learn problem-solving strategies implicit in the logic description rather than following the exact implementation path of the human code, but overall, weak functional equivalence and stylistic independence are maintained.

For dataset construction, we use Qwen-2.5-Coder-7B-Instruct-AWQ as the model for both logic extraction and code generation. This choice is motivated by three considerations: (1) the 7B parameter scale offers a favorable capability-cost trade-off for code generation; (2) AWQ quantization enables efficient inference under limited resources; (3) the model performs stably on benchmarks such as HumanEval, with generated code quality representative of typical LLM style.

CCE-Full and CCE-Worse collect 55,669 and 10,542 human–LLM code pairs, respectively.

D Design Principles for Evaluation Metrics

This appendix describes the design principles underlying the CCE evaluation framework. Rather than producing a single opaque score, we adopt a multi-dimensional metric system that distinguishes between *capability*, *naturalness*, and *stability*, interpreted relative to human-written code as the reference baseline.

Table 7: Metric design for CCE evaluation. Scores are combined by the task-dependent weights below and gated by `syntax_rate`.

Metric	Direction	Scoring Rule	Class	Function	Line
Core metrics					
`prob_mean`	Human-aligned	Prefer ≤ 0.55 (Full) / $[0.4, 0.7]$ (Worse)	0.35	0.32	0.45
`prob_std`	Lower	$1 - \sigma/\tau, \tau = 0.25/0.20/0.35$	0.27	0.26	0.30
Documentation metrics					
`doc_rate`	Higher	$\min(1, \text{rate}/0.3)$	0.10	0.10	0.05
`doc_len_mean`	Bounded	Ideal range $[30, 200]$, penalizes over and under documentation	0.08	0.08	0.05
Structural metrics					
`entropy_mean`	Lower	1 if $H \leq 2.5$, else linear penalty	0.08	0.08	0.05
`avg_cc_mean`	vs. Human	$(c_h - c)/c_h$, clipped to $[-2, 2]$	0.05	0.08	0.02
`max_cc_max`	vs. Human	$(c_h^{\max} - c^{\max})/c_h^{\max}$	0.03	0.04	0.02
`avg_depth_mean`	Bounded	$1 - \max(0, d - \tau_d)/20, \tau_d = 5$ (3 for Line)	0.02	0.02	0.03
`code_len_mean`	Bounded	$1 - \max(0, \ell - \tau_\ell)/1000, \tau_\ell = 500$ (150 for Line)	0.02	0.02	0.03
`cf_count_mean`	vs. Human	Interpreted jointly with cyclomatic complexity	–	–	–
Diagnostic (not scored)					
`unk_content_rate`	Monitoring	Detects hallucinated identifiers	–	–	–

D.1 Metric Definitions

We categorize metrics into three groups by their role in the final score:

Syntax gate. `syntax_rate` is treated as a hard gate, since even if we have other datasets for correctness evaluation, syntax correctness to our surprise fails on multiple on-edge LLM coders, which is then significantly weighted in our metric design. Scores are scaled by $\min(1, \texttt{syntax_rate}/0.9)$, so code that fails to parse or falls below 90% syntax validity is penalized proportionally. This ensures that syntactic capability is a prerequisite for all downstream metrics.

Naturalness and Stability. `prob_mean` and `prob_std` are the primary signals reflecting stylistic naturalness and prediction stability. Together they account for over 60% of the final weight in every task type. `prob_mean` is compared against human-level style probability (near 0.55 for high-quality data), `prob_std` is penalized when predictions are unstable.

Structural and documentation metrics. The remaining metrics serve as complementary indicators. Complexity metrics (`avg_cc_mean`, `max_cc_max`, `cf_count_mean`) are compared against the human baseline: lower complexity is rewarded, higher is penalized. Depth and length are bounded by relaxed thresholds. Documentation metrics (`doc_rate`, `doc_len_mean`) reward sufficient but not excessive documentation. Entropy is penalized when the AST node-type distribution is unusually scattered.

D.2 Scoring Rules and Task-Dependent Weights

Table 7 summarizes the scoring rule, preferred direction, and weight of each metric on the three granularities of the task (Class, Function, Line). Weights within each column sum to 1, the final score is a weighted combination of normalized per-metric scores, gated by syntax rate and mapped to $[0, 1]$.

E SlotGCN Architecture Details

The complete network structure of SlotGCN is as follows.

Node encoding layer. The type vocabulary has size 103, and the content vocabulary has size 418,260. Each node receives two categorical features (type and content) and two scalar features (depth and control-flow indicator). The categorical features are embedded into 16-dimensional vectors via lookup tables, then concatenated with the scalars to form a 34-dimensional raw feature vector. A linear projection maps this vector to the hidden dimension $d = 64$, followed by BatchNorm and ReLU activation.

Dual-channel GCN propagation. We use two stacked `SlotGCNLayer` modules, each consisting of two parallel GCNConv layers — one for structural edges (AST parent–child dependencies) and one for semantic edges (variable/function references). Both operate on the same node representations but with separate parameters. Each GCN layer is followed by BatchNorm and ELU activation, and the output of each channel is added to its input via a residual connection (with dropout $p = 0.3$ applied before the addition). This design preserves channel-specific information while allowing cross-channel interaction through the residual sum after the first layer.

SlotReadout. For each channel, we compute node-level importance scores via two separate projection networks (linear with Tanh), followed by a shared attention network (two-layer MLP: $64 \to 32 \to 1$ with ReLU):

$$\alpha_{\text{struct},n} = \text{MLP}_{\text{attn}}(\tanh(\mathbf{W}_{\text{proj,struct}} \cdot \mathbf{h}_{\text{struct},n})), \tag{7}$$

$$\alpha_{\text{sem},n} = \text{MLP}_{\text{attn}}(\tanh(\mathbf{W}_{\text{proj,sem}} \cdot \mathbf{h}_{\text{sem},n})). \tag{8}$$

The channel-level logits are obtained by summing these scores over all nodes and normalized with Softmax:

$$\boldsymbol{\beta} = \text{Softmax}([\sum_n \alpha_{\text{struct},n}, \sum_n \alpha_{\text{sem},n}]). \tag{9}$$

The final node representation is the weighted sum $\mathbf{h}_v = \beta_{\text{struct}}\mathbf{h}_{\text{struct}} + \beta_{\text{sem}}\mathbf{h}_{\text{sem}}$, which is then aggregated by global mean pooling over the graph to produce the graph-level feature $\mathbf{g} \in \mathbb{R}^{64}$.

Multi-modal fusion. The graph representation $\mathbf{g}$ captures local topological structure but lacks global statistics and high-level semantics. To compensate, we introduce two complementary features: (1) a six-dimensional metadata vector $\mathbf{m}$, consisting of docstring presence, control-flow complexity, type hint density, documentation ratio, normalized code length, and signature-call match rate; and (2) a CodeBERT embedding $\mathbf{c} \in \mathbb{R}^{768}$ for global semantic context. Both are projected to 32-dimensional spaces via separate MLPs, then concatenated with $\mathbf{g}$ to form a 128-dimensional fused representation:

$$\mathbf{z} = [\mathbf{g}; \text{MLP}_{\text{meta}}(\mathbf{m}); \text{MLP}_{\text{codebert}}(\mathbf{c})]. \tag{10}$$

A two-layer MLP classifier ($128 \to 64 \to 1$) with ReLU and dropout $p = 0.5$ produces the final logit. The model is trained with binary cross-entropy loss.

Training configuration. We use the AdamW optimizer with a learning rate of 1×10^{-6}, batch size 32, and a maximum of 300 epochs. Early stopping is applied with patience 5. Gradient clipping is set to max_norm $= 1.0$, and a ReduceLROnPlateau scheduler (factor 0.5, patience 3) is used to reduce the learning rate when validation loss plateaus.

F Extended Theoretical Details for Section 3

This appendix provides formal definitions, derivations, proofs, empirical interpretations, and limitations for the theoretical framework in Section 3. The purpose is to characterize conditional information-level relationships among functional and stylistic adaptation, rather than to provide a complete dynamical theory of neural network optimization.

F.1 Technical Assumptions

This subsection gives the full formal statements of the assumptions summarized in Section 3.

Assumption 1 (Empirical Information Quantities)**.** *All information quantities are empirical, computed with respect to a fixed distribution induced by a finite dataset $\mathcal{D}_{\mathcal{C}} = \{(x_i, y_{f,i}, y_{s,i})\}_{i=1}^{n}$. Hence $I(Z; Y_f, Y_s) \leq H(Y_f, Y_s) < \infty$, no intrinsic bottleneck of a deterministic network is claimed.*

Assumption 2 (Reachable Set and Reference Level)**.** *$\mathcal{Z}_{\mathcal{C}}$ is the set of representations reachable under the admissible model, parameter-update, data, and optimization constraints. $B_{\mathcal{C}}$ is a reference level for this set, not a hard bottleneck. If the supremum is unattained, ε-optimal representations may be used.*

Assumption 3 (Meaningful Proxy Metrics)**.** *Functional and stylistic informations are operationalized via standard performance metrics and style-sensitive classifiers. Note that they are meaningful proxies, not exact estimators.*

Assumption 4 (Matched Effective Level)**.** *We compare two representations $Z_0, Z_1 \in \mathcal{Z}_{\mathcal{C}}$ that attain the same effective level, i.e., $J_{\mathcal{C}}(Z_0) = J_{\mathcal{C}}(Z_1) = B_{\mathcal{C}}$. Exact matching is relaxed to $|J_{\mathcal{C}}(Z_1) - J_{\mathcal{C}}(Z_0)| \leq \delta$ in practice (Appendix F.8).*

Assumption 5 (ε-Functional Sufficiency)**.** *The representation Z satisfies the functional target in the ε-optimal sense, i.e., $I_f(Z) \geq R_f(\tau, \epsilon; \mathcal{C})$ up to the same ε as in Assumption 2. Exact attainment of the infimum is not required.*

F.2 Justification for Assumptions

Justification for Assumption 1. The framework is meant to explain an empirical observation, so the quantities it reasons about must be estimable from finite data. Working with the empirical distribution avoids two common over-claims: that a deterministic network has an intrinsic information bottleneck, and that mutual information between a continuous representation and discrete labels is well-defined without a distributional assumption. Under this assumption all quantities are finite and the identities below hold exactly. If information were instead defined with respect to an unknown true distribution, the derivations would still hold in expectation, but the empirical estimates used in the experiments would be biased, and the guarantees would become asymptotic rather than exact.

Justification for Assumption 2. The purpose of $B_{\mathcal{C}}$ is to give a common yardstick for comparing two representations trained under the same configuration, not to assert a universal capacity limit. Defining $\mathcal{Z}_{\mathcal{C}}$ by the admissible model, optimizer, and data makes this reference level configuration-specific, which matches how the experiments compare fine-tuning runs under matched settings. The ε-relaxation is included because the supremum over a non-compact representation set need not be attained. In practice, representations that come within ε of the supremum behave identically for the purposes of the reallocation identity. If $B_{\mathcal{C}}$ were instead treated as a hard bound, the framework would be making a claim the experiments cannot test, and Proposition 1 would be vacuous rather than operational.

Justification for Assumption 3. Mutual information between a representation and a semantic attribute such as code-style is not directly measurable, any empirical study must substitute observable proxies. We use unit-test pass rates for functional information and a style-sensitive classifier for stylistic information, both of which correlate with the corresponding theoretical quantities on the target distribution. The assumption states that these proxies are *meaningful*, i.e., monotone in the underlying quantity, without claiming they are exact. This is the weakest form under which the predictions of the framework can be tested: if the proxies were uncorrelated with the theoretical quantities, no experiment could falsify the predictions, and if they were assumed exact, any measurement error would be misattributed to the theory.

Justification for Assumption 4. The reallocation identity is a statement about how two representations *share* a fixed total amount of task-relevant information, so the comparison is only meaningful when that total is held fixed. If $J_{\mathcal{C}}(Z_0)$ and $J_{\mathcal{C}}(Z_1)$ differed, an observed change in style could reflect a change in total information rather than a reallocation, and Proposition 1 would no longer isolate the trade-off. Exact matching is impractical because $J_{\mathcal{C}}$ is estimated through proxies, so we relax the condition to a δ-match and treat the residual as approximation error, with the corresponding Corollary 1 stating exactly how the identity degrades. Without this assumption, a measured trade-off could always be explained away as a difference in total information, and the framework would have no empirical content.

Justification for Assumption 5. The stylistic bound is only informative for representations that actually solve the task, so we require the functional target to be met. R_f is defined as an infimum over the admissible representation–decoder class, and an infimum need not be attained by any admissible pair, requiring exact equality would exclude every realistic model. The ε-relaxation, using the same ε as Assumption 2, keeps the two approximations consistent: a representation that is ε-optimal in total information and ε-functionally sufficient satisfies the stylistic bound up to the same tolerance. If the assumption required exact attainment, Proposition 1 would apply only to an idealized class of representations that does not correspond to any trained model, and the bound would not constrain the experiments.

F.3 Empirical Probability Space and Information Quantities

This subsection supports Assumption 1 in Appendix F.1. All information quantities are defined with respect to a fixed empirical distribution induced by a finite dataset. Let

$$\mathcal{D}_{\mathcal{C}} = \{(x_i, y_{f,i}, y_{s,i})\}_{i=1}^{n} \tag{11}$$

denote the adaptation or evaluation set associated with configuration $\mathcal{C}$. The empirical distribution is $\widehat{p}_{\mathcal{C}}(X, Y_f, Y_s)$. The learned representation Z induces $\widehat{p}_{\mathcal{C}}(Z, Y_f, Y_s)$. Thus all mutual informations are empirical. For finite support,

$$I(Z; Y_f, Y_s) \le H(Y_f, Y_s) < \infty. \tag{12}$$

This formulation avoids interpreting the framework as claiming an intrinsic information bottleneck of a deterministic neural network.

F.4 Reachable Representations and Effective Information Levels

This subsection supports Assumption 2. $\mathcal{Z}_{\mathcal{C}}$ is the set of representations reachable under the admissible model, parameter-update, data, and optimization constraints. For each Z, $J_{\mathcal{C}}(Z) = I(Z; Y_f, Y_s)$, and $B_{\mathcal{C}} = \sup_{Z \in \mathcal{Z}_{\mathcal{C}}} J_{\mathcal{C}}(Z)$. $B_{\mathcal{C}}$ is a reference level associated with the reachable set, not an intrinsic hard bottleneck. If the supremum is not attained, ε-optimal representations satisfying $J_{\mathcal{C}}(Z^{\varepsilon}) \ge B_{\mathcal{C}} - \varepsilon$ can be used.

F.5 Interaction Information and the Joint Decomposition

This subsection provides the decomposition used in Section 3. The interaction information is defined as

$$I_{\text{int}}(Z) = I(Z; Y_f; Y_s) = I(Z; Y_f) - I(Z; Y_f \mid Y_s). \tag{13}$$

By the chain rule and substitution,

$$I(Z; Y_f, Y_s) = I(Z; Y_f) + I(Z; Y_s) - I(Z; Y_f; Y_s). \tag{14}$$

Thus $J_{\mathcal{C}}(Z) = I_f(Z) + I_s(Z) - I_{\text{int}}(Z)$. Interaction information can be positive or negative, so joint information is not the sum of independent functional and stylistic components.

F.6 Achievable Sets and Pareto Optimality

The achievable set is $\mathcal{A}_{\mathcal{C}} = \{(I_f(Z), I_s(Z)) : Z \in \mathcal{Z}_{\mathcal{C}}\}$. A representation $Z^{\star}$ is Pareto optimal if no $\widetilde{Z}$ improves both coordinates with at least one strict improvement. The framework does not prove that the Pareto frontier must be negatively sloped, that is an empirical question. Instead, Proposition 1 gives a conditional consistency relation.

F.7 Proof of the Fixed-Level Information Reallocation Observation

This subsection proves Proposition 1 in Section 3.2. For each Z, $J_{\mathcal{C}}(Z) = I_f(Z) + I_s(Z) - I_{\text{int}}(Z)$. Since $J_{\mathcal{C}}(Z_0) = J_{\mathcal{C}}(Z_1) = B_{\mathcal{C}}$, we have $J_{\mathcal{C}}(Z_1) - J_{\mathcal{C}}(Z_0) = 0$. Substituting the decomposition gives $\Delta I_f + \Delta I_s - \Delta I_{\text{int}} = 0$, hence $\Delta I_s = -\Delta I_f + \Delta I_{\text{int}}$.

This identity requires comparison at the same effective level, otherwise the difference may reflect a change in total information rather than reallocation.

F.8 Approximate Information-Level Matching

This subsection supports the relaxation in Assumption 4. Exact equality is difficult to establish, we provide a relaxed version.

Corollary 1 (Approximate Information Reallocation). *Suppose* $Z_0, Z_1 \in \mathcal{Z}_{\mathcal{C}}$ *satisfy* $|J_{\mathcal{C}}(Z_1) - J_{\mathcal{C}}(Z_0)| \le \delta$. *Then* $|\Delta I_f + \Delta I_s - \Delta I_{\text{int}}| \le \delta$.

Proof. Directly follows from the decomposition and the bound on the difference of J. □

In practice, controlled comparisons are used as operational approximations, the residual mismatch is interpreted as approximation error.

F.9 Functional Information Requirement

This subsection supports the definition of R_f in Section 3.2. The admissible representation–decoder class is $\mathcal{H}_{\mathcal{C}} = \{(Z, g) : Z \in \mathcal{Z}_{\mathcal{C}}, g \in \mathcal{G}_{\mathcal{C}}\}$. The required functional information is

$$R_f(\tau, \epsilon; \mathcal{C}) = \inf_{(Z,g)\in\mathcal{H}_{\mathcal{C}}} \{I(Z; Y_f) : \mathcal{L}_f(g(Z), \tau) \le \epsilon\}. \tag{15}$$

Every successful pair satisfies $I_f(Z) \ge R_f(\tau, \epsilon; \mathcal{C})$ up to the same ε used in Assumption 5.

F.10 Proof of the Necessary Stylistic Information Bound

This subsection proves Proposition 1 in Section 3.2. For any Z, $I_f(Z) + I_s(Z) - I_{\text{int}}(Z) \le B_{\mathcal{C}}$, so $I_s(Z) \le B_{\mathcal{C}} - I_f(Z) + I_{\text{int}}(Z)$. If Z achieves the functional target, $I_f(Z) \ge R_f$, hence $I_s(Z) \le B_{\mathcal{C}} - R_f + I_{\text{int}}(Z)$.

This is a necessary upper bound, it does not guarantee reachability.

F.11 SFT and DPO as Information Regimes

This subsection supports the preference-specific discussion in Section 3.2 and Prediction P3. The SFT objective is $\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{(x,y)\sim\mathcal{D}} \log p_\theta(y \mid x)$, and its targets contain both functional and stylistic information. DPO optimizes a preference objective over comparisons. Let Y_p denote the preference variable. The style-specific information in preferences is $I(Y_p; Y_s \mid Y_f)$. If this is negligible, preferences contain little additional style information beyond functionalit, in this regime DPO is functionally dominated at the supervision level. If substantial, preferences provide an additional stylistic source. This does not imply that the optimization objectives, gradients, or learned geometries are equivalent.

F.12 Structural Gain and Its Information-Theoretic Identity

This subsection supports Proposition 2 in Section 3.2. Let S be an auxiliary structural representation. Define $G_S = I(Z; Y_s \mid Y_f) - I(Z; Y_s \mid S, Y_f)$. Using the chain rule,

$$I(S, Z; Y_s \mid Y_f) = I(S; Y_s \mid Y_f) + I(Z; Y_s \mid S, Y_f), \tag{16}$$

$$I(S, Z; Y_s \mid Y_f) = I(Z; Y_s \mid Y_f) + I(S; Y_s \mid Z, Y_f). \tag{17}$$

Equating gives $G_S = I(S; Y_s \mid Y_f) - I(S; Y_s \mid Z, Y_f)$. Thus $G_S \ge 0$ is not a universal theorem, positive gain is an empirical condition.

F.13 Capacity-Matched Non-Structural Controls

This subsection details the control used in Proposition 2. A positive G_S alone does not establish structural inductive bias because incorporating S may increase capacity. Define a control S_{ctrl} with matched dimensionality, injection mechanism, trainable parameters, and computational cost, but without the relevant structure (e.g., random projection, permutation). Let $G_{\text{ctrl}} = I(Z; Y_s \mid Y_f) - I(Z; Y_s \mid S_{\text{ctrl}}, Y_f)$. With cost $\Omega(S)$ and trade-off η, define $G_S^{\text{net}} = G_S - \eta\Omega(S)$ and $G_{\text{ctrl}}^{\text{net}} = G_{\text{ctrl}} - \eta\Omega(S_{\text{ctrl}})$. A genuine benefit is declared when $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$. This comparison is conditional on the chosen Ω and η.

F.14 Structural Perturbation and Representation Reallocation

Removing or perturbing S does not necessarily cause immediate degradation because representations may reallocate to compensate. The framework predicts that if $G_S^{\text{net}} > G_{\text{ctrl}}^{\text{net}}$, then removing S should cause greater degradation or require greater compensation than removing the matched control. This allows for reallocation and avoids treating G_S as static.

F.15 Operational Connection Between Theory and Experiments

This subsection supports the operationalization referenced in Section 3. Direct estimation of high-dimensional mutual information is difficult. Experiments test observable implications: functional information is operationalized via functional performance metrics, stylistic information via style-sensitive metrics, and approximate level matching via controlled comparisons (as in the δ-relaxed condition). Detailed experimental procedures are specified in the experimental section.

F.16 Justification of the Testable Predictions

This subsection supports the predictions in Section 3.

Prediction 1 follows from Corollary 1: at matched levels, the reallocation identity constrains observed changes.

Prediction 2 follows from Proposition 1: higher functional requirement tightens the upper bound on stylistic information under comparable interaction structures.

Prediction 3 follows from the discussion in §F.11: distinct stylistic effect requires substantial conditional stylistic information in preferences.

Prediction 4 follows from the structure–control comparison in §F.13: only when the net structural gain exceeds the control is a genuine structural benefit indicated.

F.17 Limitations of the Information-Level Framework

Explicit limitations include:

1. It does not prove a universal small information budget determined by parameter count or LoRA rank.
2. It does not derive a negatively sloped trade-off from first principles, its presence is empirical.
3. Mutual information alone does not characterize optimization dynamics, SFT and DPO may differ in trajectories and geometries even with similar supervision content.
4. Empirical metrics are proxies for theoretical information quantities, not exact estimators.
5. Structural gain should be assessed through controlled comparative experiments, not as an immutable property, and Proposition 2 is conditional on the chosen Ω and η.

F.18 Relation to Information Bottleneck

The framework is conceptually related to the information bottleneck principle $I(Z;X) \leq I_c$, but differs in that it does not impose an explicit bottleneck objective and focuses on allocation of joint task-relevant information, not input compression. It is an information-decomposition analysis rather than a direct application of the classical bottleneck.

F.19 Summary of Conditional Claims

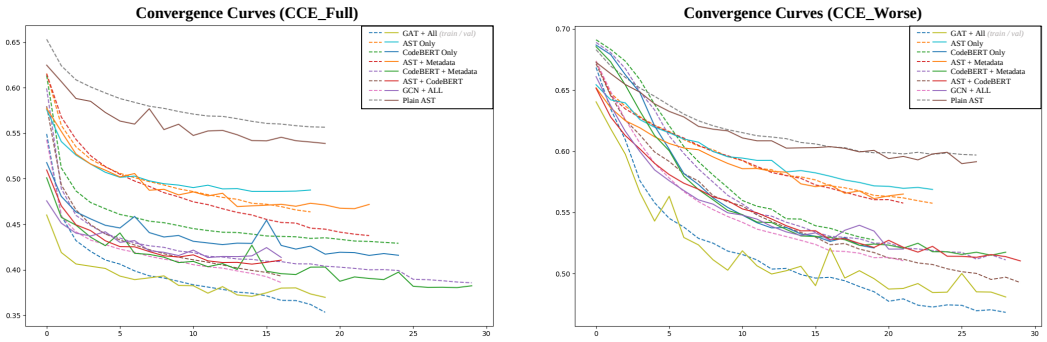
The theoretical claims are conditional on:

1. Information quantities are empirical with finite support (Assumption 1).
2. Reallocation comparisons are at the same (or approximately same) effective information level (Assumption 4).
3. Functional and stylistic metrics are meaningful proxies (Assumption 3).
4. A distinct DPO style effect requires substantial conditional preference information.
5. Structural benefits must exceed those of a capacity-matched control (Proposition 2).

Under these conditions, the framework provides a formal language for analyzing observed Function-Style Trade-offs and distinguishing information reallocation, functional demand, and structural inductive bias.

G Complete Experimental Results

G.1 Diagrams of Ablation Study and Cross-Model Evaluation



(a) CCE-Full (b) CCE-Worse

Figure 8: Training convergence of eight ablation configurations on CCE-Full and CCE-Worse. GAT + All achieves highest accuracy while GCN + All (SlotGCN) offers the best accuracy-efficiency trade-off.

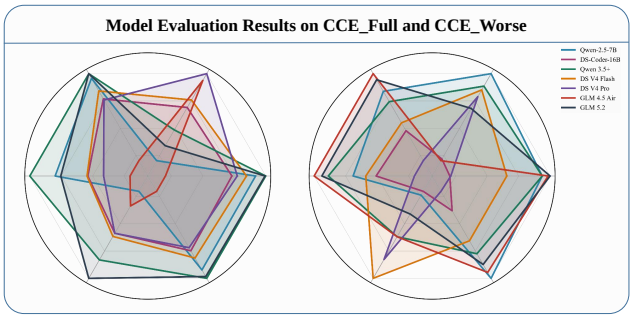


Figure 9: Radar graph for cross-model style evaluation across six dimensions (syntax, human alikeness, entropy, complexity, doc, nesting) on CCE-Full and CCE-Worse.

G.2 Complete Granularity Breakdowns

Table 8 presents the complete granularity breakdowns of syntax rate, style probability, and quality score across Class (C), Function (F), and Line (L) levels for all evaluated models on both CCE-Full and CCE-Worse. Each model occupies three rows, corresponding to the three metrics.

Table 8: Complete granularity breakdowns on CCE-Full and CCE-Worse. C = Class, F = Function, L = Line. Syntax and P(LLM) are reported in %, Q. is the Style Quality Score.

	DS V4 Pro			DS V4 Flash			Qwen 3.5+			GLM 5.2			GLM 4.5 Air			Qwen-2.5-7B			DS-Coder-16B		
Metric	**C**	**F**	**L**	**C**	**F**	**L**	**C**	**F**	**L**	**C**	**F**	**L**	**C**	**F**	**L**	**C**	**F**	**L**	**C**	**F**	**L**
CCE-Full																					
Syn.	61.1%	76.4%	92.1%	72.1%	81.8%	96.3%	95.9%	**99.4%**	98.3%	94.5%	**100.0%**	97.5%	28.6%	27.1%	30.2%	82.2%	91.2%	96.9%	72.1%	65.4%	83.8%
P(LLM)	61.4%	55.3%	55.4%	66.0%	69.5%	61.9%	78.4%	79.5%	71.1%	82.4%	84.3%	77.8%	59.1%	55.8%	62.2%	84.6%	86.8%	87.0%	75.2%	66.8%	65.3%
Q.	0.441	0.547	**0.684**	0.522	0.557	0.674	0.621	0.584	0.644	0.616	0.596	0.628	0.210	0.197	0.224	0.536	0.561	0.617	0.496	0.448	0.615
CCE-Worse																					
Syn.	70.5%	79.1%	99.0%	88.2%	90.0%	99.0%	98.5%	98.4%	99.2%	98.9%	**100.0%**	99.8%	**99.5%**	99.0%	99.7%	96.5%	99.5%	99.5%	87.4%	90.3%	89.8%
P(LLM)	75.0%	74.6%	68.9%	85.5%	82.5%	60.5%	91.4%	76.1%	58.9%	92.7%	87.9%	64.2%	90.4%	92.7%	86.5%	83.2%	84.1%	52.7%	91.3%	94.1%	85.3%
Q.	0.463	0.511	0.675	0.557	0.559	**0.682**	0.572	0.553	0.652	0.567	0.528	0.658	0.567	0.550	0.599	0.578	0.549	0.661	0.548	0.544	0.607

H Feature Definitions and The Perturbation Experiment

H.1 Complete Definitions of the 14 Dimensions of Features

Below we provide the complete definitions and computation methods for the 14 code-style features:

1. `pass_count`: The number of `pass` statements in the code. `pass` is the only null statement in Python, typically used as a placeholder in function or class stubs. Frequent use of `pass` by LLMs indicates a tendency to generate skeleton code that is structurally complete but logically empty.

2. `docstring_length`: The total character length of docstrings, extracted using the regular expression. Human-written code has significantly longer docstrings than LLM-generated code.

3. `doc_per_function`: The average docstring length per function or class definition.

4. `decorator_count`: The number of decorator usages (e.g., `@staticmethod`, `@property`). Human code tends to use decorators more frequently.

5. `unused_variables`: The number of variables that are defined but never used, identified via AST analysis as variables with a `Store` context but no corresponding `Load` context.

6. `function_def_count`: The total number of function definitions (including nested functions). LLMs tend to define more auxiliary functions on complex tasks, indicating their habit of decoupling.

7. `class_def_count`: The total number of class definitions.

8. `nested_scope_depth`: The maximum depth of nested scopes, computed as the maximum value of the AST node `depth` attribute. LLMs tend to generate deeper nesting structures.

9. `assert_count`: The number of `assert` statements. LLMs tend to add more defensive assertions.

10. `type_annotation_length`: The total character length of type annotations. On competition-style code (CCE-Worse), LLMs tend to over-annotate types.

11. `extra_error_handling`: The number of exception handling statements (`try-except`).

12. `lambda_count`: The number of lambda expressions.

13. `defense_score`: A composite score for defensive programming, combining coding habits such as `assert_count`, `extra_error_handling`, and null-checking practices.

14. `avg_identifier_len`: The average length of identifiers (variable names, function names), which may be an indicator of how LLMs tend to name objects when given the chance to improvise.

Detailed analysis: LLMs tend to under-generate on `pass_count` and over-generate on `nested_scope_depth`, forming a two-sided polarization. This polarization is even more pronounced on CCE-Full (`pass_count`: 6.98× vs. 5.79×, `nested_scope_depth`: 1.20×

vs. 1.34×), indicating that both over-engineering and under-engineering are amplified as task complexity increases.

Table 9: Complete 14-feature comparison between human and LLM code on CCE-Full and CCE-Worse. Diff = LLM − Human. Ratio = LLM / Human (when LLM>Human) or Human / LLM (when LLM<Human). Best per column in bold.

	CCE-Full				CCE-Worse			
Feature	**Human**	**LLM**	**Diff**	**Ratio**	**Human**	**LLM**	**Diff**	**Ratio**
extra_error_handling	0.093	0.071	−0.022	0.77×	0.019	0.024	+0.005	1.29×
type_annotation_length	10.904	9.611	−1.292	0.88×	0.097	0.235	+0.137	**2.41**×
docstring_length	181.390	18.473	−162.917	**0.10**×	42.168	1.850	−40.318	**0.04**×
function_def_count	2.097	2.216	+0.120	1.06×	0.761	1.111	+0.350	1.46×
class_def_count	0.352	0.381	+0.029	1.08×	0.181	0.256	+0.075	1.42×
decorator_count	0.562	0.241	−0.322	**0.43**×	0.019	0.022	+0.003	1.15×
unused_variables	2.427	2.695	+0.268	1.11×	1.457	1.713	+0.257	1.18×
avg_identifier_len	8.958	8.938	−0.020	1.00×	5.564	6.095	+0.531	1.10×
lambda_count	0.107	0.087	−0.020	0.81×	0.023	0.016	−0.006	0.72×
nested_scope_depth	0.898	1.075	+0.177	**1.20**×	0.559	0.752	+0.193	**1.34**×
assert_count	0.057	0.075	+0.018	1.30×	0.019	0.019	−0.001	0.96×
pass_count	0.044	0.305	+0.261	**6.98**×	0.013	0.074	+0.061	**5.79**×
doc_per_function	81.828	6.398	−75.430	**0.08**×	30.184	1.298	−28.885	**0.04**×
defense_score	0.121	0.108	−0.013	0.89×	0.029	0.034	+0.005	1.18×

H.2 Full Perturbation Records

H.2.1 Experimental Setup

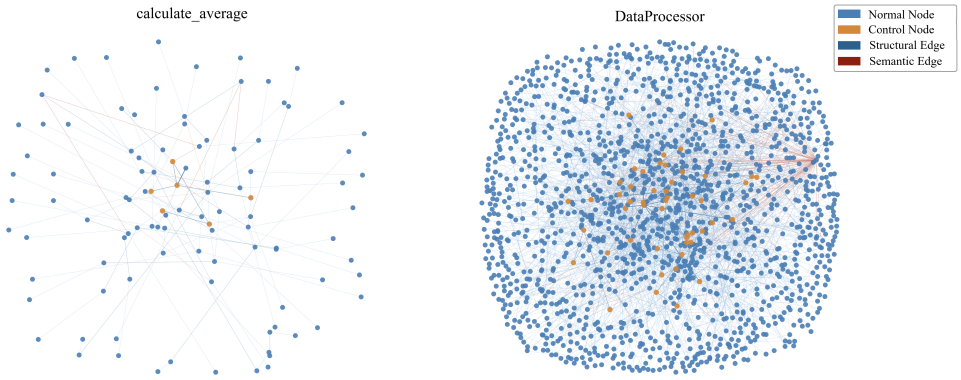


Figure 10: Augmented ASTs of the shorter `calculate_average` method (left, 104 nodes) and the longer `DataProcessor` class (right, 1,657 nodes). Semantic edges are highlighted in red.

The probing script `probe_ast.py` takes a code string or file as input, constructs an Augmented AST, and feeds it into the classifier to obtain the output probability. Perturbation operations include modifying node attributes such as `is_cf` (control-flow flag), `depth` (nesting depth), `type_id`, and `content_id`. Two types of code are selected for experiments: (1) shorter code with `calculate_average` (104 nodes) as the example, representing simple tasks; and (2) longer code `DataProcessor` (1,657 nodes) as the example, a standard data-processor class, representing industrial-scale code.

H.2.2 Perturbation Experiments on Shorter Code

Figure 10 shows the Augmented ASTs of both code samples side by side. The left panel corresponds to the shorter `calculate_average` method (104 nodes), while the right panel shows the longer `DataProcessor` class (1,657 nodes). Semantic edges are highlighted in red, distinguishing them from the structural edges that form the tree skeleton.

Raw Code:

```
def calculate_average(numbers, weights=None):
    if not numbers:
        return 0.0
```

```
    total = 0.0
    count = 0

    for num in numbers:
        try:
            if weights is not None:
                total += num * weights[count]
            else:
                total += num
            count += 1
        except IndexError:
            print(f"Warning: Index {count} out of range for weights")
            continue
        except Exception as e:
            print(f"Error processing number {num}: {e}")
            continue

    if count == 0:
        return 0.0
    return total / count
```

Critical node index table:

Table 10: Critical Nodes in the Shorter Code

Index	AST Type	Corresponding Code
1	FunctionDef	`def calculate_average(...)`
21	For	`for num in numbers:`
26	Try	`try:`
27	If	`if weights is not None:`
53	AugAssign	`count += 1`

Complete perturbation records:

- Baseline P(LLM) = 0.8246
- Remove innermost `If` (idx=27, is_cf 1→0): P = 0.8251, Δ = +0.0005
- Remove `Try` (idx=26, is_cf 1→0): P = 0.8228, Δ = -0.0018
- Remove outermost `For` (idx=21, is_cf 1→0): P = 0.8210, Δ = -0.0036
- Batch depth +1: P = 0.8402, Δ = +0.0156
- Batch depth -1: P = 0.8293, Δ = +0.0047
- Single-node `AugAssign` depth 4→1: P = 0.8590, Δ = +0.0344

Critical findings: Single-node depth adjustment (accumulator 4→1) increases the probability by 3.44 percentage points, more than double the effect of batch depth adjustment (+1.56 pp), indicating that the classifier focuses on key semantic nodes rather than average depth. The diminishing marginal effect of control-flow markers (cumulative -0.36 pp across three layers) suggests that discriminative information is sparsely encoded.

H.2.3 Perturbation Experiments on Longer Code

Original code (`DataProcessor` class, 1,657 AST nodes):

Since the code is approximately 150 lines long, we only describe its core structure in words. The complete `DataProcessor` class contains the following 9 methods: `__init__`, `_setup_logging`, `load_data`, `validate_records`, `transform_data`, `aggregate_by`, `generate_quality_report`, `run_pipeline`, and `reset`, simulating an industrial-level

ETL data processing module with comprehensive logging, exception handling, and data transformation pipelines. The full code is provided in Appendix H.

The right panel of Figure 10 shows the Augmented AST of this longer sample. Despite its dense appearance, the graph remains relatively sparse with a nearly 1:1 edge-to-node ratio.

Complete perturbation records:

- Baseline P(LLM) = 0.4355 (classified as Human)
- Batch depth +1: P = 0.4856, Δ = +0.0501
- Remove all control-flow markers (all is_cf set to 0): P = 0.4379, Δ = +0.0024
- Replace all `logger` calls with `print`: P = 0.5520, Δ = +0.1165, judgment flips

Critical findings: Removing all control-flow markers has almost no effect (+0.24 pp), indicating that discriminative signals in large-scale code shift from shallow control-flow to deeper semantic patterns. The `logger`→`print` replacement triggers an 11.65 pp flip, providing the strongest evidence for the *engineering professionalism* dimension.

H.2.4 Scaled Statistical Validation of Node-Level Perturbations

To verify whether the perturbation experiment findings in Section 5.4 generalize beyond the two representative samples, we randomly sampled 100 code snippets from CCE-Full (prioritizing the function level). For each sample, we searched all nodes for depth perturbations (depth $\pm$ 1) and control-flow marker flips (is_cf), and recorded the perturbation with the largest impact on the classifier's output probability.

Table 11 presents the detailed distribution and impact statistics. Among the 100 samples, `is_cf_flip` appears most frequently as the strongest perturbation (51 times, 51.0%), followed by `depth_+1` (25 times) and `depth_-1` (24 times). However, `depth_-1` exhibits the largest average absolute change (0.0309), substantially exceeding `is_cf_flip` (0.0235) and `depth_+1` (0.0266). Moreover, the directionality reveals a consistent pattern: both `depth_-1` and `is_cf_flip` tend to increase the LLM probability (mean signed changes of +0.0164 and +0.0190, respectively), while `depth_+1` tends to decrease it (-0.0143). This suggests that shallower nesting and flipped control-flow markers make code appear more "LLM-like" to the classifier.

The single largest perturbation occurred on sample 71, where decreasing the depth of a key semantic node flipped the probability from 0.5608 to 0.7365 (+0.1757), underscoring the classifier's sensitivity to nesting depth as a core representation of code structure. These results are consistent with the finding in Section 5.4 that the classifier focuses on key semantic nodes rather than average depth, nesting depth exerts a stronger influence on classifier decisions than shallow control-flow markers.

Table 11: Impact of node-level perturbations on classifier probability across 100 samples. "Count" indicates how many samples had each operation as the strongest perturbation, "Mean Δ" indicates the average direction of change (positive = increases LLM probability). All change columns are reported in %.

Operation	Count	Mean $\lvert\Delta\rvert$	Std	Med.	Max	Mean Δ
is_cf_flip	51	2.35%	1.81%	2.07%	7.83%	+1.90%
depth_+1	25	2.66%	1.59%	2.85%	5.72%	−1.43%
depth_-1	24	**3.09%**	3.78%	1.70%	**17.57%**	+1.64%

H.2.5 Summary of Perturbation Analysis

Combining the perturbation experiments on both short and long code snippets, we see three phenomena emerge:

The Leverage effect: Single-node depth adjustment (accumulator 4→1) increases probability by 3.44 pp, more than double the effect of batch adjustment, indicating that the classifier focuses on key semantic nodes rather than average depth.

Diminishing returns: Removing three control-flow nodes on short code yields a cumulative decrease of only 0.36 pp, suggesting that discriminative information is sparsely encoded and that different control-flow nodes carry redundant information.

Engineering habit preference: The `logger→print` flip of 11.65 pp far exceeds the effect of removing all control-flow markers (+0.24 pp), demonstrating that the classifier is far more sensitive to the *engineering professionalism* dimension than to the presence of control flow.

I Comparison for Classic Code Samples

Sample 1: `project_point_segment(a, b, c)`

Human implementation (6 lines, directly calling helper functions):

```
def project_point_segment(a, b, c):
    ab = sub(b, a)
    ac = sub(c, a)
    t = inner_product(ac, ab) / inner_product(ab, ab)
    t = max(0.0, min(1.0, t))
    return add(a, mul(ab, t))
```

LLM implementation (approximately 30 lines, re-implementing 4 helper functions from scratch and adding type annotations):

```
from typing import Union, Tuple

def sub(p1, p2):
    return (p1[0] - p2[0], p1[1] - p2[1])
# ... complete implementation of inner_product, mul, add ...
def project_point_segment(a, b, c):
    ab = sub(b, a)
    ac = sub(c, a)
    t = inner_product(ac, ab) / inner_product(ab, ab)
    t = max(0.0, min(1.0, t))
    return add(a, mul(ab, t))
```

The LLM not only re-implemented all 4 helper functions from scratch despite its prompt specifically noting their existence, but also added nested type annotations to each function, resulting in code approximately 5 times longer than the human version.

Sample 2: `_xyxyxyxy_to_xywhr`

Human implementation (18 lines of PyTorch official code, vectorized with in-place update):

```
def _xyxyxyxy_to_xywhr(bboxes, inplace=False):
    if not inplace:
        bboxes = bboxes.clone()
    need_cast = bboxes.dtype != torch.float32
    if need_cast:
        bboxes = bboxes.float()
    # Vectorized computation of four corner points
    x1, y1, x2, y2, x3, y3, x4, y4 = bboxes.unbind(-1)
    cx = (x1 + x2 + x3 + x4) / 4
    cy = (y1 + y2 + y3 + y4) / 4
    # ... computes width, height, and rotation angle ...
    return torch.stack([cx, cy, w, h, theta], dim=-1)
```

DeepSeek Coder implementation (approximately 30 lines, functionally correct but relatively conservative):

```
def _xyxyxyxy_to_xywhr(bboxes):
```

```
    # Similar structure to human implementation, but uses
    # new tensor construction instead of in-place updates
    ...
    return torch.stack([cx, cy, w, h, theta], dim=-1)
```

Qwen Coder implementation (extremely long, verbose implementation with chaotic chained .view() operations and syntax errors even after repeated generation and clear functional request from user):

```
def _xyxyxyxy_to_xywhr(bboxes):
    # Large number of repetitive .view() and .reshape()
    x1 = bboxes[..., 0].view(-1, 1)
    y1 = bboxes[..., 1].view(-1, 1)
    # ... repeats similar operations for tens of lines ...
    # Ultimately produces syntax errors or fails to parse
```

J Complete Fine-Tuning Configurations and Results

J.1 Experimental Overview

All fine-tuning experiments use **Qwen2.5-Coder-1.5B-Instruct** as the base model with LoRA ($r = 64$, $\alpha = 16$, dropout 0.05) applied to the seven projection modules `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. Experiments are uniformly numbered E0–E14. Table 12 summarizes their configurations.

Table 12: Unified overview of fine-tuning experiments E0–E14. RFT denotes rejection-sampling fine-tuning, CCE denotes the style-training split of our benchmark.

ID	Family	Method	Training Data	Key Hyperparameters
E0	—	Base (no fine-tuning)	—	—
E1	SFT	Function-only SFT	HumanEval+ 164	lr 2e-4, 3 ep
E2	SFT	Naive joint SFT	HE 164 + CCE 1000	lr 2e-4, 3 ep
E3	SFT	Balanced joint SFT	HE×8 + CCE×1	lr 1e-4, 2 ep
E4	RFT	Rejection-sampling SFT (×8)	RFT×8 + CCE×1	lr 1e-4, 2 ep
E5	RFT	Same, lower learning rate	RFT×8 + CCE×1	lr 5e-5, 2 ep
E6	RFT	Rejection-sampling SFT (×15, no CCE in RFT selection)	RFT×15	lr 1e-4, 2 ep
E7	RFT	**Recommended SFT**	RFT×15 + CCE×1	lr 1e-4, 2 ep
E8	DPO	Style-preference DPO	CCE style pairs 949	lr 5e-6, $\beta = 0.1$, 2 ep
E9	DPO	Hybrid-preference DPO	CCE+function pairs 1111	lr 5e-6, $\beta = 0.1$, 2 ep
E10	ORPO	Reference-free ORPO	CCE+function pairs 1111	lr 1e-5, $\beta = 0.1$, 2 ep
E11	SimPO	Reference-free SimPO	CCE+function pairs 1111	lr 5e-6, $\beta = 2.0$, 2 ep
E12	DPO	Function-only DPO (no CCE)	Function pairs 162	lr 5e-6, $\beta = 0.1$, 2 ep
E13	ORPO	Function-only ORPO (no CCE)	Function pairs 162	lr 1e-5, $\beta = 0.1$, 2 ep
E14	SimPO	Function-only SimPO (no CCE)	Function pairs 162	lr 5e-6, $\beta = 2.0$, 2 ep

J.2 Data and Evaluation Protocol

HumanEval+ contains 164 tasks (prompt, canonical solution, unit tests) and is used for functional SFT and evaluation. CCE-Full contains 58,343 human expert code snippets paired with LLM-generated counterparts. CCE-Full_Worse contains 10,542. From CCE-Full we sample 1,000 snippets for style SFT and preference construction, and evaluate on 300 CCE-Full and 150 CCE-Worse samples (seed 42, disjoint from training). HumanEval+ is evaluated in full (164 tasks, $k = 10$).

We report four axes. (i) *Strong functional correctness*: HumanEval+ unit-test pass rate and pass@k. (ii) *Weak functional correctness*: syntax rate via `ast.parse`. (iii) *Style naturalness*: `prob_llm` $= P(\text{LLM} \mid c)$ from our classifier, where *lower is more human-like*. (iv) *Quality score*: the composite metric of the main paper, aggregated over complexity, nesting depth, documentation rate, and style mean/variance.

Three evaluation issues were corrected before all reported numbers were produced, as they systematically underestimated fine-tuned performance. First, the style classifier used a dynamic in-process vocabulary and unnormalized metadata at inference, aligning training and inference statistics raised its discrimination accuracy on 300 CCE human-LLM pairs from 67.3% to 81.0% (AUC $0.732 \rightarrow 0.893$). Second, HumanEval+ canonical solutions are function *bodies*, so SFT models output indented continuations, and an old code extractor stripped the leading indentation and misflagged syntax error. After repair, E2 pass rate recovered from 0.002 to 0.218 and E3 from 0.000 to

0.304. Third, CCE generations were truncated at 256 tokens, which broke long, engineering-style outputs mid-string; extending to 512 tokens raised CCE-Full syntax from 0.917 to 0.953 for base and from 0.773 to 0.847 for the RFT model, but due to limited experiment environments, furthering token length is sadly not an option.

J.3 Fine-Tuning Families

SFT (E1–E3). Standard autoregressive cross-entropy on `(instruction, output)` pairs. Function-only SFT (E1) learns canonical solutions but degrades pass rate ($0.538 \rightarrow 0.114$) because the model is pushed into a body-only output pattern, naive joint SFT (E2) improves CCE-Full style ($0.809 \rightarrow 0.366$) at a severe functional cost (0.218), and balanced joint SFT (E3) only partially recovers pass rate (0.304) while retaining CCE-Full style at 0.365. Plain SFT can change style, but not without sacrificing output format.

RFT (E4–E7). Rejection-sampling fine-tuning writes both objectives into *data selection*. From the base model we sample 10 candidates per task at temperature 0.9, retain only candidates that pass unit tests, and select the one with the lowest `prob_llm`. This yields 839 passing candidates out of 1,640 (51.2%), covering 138 tasks. The selected samples are repeated $\times 8$ or $\times 15$ and mixed with CCE before LoRA SFT. Because the targets are complete functions generated by the model itself, the instruct-format output pattern is not disturbed.

DPO (E8, E9, E12). Direct preference optimization on pairwise preferences, using TRL `DPOTrainer` with LoRA on top of the base model and $\beta = 0.1$, lr 5×10^{-6}, 2 epochs. Style preference pairs are constructed by pairing human `raw_body` with the LLM-generated `llm_body` on the same CCE task (949 pairs). Functional preference pairs use the lowest-`prob_llm` passing candidate as `chosen` and the highest-`prob_llm` failing candidate as `rejected` (162 pairs). The hybrid set contains both (1,111 pairs).

ORPO (E10, E13). Reference-free preference optimization that combines an SFT term on `chosen` with an odds-ratio penalty on `rejected`. Used from the base model with lr 1×10^{-5}, $\beta = 0.1$, 2 epochs.

SimPO (E11, E14). Reference-free preference optimization with length-normalized average log-probabilities and a margin term ($\beta = 2.0$, `cpo_alpha`=0.1, lr 5×10^{-6}, 2 epochs).

J.4 Complete Results

Table 13 reports the full E0–E14 results. Style columns are lower-is-better, all other columns are higher-is-better. Entries marked with † were produced under the 256-token CCE protocol and are not directly comparable in CCE syntax, entries marked with ‡ are affected by the early style-classifier bug and are shown for completeness only.

J.5 SFT Series Relative to Base

Table 14 reports the change of every SFT/RFT variant relative to the base model. Plain joint SFT (E2, E3) achieves large style gains (CCE-Full P(LLM) -44.3% and -44.4%) at the cost of functional collapse (-32.0% and -23.4% in HE pass rate). The mechanism is not the objective but the output format: fitting canonical function bodies disrupts the instruct model's habit of producing complete functions, while fitting long expert code from CCE overweights a distribution that HumanEval+ does not share, so the two axes move in opposite directions.

RFT removes this failure mode by writing both criteria into *data selection* rather than into the loss. From the base model we sample 10 candidates per task at temperature 0.9, retain only candidates that pass unit tests, and select the one with the lowest `prob_llm`. Because the targets are complete functions generated by the model itself, the instruct output pattern is preserved and the model does not need to be told to prioritize style through a loss term. This produces the first configuration that improves both axes (E4–E7).

Table 13: Full fine-tuning results on HumanEval+ and CCE (E0–E14). †: CCE under 256-token protocol. ‡: HE style/quality affected by the early evaluation bug. Arrows compare each method to its own training start (Base for SFT, E7 for DPO/SimPO). Best per column in bold.

ID	Method	CCE	Start	HE Pass	HE pass@k	HE Syn.	HE P(LLM)↓	HE Q.	Both↑
E0	Base	×	—	53.8%	83.5%	99.5%	37.6%	0.646	—
E1	SFT function-only†	×	Base	11.4%↓	46.3%↓	97.6%↓	24.6%↓	0.666↑	×
E2	SFT naive joint†	✓	Base	21.8%↓	62.8%↓	99.2%↓	41.1%↑	0.321↓	×
E3	SFT balanced joint†	✓	Base	30.4%↓	59.8%↓	98.8%↓	39.6%↑	0.293↓	×
E4	RFT ×8 + CCE	✓	Base	65.9%↑	87.2%↑	99.6%↑	31.8%↓	0.649↑	✓
E5	RFT ×8 + CCE, lr 5e-5	✓	Base	56.5%↑	84.8%↑	99.6%↑	34.1%↓	0.648↑	✓
E6	RFT ×15, no CCE in RFT selection	×	Base	81.1%↑	89.0%↑	99.6%↑	19.0%↓	0.648↑	✓
E7	**RFT ×15 + CCE**	✓	Base	**82.9%**↑	88.4%↑	**99.8%**↑	17.4%↓	0.646	✓
E8	DPO style (CCE)	✓	E7	82.0%↓	86.6%↓	98.8%↓	16.0%↓	0.649↑	×
E9	**DPO hybrid (CCE+func)**	✓	E7	81.7%↓	87.8%↓	98.8%↓	**14.9%**↓	**0.653**↑	×
E10	ORPO hybrid (from base)	✓	Base	54.4%↑	81.7%↓	99.0%↓	35.8%↓	0.647↑	×
E11	**SimPO hybrid (CCE+func)**	✓	E7	77.8%↓	**89.6%**↑	97.7%↓	**13.2%**↓	**0.659**↑	×
E12	DPO function-only (no CCE)	×	E7	**83.2%**↑	89.0%↑	**99.8%**	17.6%↑	0.646	×
E13	ORPO function-only (no CCE)	×	Base	54.2%↑	86.6%↑	98.8%↓	37.2%↓	0.647↑	×
E14	SimPO function-only (no CCE)	×	E7	82.1%↓	87.8%↓	99.6%↓	17.3%↓	0.647↑	×

ID	Method	CCE-F Syn.	CCE-F P(LLM)↓	CCE-F Q.	CCE-W Syn.	CCE-W P(LLM)↓	CCE-W Q.
E0	Base	95.3%	80.9%	0.605	96.0%	65.9%	0.643
E1	SFT function-only†	81.3%↓	77.8%↓	0.552↓	78.0%↓	64.0%↓	0.547↓
E2	SFT naive joint†	83.0%↓	**36.6%**↓	0.611↑	92.7%↓	**39.5%**↓	0.604↓
E3	SFT balanced joint†	80.0%↓	**36.5%**↓	0.596↓	94.0%↓	**39.9%**↓	0.620↓
E4	RFT ×8 + CCE	84.7%↓	**35.0%**↓	0.631↑	99.3%↑	**39.2%**↓	0.638↓
E5	RFT ×8 + CCE, lr 5e-5	85.7%↓	**34.6%**↓	0.639↑	96.7%↑	**39.3%**↓	**0.654**↑
E6	RFT ×15, no CCE in RFT selection	93.7%↓	67.0%↓	0.643↑	94.0%↓	41.2%↓	0.625↓
E7	**RFT ×15 + CCE**	86.0%↓	**34.0%**↓	0.637↑	**99.3%**↑	**39.6%**↓	0.637↓
E8	DPO style (CCE)	84.7%↓	**33.1%**↓	0.631↓	98.0%↓	**38.1%**↓	0.631↓
E9	**DPO hybrid (CCE+func)**	83.3%↓	33.3%↓	0.621↓	98.0%↓	**37.3%**↓	0.631↓
E10	ORPO hybrid (from base)	**95.3%**	77.8%↓	0.617↑	96.7%↑	62.1%↓	**0.648**↑
E11	**SimPO hybrid (CCE+func)**	83.0%↓	**33.0%**↓	0.619↓	98.7%↓	**36.9%**↓	0.632↓
E12	DPO function-only (no CCE)	84.7%↓	**34.0%**	0.631↓	98.0%↓	**39.1%**↓	0.639↑
E13	ORPO function-only (no CCE)	96.0%↑	81.2%↑	0.605	97.3%↑	66.5%↑	0.644↑
E14	SimPO function-only (no CCE)	84.3%↓	34.3%↑	0.626↓	98.0%↓	38.3%↓	0.633↓

The individual RFT variants then isolate where the remaining trade-off lives. Comparing E4 and E6 separates the two selection criteria: at a fixed CCE ratio, increasing functional repetition from $\times 8$ to $\times 15$ raises HE pass from 65.9% to 81.1% and lowers HE P(LLM) from 31.8% to 19.0%, but leaves CCE-Full P(LLM) at 67.0%, barely below the base 80.9% and far above E4's 35.0%. In other words, additional functional repetition buys short-code performance at the cost of long-code style. The cause is that the classifier scores used for selection are computed on short-code candidates, so the style signal does not transfer across task complexity: the trade-off has not disappeared, only shifted from HumanEval+ to CCE-Full.

Adding CCE closes this gap. E7 combines $\times 15$ functional repetition with CCE and is the only configuration that simultaneously improves HE pass ($+29.1\%$), HE P(LLM) (-20.1%), CCE-Full P(LLM) (-47.0%), and CCE-Worse P(LLM) (-26.3%), with CCE-Full quality score (0.637) above base (0.605). The only axis on which E7 trails E6 is CCE-Full syntax rate (86.0% vs. 93.7%), and this gap is a truncation artifact rather than a functional regression. Both base and fine-tuned models are evaluated under the same 512-token generation limit, but fine-tuned models produce substantially longer, more engineering-style outputs (more logging, defensive branches, and documentation) and therefore reach the limit far more often, when a long output is cut mid-string or mid-bracket, the parser flags it as a syntax error even though the model was writing valid code. This reading is confirmed by the two protocols: raising the limit from 256 to 512 tokens moves base only from 91.7% to 95.3%, but moves E7 from 77.3% to 86.0%, i.e., most of the gap is recoverable by giving the model more room to finish. The residual difference is therefore a measurement artifact of the evaluation budget, not evidence that the fine-tuned model writes less valid code.

Table 14: Change of SFT/RFT variants relative to the base model (E0). Negative style deltas indicate improvement. Best per column in bold.

ID	Recipe	ΔHE Pass	ΔHE Style	ΔCCE-F Style	ΔCCE-W Style	ΔCCE-F Syn.
E1	Function-only	-0.424	-0.130	-0.031	-0.019	-0.140
E2	Joint (HE+CCE)	-0.320	$+0.035$	$\mathbf{-0.443}$	$\mathbf{-0.263}$	-0.123
E3	Balanced joint (HE×8)	-0.234	$+0.020$	$\mathbf{-0.444}$	$\mathbf{-0.260}$	-0.153
E4	RFT ×8 + CCE	$\mathbf{+0.121}$	-0.058	-0.460	-0.267	-0.107
E5	RFT ×8 + CCE, lr 5e-5	$+0.027$	-0.035	-0.463	-0.266	-0.097
E6	RFT ×15 (no CCE)	$\mathbf{+0.273}$	$\mathbf{-0.186}$	-0.139	-0.247	-0.017
E7	**RFT ×15 + CCE**	$\mathbf{+0.291}$	$\mathbf{-0.201}$	$\mathbf{-0.470}$	$\mathbf{-0.263}$	-0.093

J.6 Preference Optimization Relative to Its Starting Point

Table 15 reports the change of every preference-optimization variant relative to its own training start: E8, E9, E11, E12, and E14 start from E7, while E10 and E13 start from base. The comparison separates two questions: does the method improve over its own start, and does it outperform the base? DPO and SimPO consistently sharpen the style signal inherited from E7, while ORPO from base and function-only DPO (E12) stay at or near their starting point.

Table 15: Preference-optimization variants relative to base and to their own start. Style deltas are relative to the start. Negative style deltas indicate improvement. Best per column in bold.

ID	Method / Data	Start	ΔHE Pass vs base	ΔHE Style vs base	ΔHE Style vs start	ΔCCE-F Style vs start	Note
E8	DPO / CCE	E7	+0.282	−0.216	−0.014	−0.009	Small style gain
E9	**DPO / CCE+function**	E7	+0.279	**−0.226**	**−0.025**	**−0.007**	Balanced best
E10	ORPO / CCE+function	base	+0.006	−0.018	—	−0.031	Weak from base
E11	**SimPO / CCE+function**	E7	+0.240	**−0.244**	**−0.043**	**−0.010**	Most human-like
E12	DPO / function (no CCE)	E7	+0.295	−0.200	+0.001	+0.001	≈ start
E13	ORPO / function (no CCE)	base	+0.004	−0.003	—	+0.002	≈ base
E14	SimPO / function (no CCE)	E7	+0.284	−0.203	−0.002	+0.003	≈ start

J.7 With and Without CCE: The Core Ablation

Table 16 isolates the contribution of CCE at both stages. At the SFT stage, with functional repetition held fixed, adding CCE (E6 → E7) changes HE pass by only +1.8 pt and HE style by −1.5 pt, but reduces CCE-Full style by 33.1 pt. At the preference stage, adding CCE style pairs on top of the same functional pairs yields much smaller marginal gains: DPO improves CCE-Full style by 0.8 pt, SimPO by 1.0 pt, and ORPO by 3.3 pt, while HE style improves by 1.5–4.1 pt.

Table 16: Core ablation: with vs. without CCE at the SFT stage (top) and the preference stage (bottom).

Stage	Recipe	HE Pass	HE P(LLM)↓	CCE-F P(LLM)↓	CCE-W P(LLM)↓
SFT	E6: RFT ×15, no CCE	81.1%	19.0%	67.0%	41.2%
SFT	E7: RFT ×15, with CCE	**82.9%**	**17.4%**	**34.0%**	**39.6%**
SFT	Contribution	+1.8 pt	−1.5 pt	−33.1 pt	−1.7 pt
Method	With CCE / Without CCE	HE Pass	HE P(LLM)↓	CCE-F P(LLM)↓	CCE-W P(LLM)↓
DPO	E9 (with) vs. E12 (without)	81.7% / **83.2%**	**14.9%** / 17.6%	**33.3%** / 34.0%	**37.3%** / 39.1%
SimPO	E11 (with) vs. E14 (without)	77.8% / **82.1%**	**13.2%** / 17.3%	**33.0%** / 34.3%	**36.9%** / 38.3%
ORPO	E10 (with) vs. E13 (without)	**54.4%** / 54.2%	**35.8%** / 37.2%	**77.8%** / 81.2%	**62.1%** / 66.5%

J.8 Discussion of Findings

Plain SFT trades off the two axes. E1–E3 confirm the trade-off in its most direct form: improving style degrades function (E2, E3) or improving function degrades style (E1). The mechanism is not the objective but the output format: fitting canonical bodies disrupts the instruct pattern, while fitting long expert code from CCE overweights a distribution that HumanEval+ does not share.

RFT moves the trade-off to where the style signal is defined. RFT writes both criteria into data selection rather than into the loss, so the model is trained on its own complete functions. This removes the trade-off on HumanEval+ (E4–E7) but only partially on CCE: E6 (RFT ×15, no CCE) reaches HE pass 0.811 and HE style 0.190, yet CCE-Full style stalls at 0.670, far above the E4 value of 0.350. Comparing E4 and E6 isolates the cause: at fixed CCE ratio, increasing functional repetition from ×8 to ×15 buys short-code functionality at the cost of long-code style, because the classifier scores used for selection were computed on short-code candidates and therefore do not transfer. The trade-off has not vanished, it has shifted from HumanEval+ to CCE-Full.

CCE closes the trade-off by grounding the style signal in long code. E7 combines ×15 functional repetition with CCE and is the only configuration that improves HE pass, HE style, CCE-Full style, and CCE-Worse style simultaneously. Its CCE-Full quality (0.637) exceeds the base (0.605), so the improvement is not a syntactic artifact. The only axis on which E7 trails E6 is CCE-Full syntax (0.860 vs. 0.937), and the gap is explained by the 512-token protocol and by the fact that fine-tuned code is longer and more engineering-like, hence more prone to truncation. This pattern

matches Prediction P2 in Section 3: style detectability is bounded by task complexity, and only style supervision drawn from the same task distribution can close the gap.

Preference optimization amplifies but does not create style. DPO and SimPO sharpen the style signal that E7 already carries: E9 pushes HE style to 0.149 and CCE-Full style to 0.333, and E11 reaches 0.132 and 0.330. The gain over E7 is small (-0.025 and -0.043 in HE style) but consistent, and comes at a modest functional cost (-1.2 pt and -5.1 pt in HE pass). ORPO from base (E10) leaves CCE-Full style at 0.778, essentially unchanged from base, even though it uses the same preference data. The starting point therefore bounds what preference optimization can achieve.

CCE is the only new style information at the preference stage. E12 and E14, trained on functional pairs without CCE, are essentially equivalent to their start E7 (HE style $0.176/0.173$, CCE-Full style $0.340/0.343$). The reason is that the chosen candidates in the functional pairs are exactly the RFT samples E7 was trained on, so their style signal has already been absorbed. CCE style pairs are the only source of new information. This is why the marginal value of CCE is large at the SFT stage (-33.1 pt on CCE-Full style) and small but directionally consistent at the preference stage (-0.8 to -3.3 pt).

Recommended recipes. Based on the above, we recommend E7 for the simplest single-stage pipeline that improves both axes; E9 for balanced deployment where HE quality matters; E11 for style-first settings; and E12 or E7 for function-first settings. E10 and E13 are not recommended as standalone methods, since preference optimization without a style-aware SFT start does not recover the same effect.

K Complete Generalization Experiment Settings

Generalization on HumanEval+ and EvoEval:

- Training data: HumanEval+ + EvoEval mixed (Human or Qwen-generated)
- Train/validation split: 8:2
- Model: SlotGCN + All (same configuration as in Section 4.2)
- Early stopping: triggered when validation accuracy reaches 83.12%
- Overfitting observation: Without early stopping, accuracy continues to climb beyond 90%, but prediction accuracy on code generated by other models drops, indicating that the classifier is detecting the style of a single LLM rather than LLM style in general

Generalization on LPcodedec:

- LPcodedec dataset: 21,355 code snippets across 4 programming languages (C, C++, Java, Python)
- Human code source: GitHub (4,271 samples)
- LLM-rewritten sources: ChatGPT, Gemini-Pro, WizardCoder, DeepSeek-Coder
- Task paradigm: Rewrite detection (LLM polishing/rewriting existing code)
- Classifier: Same SlotGCN + All configuration
- Result: Validation accuracy $\sim$55% (does not converge)

Paradigm difference analysis: Rewritten code preserves most of the original AST structure: function definitions, control flow, and nesting depth remain largely unchanged, with only variable names, comments, and minor syntactic style variations. Our Augmented AST classifier relies on structural differences for discrimination, when such structural information is flattened, the classifier loses effective discriminative signals. This result is consistent with the motivation in Section 3.3 for choosing graph-based representations as the core feature: graph structures preserve more structural signal and pay less attention to semantics.

L Detailed Comparison with Existing Style Benchmarks

Existing style-related benchmarks include LPcodedec (Park et al., 2025) for rewrite detection, AIGCodeSet (Demirok & Kutlu, 2025) and CodeMirage (Guo et al., 2025) for source binary classification, and HybridCodeAuthorship (Patterson et al., 2026) for line-level authorship attribution. Multi-AIGCD (Demirok et al., 2025) and DroidCollection (Orel et al., 2025) target cross-scenario generalization. These benchmarks are inadequate for code-style evaluation tasks for three reasons. First, their uncertainty in enforcing functional equivalence means style differences can be confounded with algorithmic or logical variations. Second, they lack systematic task difficulty stratification, making it impossible to isolate the effect of code complexity on style. Third, their task paradigms are not designed for attribution, as they lack complete AST node annotations to support fine-grained perturbation experiments. Our CCE benchmark addresses all three gaps: (1) weak functional equivalence via code decomposition, logic extraction, and LLM regeneration; (2) dual-difficulty stratification with CCE-Full (industrial) and CCE-Worse (educational/competition); and (3) full AST preservation enabling node-level perturbation analysis.

M Justification of Classifier as Proxy for Stylistic Naturalness

The source label y (an objective fact) and stylistic naturalness $s(c)$ (the latent variable of our interest) are not identical: a human-written code that is unusually rigid may deviate from the human distribution center toward the LLM side, while an LLM-generated code may occasionally exhibit human-like features. However, a meaningful proxy relationship exists under the design of our benchmark.

In CCE-Full, human code is sampled from *high-quality, well-maintained open-source Python libraries*, code that has been reviewed, tested, and refined by experienced developers over time. This distribution represents a *de facto standard of professional software style*. Under weak functional equivalence (i.e., corresponding tasks with identical specifications), semantic signals are weakened, and the model is forced to exploit systematic differences in code samples: stable discriminators such as naming conventions, control-flow organization, nesting depth, and documentation density. These features are precisely what constitutes code style from a software engineering perspective. Consequently, the binary classification boundary between human and LLM code in CCE-Full closely approximates the boundary between "professionally-styled" and "model-styled" code.

In contrast, CCE-Worse draws human code from algorithm competition and educational settings, where stylistic consistency is weaker and the code is often written for pedagogy or performance rather than maintainability. The classification boundary in this setting carries a less direct interpretation as style naturalness. The systematic comparison between the two difficulty strata allows us to isolate the effect of task complexity on style detectability—demonstrating that the classifier's discriminative power correlates with the richness of stylistic signals in the data.

Empirical validation of this proxy relationship is provided in Section 5.4, where the 14-dimensional feature analysis confirms that the classifier relies on stable discriminators such as `docstring_length`, `pass_count`, and `nested_scope_depth`, all of which are widely recognized as indicators of code quality in software engineering practice.