UCoder: Unsupervised Code Generation by Internal Probing of Large Language Models

Jiajun Wu[1], **Jian Yang**[1]*, **Wei Zhang**[1], **Lin Jing**[1], **Yuqing Ma**[1], **Ensheng Shi**[2], **Yuchi Ma**[2], **Zhoujun Li**[1], **Xianglong Liu**[1]
[1]Beihang University; [2]Huawei;
{wuyuverse,jiayang}@buaa.edu.cn

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in code generation tasks. However, their effectiveness heavily relies on supervised training with extensive labeled (e.g., question-answering pairs) or unlabeled datasets (e.g., code snippets), which are often expensive and difficult to obtain at scale. To address this limitation, this paper introduces a method **IPC**, an unsupervised framework that leverages **I**nternal **P**robing of LLMs for **C**ode generation without any external corpus, even unlabeled code snippets. We introduce the problem space probing, test understanding probing, solution space probing, and knowledge consolidation and reinforcement to probe the internal knowledge and confidence patterns existing in LLMs. Further, IPC identifies reliable code candidates through self-consistency mechanisms and representation-based quality estimation to train UCoder (coder with unsupervised learning). We validate the proposed approach across multiple code benchmarks, demonstrating that unsupervised methods can achieve competitive performance compared to supervised approaches while significantly reducing the dependency on labeled data and computational resources. Analytic experiments reveal that internal model states contain rich signals about code quality and correctness, and that properly harnessing these signals enables effective unsupervised learning for code generation tasks, opening new directions for training code LLMs in resource-constrained scenarios.

1 Introduction

Large language models (LLMs) have demonstrated strong capabilities in code generation, producing functional code from natural language descriptions. This progress has attracted substantial interest from both academia and industry due to its practical impact on software development. Closed-source

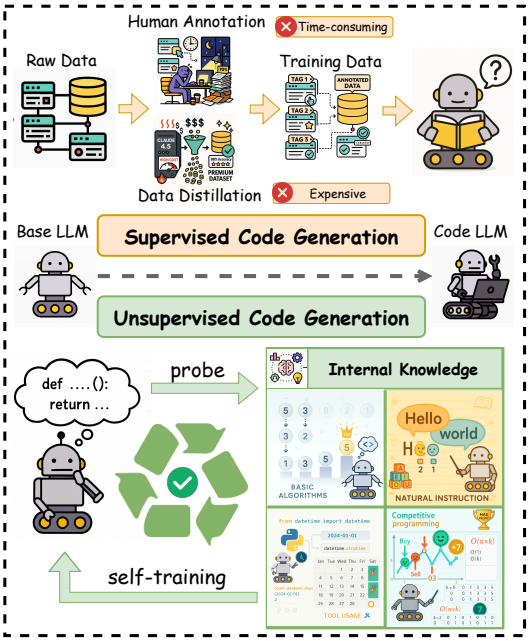

Figure 1: Comparison between supervised and unsupervised paradigms for code generation.

LLMS such as GPT-5 (OpenAI, 2025) and Claude-4.5 (Anthropic, 2025) can generate file-level code with high accuracy, while open-source alternatives, including StarCoder (Li et al., 2023; Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024), and QwenCoder (Hui et al., 2024) have emerged as competitive solutions for code intelligence.

Most existing approaches for improving code generation rely on supervised instruction tuning, where LLMs are fine-tuned on curated problem-solution pairs annotated by human experts or LLM-based annotated. However, creating high-quality instruction data requires substantial human effort in problem design, implementation, and verification, with costs increasing as model capabilities advance. The recent (Yue et al.; Ye et al., 2025; Chu et al., 2025) works emphasize that pre-training brings the knowledge, and post-training is weak at knowledge integration but focuses on knowledge utilization and alignment. *These challenges motivate a fundamental question: Can LLMs au-*

*Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 5642–5655
July 2-7, 2026

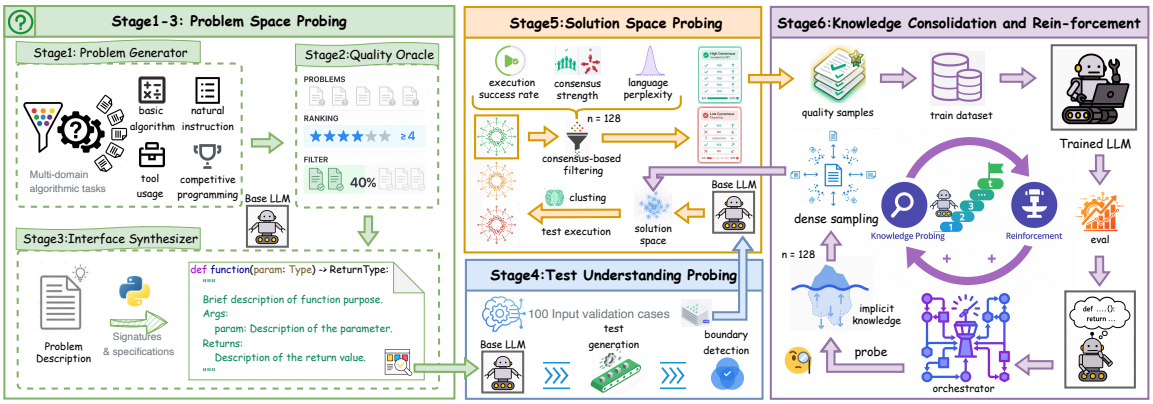

Figure 2: Overview of the proposed six-stage self-bootstrapping framework for unsupervised code generation.

tonomously improve their generation capabilities using post-training without any external corpus, relying only on pre-trained knowledge?

In this work, we introduce an unsupervised framework that performs **I**nternal **P**robing of LLMs for **C**ode generation, enabling post-training without external corpora or human-annotated instruction data. Our approach exploits latent programming knowledge in LLMs and uses execution feedback as a scalable, deterministic supervision signal grounded in program semantics. We implement a six-stage self-bootstrapping process that generates diverse programming tasks, synthesizes test suites, samples candidate solutions, and applies execution-driven consensus clustering to identify correct implementations. High-consensus solutions are iteratively consolidated as training data, forming a feedback loop that progressively improves model performance.

Despite using no external data, UCoder achieves comparable performance to the supervised baseline across multiple benchmarks. The primary contributions of this work are:

- We successfully probe latent programming knowledge in LLMs by forcing models to generate programming problems and their solutions, then identify correct solutions by finding clusters of similar implementations. Then, the self-training method progressively improves the LLM by reinforcing solutions.
- Based on the self-generated data from the unsupervised framework using internal probing of LLMs (IPC) without any external data, UCoder (7B, 14B, 32B) achieves performance competitive with supervised baselines.
- We provide empirical analysis showing that self-generated data maintains rich lexical, semantic, and structural diversity, while consensus-based selection improves solution quality and exhibits inverse scaling behavior.

2 Unsupervised Code Generation

2.1 Task Definition

2.1.1 Supervised Code Generation

Supervised code generation is formulated as a sequence-to-sequence learning task. Given a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, the model parameters are optimized by maximizing log-likelihood:

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{N} \log p_\theta(y_i|x_i) \quad (1)$$

where $x_i \in \mathcal{X}$ denotes a natural language query, $y_i \in \mathcal{Y}$ represents the corresponding reference implementation, and $p_\theta(y|x)$ is the conditional distribution over code sequences parameterized by θ. We adopt Pass@k (Chen et al., 2021a) to evaluate code correctness, which measures the probability that at least one of k independently sampled solutions passes all test cases.

2.1.2 Unsupervised Code Generation

Unsupervised code generation aims to improve LLM code generation capabilities without human-annotated supervision. Given an initial model $M_0 : \mathcal{X} \to \mathcal{Y}$, the objective is to develop a self-improvement algorithm $\mathcal{A}$ producing an enhanced model $M^* = \mathcal{A}(M_0)$ such that $\texttt{Pass@}k(M^*) > \texttt{Pass@}k(M_0)$ on held-out test sets, without access to paired training data $(x, y) \in \mathcal{X} \times \mathcal{Y}$. This presents three fundamental challenges: (1) **Problem Space Construction.** Automatically generating diverse programming problems with appropriate difficulty distributions while maintaining semantic clarity; (2) **Unsupervised Correctness Verification.** Assessing functional correct-

ness without reference implementations; (3) **Self-Bootstrapping Signal Construction.** Extracting reliable training signals from noisy candidates while ensuring iterative stability. We address these through an execution-driven consensus mechanism coupled with a self-bootstrapping framework, detailed in the following sections.

2.2 Probing Internal Knowledge in LLMs

LLMs encode extensive programming knowledge through pre-training, yet this knowledge remains implicit and difficult to elicit. We propose a six-stage framework to surface and reinforce these latent capabilities, as shown in Figure 2. First, we probe the problem space (Stages 1–3) by prompting the model to generate algorithmic problems with complete specifications, revealing its understanding of programming paradigms and data structures. Representative examples of problem generation, difficulty assessment, and solution skeleton construction are illustrated in Figure 3.

We then assess semantic understanding (Stage 4) by generating approximately 100 test cases per problem to identify boundary conditions and edge cases. At the core (Stage 5), we probe the solution space via dense sampling, where execution-driven consensus clustering reveals that correct implementations form tight clusters while incorrect ones are dispersed. We quantify solution quality using execution success rate $e(r)$, consensus strength $s(r)$, and code fluency $f(r)$. Finally (Stage 6), we consolidate high-consensus samples through supervised fine-tuning, reinforcing correct patterns. The process forms a positive feedback loop: at iteration t, the improved LLM M_t produces higher-quality candidates, enabling more reliable selection and further strengthening M_{t+1}. Our experimental results in subsection 3.2 validate that pre-trained models already contain the knowledge required to solve target tasks in implicit form.

2.3 Execution-Driven Consensus Clustering

Our approach exploits that correctness is singular while incorrectness is diverse: correct implementations produce identical outputs, but incorrect ones fail heterogeneously. This clustering structure allows the maximum-consensus cluster to indicate correctness without ground truth, formalized in Theorem 2.4 and validated in subsection 4.2.

2.3.1 Definitions

Consensus clustering has three definitions.

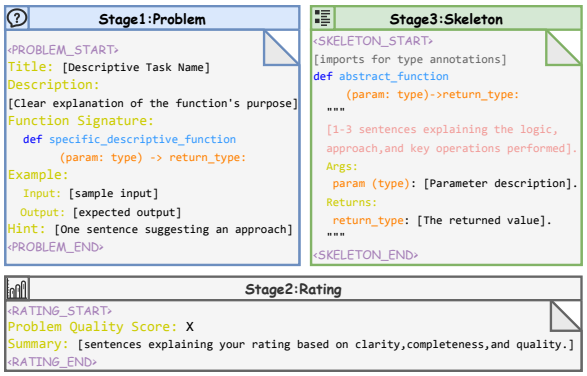

Figure 3: Problem space probing proceeds through three stages: problem generation with function signatures and input-output contracts, difficulty rating assessment and categorization, and solution skeleton generation with implementation structure.

Definition 2.1 (Execution Signature)**.** Given candidates $R = \{r_1, \ldots, r_n\}$ and tests $T = \{t_1, \ldots, t_m\}$, define $\text{Exec} : R \times T \to \mathcal{O} \cup \{\perp\}$ as the execution function that returns the output of running candidate r_i on test input t_j, or $\perp$ if execution fails (e.g., runtime error or timeout). The execution signature of r_i on T is

$$\sigma(r_i; T) = \bigoplus_{j=1}^{m} \text{Exec}(r_i, t_j), \tag{2}$$

where $\bigoplus$ denotes ordered concatenation. Two candidates r_i, r_j share the same signature $\sigma(r_i; T) = \sigma(r_j; T)$ if and only if they produce identical outputs on all test inputs, indicating behaviorally equivalent implementations.

Definition 2.2 (Consensus Clusters)**.** If $\sigma(r_i, T) = \sigma(r_j, T)$, we can regard the r_i and r_j as equivalent solution. Given the value of $\sigma(r_i, T)$, we can partition R into clusters $\mathcal{C} = \{C_1, \ldots, C_\ell\}$ of behaviorally identical candidates.

Definition 2.3 (Quality Metrics)**.** Each candidate $r \in R$ is scored by:

$$\begin{aligned} e(r) &= \frac{|\{t \in T : \text{Exec}(r, t) \neq \perp\}|}{|T|}, \\ s(r) &= |\{r' \in R : \sigma(r') = \sigma(r)\}|, \\ f(r) &= \exp\left(-\frac{1}{|r|}\sum_{i=1}^{|r|} \log p(x_i \mid x_{<i})\right), \end{aligned} \tag{3}$$

where $e(r)$ measures execution success, $s(r)$ consensus strength, and $f(r)$ code fluency.

2.3.2 Hierarchical Selection

We select the valid candidates using three criteria:

Model	HumanEval		MBPP		BCB-Complete		BCB-Instruct		LiveCode-Bench	FullStack-Bench
	HE	HE+	MBPP	MBPP+	Full	Hard	Full	Hard		
6B+ LLMs										
CodeLlama-7B-Instruct	40.9	33.5	39.9	33.6	25.7	4.1	21.9	3.4	7.1	25.40
DS-Coder-6.7B-Instruct	74.4	71.3	74.9	65.6	43.8	15.5	35.5	10.1	15.5	40.16
OpenCoder-8B-Instruct	83.5	78.7	79.1	69.0	50.9	18.9	**43.2**	**18.2**	**23.2**	41.08
Qwen2.5-Coder-7B	61.6	53.0	76.9	62.9	45.8	16.2	-	-	-	-
Qwen2.5-Coder-7B-Instruct	**88.4**	**84.1**	83.5	71.7	48.8	20.3	41.0	**18.2**	18.2	47.95
UCoder-7B	83.5	76.8	**85.2**	**72.2**	**52.0**	**22.3**	41.1	15.5	22.9	**51.27**
13B+ LLMs										
CodeLlama-13B-Instruct	40.2	32.3	60.3	51.1	31.7	6.8	28.5	9.5	6.1	27.00
Starcoder2-15B-Instruct-v0.1	67.7	60.4	78.0	65.1	45.1	14.9	37.2	11.5	12.1	42.68
DS-Coder-V2-Lite-Instruct	81.1	75.6	82.8	70.4	47.6	18.2	36.8	16.2	**24.3**	-
Qwen2.5-Coder-14B	64.0	57.9	81.0	66.7	51.8	22.3	-	-	-	-
Qwen2.5-Coder-14B-Instruct	**89.6**	**87.2**	86.2	72.8	**56.7**	**29.7**	**48.4**	**22.2**	23.4	**55.28**
UCoder-14B	87.8	81.1	**86.5**	**74.3**	53.9	24.3	40.9	16.2	20.6	52.52
32B+ LLMs										
CodeLlama-34B-Instruct	48.2	40.2	61.1	50.5	35.6	10.8	29.0	8.8	8.4	27.56
DS-Coder-33B-Instruct	81.1	75.0	80.4	70.1	51.1	20.9	42.0	17.6	21.3	48.19
DS-Coder-V2-Instruct	85.4	82.3	89.4	75.1	**59.7**	29.7	48.2	24.3	27.9	56.37
Qwen2.5-Coder-32B	65.9	60.4	83.0	68.2	53.6	26.4	-	-	-	-
Qwen2.5-Coder-32B-Instruct	**92.7**	**87.2**	**90.2**	75.1	58.0	**33.8**	**49.6**	**27.0**	**31.4**	**56.88**
UCoder-32B	89.0	82.9	89.7	**75.7**	55.4	27.7	45.7	17.6	21.4	53.35
Closed-APIs										
GPT-4o-2024-08-06	92.1	86.0	86.8	72.5	-	36.5	50.1	25.0	34.6	58.89
Claude-3.5-Sonnet-20241022	92.1	86.0	91.0	74.6	58.6	35.1	46.8	25.7	31.6	60.70

Table 1: Performance comparison of **Qwen2.5-Coder** Base and Instruct models with our iterative SFT models across code generation benchmarks. All metrics represent Pass@1 execution rates (%). Complete split is reported for Base models and Instruct split for Instruct models. **Bold** indicates best performance within each size category. "-" denotes unavailable or inapplicable results.

(1) Reliability Filtering. Candidates with low execution success are removed (threshold $\rho = 0.8$): $R' = \{r \in R : e(r) \geq \rho\}$.
(2) Consensus Selection. We select the largest non-trivial cluster: $C^* = \arg\max_{C \in \mathcal{C}', |C| \geq \tau} |C|$
(3) Intra-Cluster Selection. Within C^*, we choose $r^* = \arg\max_{r \in C^*} \langle e(r), -f(r) \rangle$.

2.3.3 Theoretical Guarantee

Theorem 2.4 (Consensus Convergence)**.** Let $R = \{r_1, \ldots, r_n\}$ be n candidates sampled independently from a model, and let T denote a set of unit tests. Assume that at least k candidates in R are functionally correct with probability at least $1 - \delta$, and that any pair of incorrect implementations produces identical outputs on a single test with probability at most $p < 1$.

If the test set size satisfies

$$|T| \geq \frac{\log(n/k)}{-\log p},$$

then the largest consensus cluster $C_{\max}$ contains only correct implementations with probability at least

$$P(C_{\max} \text{ is correct}) \geq 1 - \delta - n^2 p^{|\mathcal{T}|}.$$

2.4 Iterative Self-Training

We formalize the iterative self-training procedure and explain why it yields consistent improvement.

Definition 2.5 (Iterative Update)**.** At iteration t, we construct training set $\mathcal{D}_t = \{(q_i, r_i^*)\}$, where each r_i^* is selected via consensus from n candidates sampled from $\mathcal{M}_t$, and update:

$$\theta_{t+1} = \arg\max_{\theta} \sum_{(q, r^*) \in \mathcal{D}_t} \log p_\theta(r^* \mid q). \tag{4}$$

Why Self-Training Improves Performance? Iterative self-training is effective because consensus selection acts as a quality filter. Let $Q(r) \in [0, 1]$ denote candidate quality. For n independent samples, random selection yields expected quality $\mathbb{E}_{r \sim \mathcal{M}_t}[Q(r)]$, whereas consensus selection favors correct implementations that cluster by execution behavior, achieving (for some $\Delta > 0$):

$$\mathbb{E}[Q(r^*)] = \mathbb{E}_{r \sim \mathcal{M}_t}[Q(r)] + \Delta, \tag{5}$$

Optimizing on $\mathcal{D}_t$ shifts the model toward higher-quality samples. As $\mathcal{M}_{t+1}$ increases $p_\theta(r^* \mid q)$ for above-average outputs:

$$\mathbb{E}_{r \sim \mathcal{M}_{t+1}}[Q(r)] \geq \mathbb{E}_{r \sim \mathcal{M}_t}[Q(r)], \tag{6}$$

Iter	HumanEval		MBPP		BCB-Complete		BCB-Instruct		LiveCode-Bench	FullStack-Bench
	HE	HE+	MBPP	MBPP+	Full	Hard	Full	Hard		
Ucoder-7B										
0	77.4	67.1	72.0	63.0	44.4	15.5	34.6	14.9	13.0	40.2
1	81.7$_{+4.3}$	74.4$_{+7.3}$	72.0$_{0.0}$	63.0$_{0.0}$	51.3$_{+6.9}$	**23.0**$_{+7.5}$	**42.2**$_{+7.6}$	**20.9**$_{+6.0}$	15.3$_{+2.3}$	48.2$_{+7.9}$
2	**84.1**$_{+6.7}$	**77.4**$_{+10.3}$	79.1$_{+7.1}$	66.4$_{+3.4}$	44.7$_{+0.3}$	14.9$_{-0.6}$	35.8$_{+1.2}$	12.2$_{-2.7}$	14.5$_{+1.5}$	40.2$_{-0.1}$
3	**84.1**$_{+6.7}$	76.8$_{+9.7}$	81.2$_{+9.2}$	69.0$_{+6.0}$	44.4$_{0.0}$	15.5$_{0.0}$	34.5$_{-0.1}$	13.5$_{-1.4}$	21.4$_{+8.4}$	40.2$_{0.0}$
4	**84.1**$_{+6.7}$	**77.4**$_{+10.3}$	79.1$_{+7.1}$	66.4$_{+3.4}$	44.7$_{+0.3}$	14.9$_{-0.6}$	35.8$_{+1.2}$	12.2$_{-2.7}$	14.5$_{+1.5}$	40.2$_{-0.1}$
5	81.7$_{+4.3}$	75.0$_{+7.9}$	83.9$_{+11.9}$	71.2$_{+8.2}$	**52.2**$_{+7.8}$	19.6$_{+4.1}$	40.7$_{+6.1}$	14.2$_{-0.7}$	20.6$_{+7.6}$	50.0$_{+9.7}$
6	83.5$_{+6.1}$	76.8$_{+9.7}$	**85.2**$_{+13.2}$	**72.2**$_{+9.2}$	52.0$_{+7.6}$	22.3$_{+6.8}$	41.1$_{+6.5}$	15.5$_{+0.6}$	**22.9**$_{+9.9}$	**51.3**$_{+11.0}$
Ucoder-14B										
0	83.5	76.8	75.9	64.0	53.3	23.6	43.2	**16.2**	22.1	50.1
1	85.4$_{+1.9}$	77.4$_{+0.6}$	**87.8**$_{+11.9}$	73.3$_{+9.3}$	53.1$_{-0.2}$	21.6$_{-2.0}$	41.0$_{-2.2}$	14.2$_{-2.0}$	17.6$_{-4.5}$	**53.6**$_{+3.5}$
2	84.8$_{+1.3}$	76.2$_{-0.6}$	84.9$_{+9.0}$	70.6$_{+6.6}$	50.4$_{-2.9}$	23.6$_{0.0}$	40.9$_{-2.3}$	15.5$_{-0.7}$	19.1$_{-3.0}$	49.9$_{-0.2}$
3	84.8$_{+1.3}$	78.0$_{+1.2}$	83.6$_{+7.7}$	72.2$_{+8.2}$	51.6$_{-1.7}$	23.0$_{-0.6}$	41.8$_{-1.4}$	15.5$_{-0.7}$	18.3$_{-3.8}$	49.8$_{-0.3}$
4	84.8$_{+1.3}$	78.0$_{+1.2}$	84.1$_{+8.2}$	72.8$_{+8.8}$	51.8$_{-1.5}$	18.2$_{-5.4}$	41.1$_{-2.1}$	12.8$_{-3.4}$	22.1$_{0.0}$	50.4$_{+0.3}$
5	**87.8**$_{+4.3}$	**81.1**$_{+4.3}$	86.5$_{+10.6}$	**74.3**$_{+10.3}$	**53.9**$_{+0.6}$	**24.3**$_{+0.7}$	40.9$_{-2.3}$	**16.2**$_{0.0}$	20.6$_{-1.5}$	52.5$_{+2.4}$
6	87.2$_{+3.7}$	80.5$_{+3.7}$	84.4$_{+8.5}$	71.7$_{+7.7}$	53.3$_{0.0}$	23.0$_{-0.6}$	**43.5**$_{+0.3}$	**16.2**$_{0.0}$	**22.9**$_{+0.8}$	51.6$_{+1.5}$
Ucoder-32B										
0	86.0	78.0	86.2	72.2	54.8	**28.4**	44.6	**18.9**	22.1	53.0
1	87.8$_{+1.8}$	82.3$_{+4.3}$	**89.9**$_{+3.7}$	**75.9**$_{+3.7}$	55.4$_{+0.6}$	23.6$_{-4.8}$	44.5$_{-0.1}$	17.6$_{-1.3}$	17.6$_{-4.5}$	54.3$_{+1.2}$
2	86.6$_{+0.6}$	81.1$_{+3.1}$	88.4$_{+2.2}$	74.6$_{+2.4}$	**56.2**$_{+1.4}$	23.0$_{-5.4}$	44.8$_{+0.2}$	**18.9**$_{0.0}$	16.0$_{-6.1}$	52.4$_{-0.6}$
3	87.8$_{+1.8}$	81.1$_{+3.1}$	88.9$_{+2.7}$	74.6$_{+2.4}$	54.3$_{-0.5}$	22.3$_{-6.1}$	43.8$_{-0.8}$	16.9$_{-2.0}$	19.8$_{-2.3}$	**54.7**$_{+1.7}$
4	**89.0**$_{+3.0}$	**82.9**$_{+4.9}$	89.7$_{+3.5}$	75.7$_{+3.5}$	55.4$_{+0.6}$	27.7$_{-0.7}$	**45.7**$_{+1.1}$	17.6$_{-1.3}$	21.4$_{-0.7}$	53.4$_{+0.3}$
5	88.4$_{+2.4}$	81.7$_{+3.7}$	87.8$_{+1.6}$	73.3$_{+1.1}$	54.5$_{-0.3}$	24.3$_{-4.1}$	43.8$_{-0.8}$	**18.9**$_{0.0}$	**22.9**$_{+0.8}$	53.3$_{+0.2}$
6	88.4$_{+2.4}$	82.3$_{+4.3}$	89.2$_{+3.0}$	74.1$_{+1.9}$	54.0$_{-0.8}$	23.6$_{-4.8}$	44.1$_{-0.5}$	16.9$_{-2.0}$	**22.9**$_{+0.8}$	53.8$_{+0.7}$

Table 2: Performance (Pass@1) across iterative SFT rounds at different model scales using Qwen2.5-Coder as the base model. Orange-highlighted rows show Iter 0 (initial model trained on seed data), while subsequent iterations use self-generated synthetic data. Blue-highlighted rows indicate the best-performing iteration for each scale. Bold denotes best performance, underline denotes second-best performance for each metric within each scale, and subscripts show differences from Iter 0 (*green* for improvement, *red* for decline).

where it induces a positive feedback loop where improved models generate higher-quality candidates and more reliable training signals.

3 Experiments

3.1 Training and Evaluation Details

Model Configuration. We experiment with Qwen2.5-Coder(Hui et al., 2024) models at 7B, 14B, and 32B scales, starting from base checkpoints without prior instruction tuning and applying identical self-bootstrapping procedures across all scales for fair comparison.

Training Hyperparameters. We employ consistent training settings across experiments. Models are fine-tuned for 3 epochs per iteration using AdamW with a learning rate of 5e-6 and a cosine decay schedule. We use a batch size of 128 with gradient accumulation to fit memory constraints.

Evaluation Benchmarks. We evaluate on six benchmarks: **HumanEval** (Chen et al., 2021b) and **MBPP/MBPP+** (Austin et al., 2021) assess classic Python programming; **LiveCodeBench** (Jain et al., 2024) provides contamination-free competitive programming problems; **BigCodeBench (BCB)** (Zhuo et al., 2024) evaluates function completion with broader context and API usage (both Complete and Instruct variants); and **FullStackBench** (Liu et al., 2024) covers diverse real-world scenarios. We report Pass@1 accuracy with execution-based validation; solutions must pass all test cases.

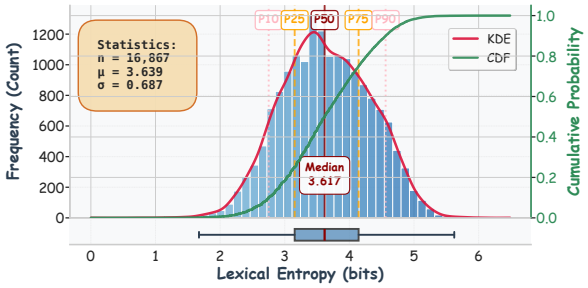

Figure 4: Lexical entropy distribution of 16,867 generated problems. Histogram with KDE shows per-problem entropy; CDF (green) and boxplot show cumulative coverage.

3.2 Main Results

Table 1 demonstrates that unsupervised self-bootstrapping achieves performance comparable to supervised instruction tuning across diverse code generation benchmarks. Compared with other

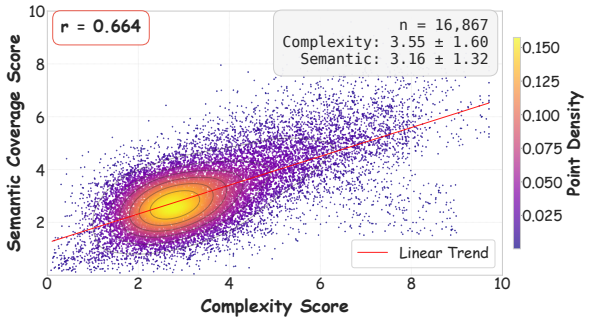

Figure 5: Complexity versus semantic coverage distribution. Color encodes density; red line shows linear trend ($r = 0.664$). Score definitions in Appendix B.1.

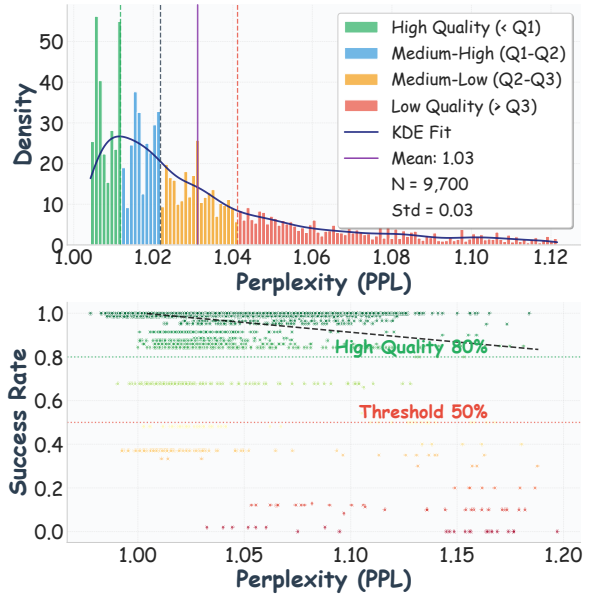

Figure 6: Quality distribution characterization. Top: perplexity distribution (computed per-token over generated code) across 9,700 samples with quartile stratification and KDE overlay (truncated at 1.12). Bottom: perplexity versus execution success rate, showing high-quality samples (80%+ success) concentrated below 1.05.

instruction-tuned models of similar scale, UCoder exhibits competitive or superior performance, consistently matching or exceeding Qwen2.5-Coder-Instruct on challenging benchmarks including MBPP+, BigCodeBench-Complete, and FullStack-Bench at all model scales (7B, 14B, 32B). Although certain Instruct models retain advantages on HumanEval, our approach progressively narrows this gap as model scale increases. These results indicate that execution-driven self-training can effectively elicit the latent instruction-following capabilities embedded in pre-trained code models, achieving supervised-level performance without requiring any human annotations or curated instruction data.

3.3 Effects of Iterative Self-Improvement

Table 2 reports performance across six self-bootstrapping iterations at three model scales, demonstrating both framework effectiveness and scale-dependent dynamics.

Framework Effectiveness. Iterative self-bootstrapping consistently improves performance without external supervision. Across benchmarks, all model scales show substantial gains over seed-trained baselines (Iter 0), with improvements of +6.1 to +13.2 points at 7B, +4.3 to +10.6 at 14B, and +3.0 to +4.9 at 32B. Gains are most pronounced on benchmarks requiring diverse programming skills, such as MBPP, FullStack-Bench, and LiveCodeBench, rather than narrowly scoped tasks like HumanEval. This supports our hypothesis that self-generated problem diversity combined with execution-driven consensus expands capability coverage beyond seed data.

Inverse Scaling of Improvement. Performance gains exhibit an inverse scaling trend, with smaller models benefiting disproportionately. We attribute this effect to *latent capability gaps*, where pre-trained knowledge is only partially accessible through standard prompting. Consensus-based selection mitigates this gap by reinforcing correct patterns that smaller models generate inconsistently. Notably, the self-improved 7B model reaches 85.2% on MBPP, approaching the 32B baseline (86.2%), highlighting self-bootstrapping as a compute-efficient alternative to model scaling.

Convergence Characteristics. The optimal number of iterations decreases with model scale—six for 7B, five for 14B, and four for 32B. Beyond these points, performance exhibits mild oscillation rather than degradation, reflecting a trade-off between specialization on synthetic data and generalization to held-out distributions. These observations motivate early stopping based on validation performance.

4 Analysis

4.1 Diversity of Self-Generated Problems

Lexical Diversity. We quantify lexical diversity using Shannon entropy $H = -\sum_i p_i \log_2 p_i$, where p_i denotes token probability. Figure 4 shows a near-Gaussian entropy distribution (mean $\mu = 3.64$ bits, $\sigma = 0.69$, median 3.62), indicating natural variation rather than templated construction. The smooth CDF and moderate interquartile range

Method	HumanEval		MBPP		BCB-Complete		BCB-Instruct		LiveCode-Bench	FullStack-Bench
	HE	HE+	MBPP	MBPP+	Full	Hard	Full	Hard		
7B Models										
UCoder	**77.4**	67.1	**72.0**	**63.0**	44.4	15.5	34.6	**14.9**	**13.0**	**40.25**
Random	73.8	64.6	65.6	56.6	45.5	14.9	**37.5**	13.5	10.7	33.55
Cluster	73.8	66.5	68.8	58.5	41.3	12.2	36.5	12.2	10.7	38.35
Low PPL	**77.4**	**70.7**	70.1	59.3	**45.6**	**17.6**	37.3	12.2	12.2	37.11
Success Rate	75.0	67.7	66.9	57.1	41.4	11.5	33.0	12.2	12.2	40.01
14B Models										
UCoder	**83.5**	76.8	**75.9**	**64.0**	53.3	**23.6**	43.2	16.2	**22.1**	**50.09**
Random	81.7	76.2	71.7	60.1	**53.8**	20.9	**43.3**	17.6	16.8	47.95
Cluster	82.3	76.8	73.0	**64.0**	50.4	18.2	42.4	15.5	17.6	48.55
Low PPL	82.9	**77.4**	73.3	63.5	50.7	20.3	42.2	**23.0**	19.1	49.32
Success Rate	82.3	75.6	74.1	63.2	50.6	20.9	41.3	14.9	17.6	47.36
32B Models										
UCoder	**86.0**	78.0	**86.2**	**72.2**	**54.8**	**28.4**	44.6	18.9	**22.1**	**53.05**
Random	84.8	**79.3**	80.7	69.6	53.7	25.0	**46.6**	21.6	15.3	39.18
Cluster	83.5	75.6	81.5	71.2	53.9	23.0	45.1	16.9	13.0	49.91
Low PPL	84.1	76.2	79.9	67.5	53.5	27.7	45.4	**23.0**	18.3	50.09
Success Rate	82.3	75.6	81.2	69.6	54.1	27.7	44.2	17.6	21.4	50.50

Table 3: Ablation study comparing data selection strategies across model scales: **UCoder** (execution-driven consensus), **Random** (random sampling from successful solutions), Cluster (clustering-based), **Low PPL** (lowest perplexity), and **Success Rate** (weighted by execution success). All metrics show Pass@1 execution rates (%). **Bold/Underline** indicate best/second-best performance per size category.

further suggest balanced coverage from concise to elaborate specifications.

Semantic Coverage. As shown in Figure 8 (Appendix), the generated problems contain 229 domain-specific terms across seven categories, with Data Structures (18.3%), Algorithms (14.8%), and String Processing (11.4%) being most prominent. No category exceeds 20%, demonstrating broad semantic coverage, while concrete algorithmic terms (e.g., *dijkstra*, *greedy*, *traversal*) indicate non-generic, verifiable challenges.

Complexity Distribution. We assess problem difficulty using a *Complexity Score* (aggregating parameter count, description length, and algorithmic keywords) and conceptual breadth using a *Semantic Coverage Score* (weighted keyword matches across seven categories). As shown in Figure 5, the two metrics exhibit moderate correlation ($r = 0.664$) with continuous distributions (complexity: 3.55 ± 1.60, semantic: 3.16 ± 1.32), suggesting a natural difficulty continuum for curriculum learning. Detailed definitions are in Appendix B.1.

4.2 Execution-Driven Consensus Effectiveness

Complexity Distribution. We assess problem difficulty and conceptual breadth using a *Complexity Score* (0–10, aggregating parameter count, description length, algorithmic keywords, and constraints) and a *Semantic Coverage Score* (weighted keyword matches across seven categories). Detailed score definitions are provided in Appendix B.1. Figure 5 shows a moderate positive correlation ($r = 0.664$), indicating that more complex problems tend to integrate multiple concepts. Problems span the full score ranges (complexity: 3.55 ± 1.60, semantic: 3.16 ± 1.32) with a continuous density profile, suggesting a natural difficulty continuum suitable for curriculum learning.

Quality Distribution Characterization. Given this diversity, we examine whether solution quality exhibits sufficient separation for reliable selection. Figure 6 shows the perplexity distribution over 9,700 sampled candidates and its relationship with execution success. The distribution is right-skewed (mean 1.03, std 0.03) with clear stratification: high-quality samples concentrate at low perplexity values around 1.01, while lower-quality samples progressively shift toward higher perplexity, extending beyond 1.10. Consistently, solutions with execution success rates above 80% cluster predominantly below perplexity 1.05, with rapid performance degradation beyond this range. This sharp transition indicates that high-quality solutions form a distinct and identifiable subset within the candidate pool.

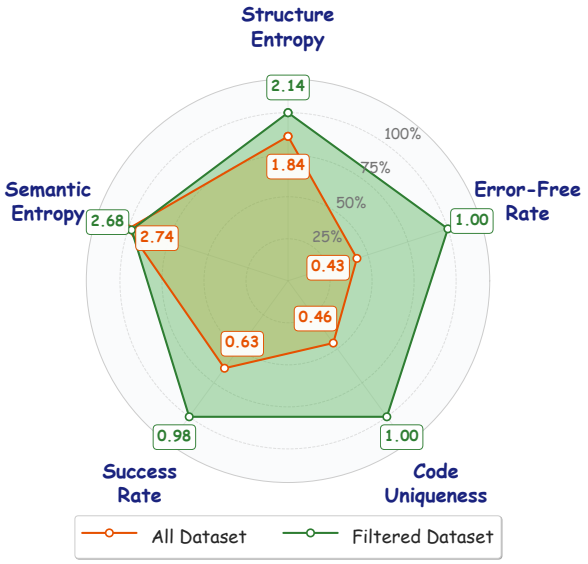

Figure 7: Filtered dataset (green) improves quality over full dataset (gray) while maintaining diversity.

Quality Improvement. Consensus-based filtering yields substantial improvements across all quality metrics. In Figure 7, the filtered dataset consistently outperforms the full dataset in success rate, error-free rate, and code uniqueness. These gains are achieved without sacrificing diversity, as structural and semantic entropy remain comparable between datasets. This demonstrates that execution-driven consensus selection effectively resolves the quality–diversity trade-off. Detailed metric definitions are provided in Appendix B.2.

4.3 Ablation Study

To isolate the effect of data selection, we compare our consensus-based approach with four alternatives: **Random** uniform sampling, **Cluster** selection from dominant output-hash clusters, **Low PPL** selection based on minimal perplexity, and **Success Rate** filtering with a 0.5 execution threshold. Table 3 shows our approach consistently outperforms all baselines across three model scales, achieving the best results on most benchmarks. The performance gap widens with scale, on FullStackBench, the margin over Random increases from 6.7 points at 7B to 13.9 points at 32B, suggesting larger models produce more diverse candidate pools where principled selection matters most. While individual baselines excel on specific benchmarks, none matches the robustness of consensus-based selection. Notably, Success Rate filtering fails to consistently outperform Random, indicating binary pass/fail criteria inadequately capture quality differences, whereas consensus-based selection leverages behavioral consistency across test inputs as a richer quality signal.

5 Related Work

Unsupervised Learning for Code Post-training. Unsupervised learning has become increasingly prominent in code generation through pre-training on vast unlabeled code repositories, building on early findings that source code exhibits statistical regularities similar to natural language (Rozière et al., 2023; Guo et al., 2024; Li et al., 2023; Lozhkov et al., 2024; Zhang et al., 2025; Yang et al., 2025). Recent work on unsupervised code post-training has focused on leveraging unlabeled code snippets and use LLM to generate synthetic question-answering data. Magicoder (Wei et al., 2023) uses open-source code examples to teach LLMs how to create varied coding instructions and training data. Besides, WizardCoder (Luo et al., 2023) progressively evolves simple coding instructions into complex ones for training. Further, WaveCoder (Yu et al., 2023) and CodeArena (Yang et al., 2024) generate more diverse and high-quality instruction data from the open source code dataset. Besides, there are some works (Pravilov et al., 2021; Ahmad et al., 2023; Zhang et al., 2026; Li et al., 2026; Liu et al., 2025; Yang et al., 2026b,c) that adopt supervised learning for code translation and code change tasks.

Code Instruction Tuning. Instruction tuning (Huang et al., 2025; Hui et al., 2024; Yang et al., 2025, 2026a; Li et al., 2024; Lai et al., 2026) has emerged as an effective approach for improving LLMs by fine-tuning on instruction-based datasets, enabling better generalization and instruction-following capabilities. To enhance code LLMs, researchers researchers have enhanced multiple code tasks and benchmarks, such as code generation of multiple domains (e.g., BigCodeBench (Zhuo et al., 2024), FullStackBench (Liu et al., 2024)), multilingual code generation (e.g., MultiPl-E (Cassano et al., 2023), McEval (Chai et al., 2024)), and competitive programming (e.g., LiveCodeBench (Jain et al., 2024)).

6 Conclusion

In this work, we introduce an execution-driven self-bootstrapping framework that removes the need for human-annotated instruction data in code generation. UCoder exploits latent programming knowledge in pre-trained models and uses execu-

tion feedback as a scalable, deterministic supervision signal for autonomous improvement. Guided by execution-driven consensus clustering, iterative self-training identifies correct implementations via behavioral consistency and constructs high-quality training data. Our results demonstrate that code models can achieve performance competitive with supervised baselines through fully autonomous learning, highlighting a scalable and cost-effective path for advancing code intelligence.

Limitations

Despite the promising results, our work has several limitations that warrant future investigation. First, the effectiveness of our consensus-based selection mechanism relies on the availability of executable test cases, which may not always be feasible for certain programming tasks or domains where formal specifications are difficult to construct. Second, while our method demonstrates strong performance on standard benchmarks, the computational cost of generating and evaluating 128 candidate solutions per problem remains substantial, potentially limiting its applicability in resource-constrained scenarios. Third, our approach primarily focuses on functional correctness through execution-based validation, which may not capture other important code quality attributes such as maintainability, documentation quality, or adherence to specific coding standards beyond those explicitly testable. Fourth, the iterative self-training process exhibits diminishing returns and potential overfitting to synthetic data distributions after a certain number of iterations, requiring careful validation-based early stopping. Finally, our analysis is primarily conducted on Python programming tasks, and the generalizability of our findings to other programming languages with different execution characteristics and paradigms remains to be thoroughly validated. These limitations highlight important directions for future work in unsupervised code generation.

Ethics Statement

This work on unsupervised code generation acknowledges several ethical considerations. Automated code generation systems can be misused to produce malicious code or security vulnerabilities, requiring appropriate safeguards, including content filtering and usage monitoring. All experiments used publicly available benchmarks and open-source models, ensuring transparency without collecting proprietary data. We advocate for responsible development with clear documentation, transparent methodologies, and adherence to intellectual property rights.

Acknowledgment

This work is supported by the Fundamental Research Funds for the Central Universitie (Grant No. GW2025-19) and supported by State Key Laboratory of Complex & Critical Software Environment (Grant No. SKLCCSE-2025ZX-26).

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and generate to back-translate: Unsupervised translation of programming languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1528–1542.

Anthropic. 2025. Introducing claude sonnet 4.5.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. 2024. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie

Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. abs/2107.03374.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. 2025. Sft memorizes, rl generalizes: A comparative study of foundation model post-training. *arXiv preprint arXiv:2501.17161*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Xianzhen Luo, Qiufeng Wang, Yuan-Tao Fan, Qingfu Zhu, Zhaoxiang Zhang, Yang Gao, Jie Fu, Qian Liu, Houyi Li, Ge Zhang, Yuan Qi, Yinghui Xu, Wei Chu, and Zili Wang. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 33167–33193. Association for Computational Linguistics.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Peng Lai, Zhihao Ou, Yong Wang, Longyue Wang, Jian Yang, Yun Chen, and Guanhua Chen. 2026. Biasscope: Towards automated detection of bias in LLM-as-a-judge evaluation. In *The Fourteenth International Conference on Learning Representations*.

Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, and Yongshun Gong. 2026. What papers don't tell you: Recovering tacit knowledge for automated paper reproduction.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Yixia Li, Boya Xiong, Guanhua Chen, and Yun Chen. 2024. Setar: Out-of-distribution detection with selective low-rank approximation. *Advances in Neural Information Processing Systems*, 37:72840–72871.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. 2025. Context as a tool: Context management for long-horizon swe-agents.

Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, ZY Peng, et al. 2024. Fullstack bench: Evaluating llms as full stack coder. *arXiv preprint arXiv:2412.00535*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

OpenAI. 2025. Introducing upgrades to codex: Gpt-5-codex. `https://openai.com/index/introducing-upgrades-to-codex/`.

Mikhail Pravilov, Egor Bogomolov, Yaroslav Golubev, and Timofey Bryksin. 2021. Unsupervised learning of general-purpose embeddings for code changes. In *Proceedings of the 5th international workshop on machine learning techniques for software quality evolution*, pages 7–12.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet,

Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *CoRR*, abs/2312.02120.

Jian Yang, Xianglong Liu, Weifeng Lv, Ken Deng, Shawn Guo, Lin Jing, Yizhi Li, Shark Liu, Xianzhen Luo, Yuyu Luo, Changzai Pan, Ensheng Shi, Yingshui Tan, Renshuai Tao, Jiajun Wu, Xianjie Wu, Zhenhe Wu, Daoguang Zan, Chenchen Zhang, Wei Zhang, He Zhu, Terry Yue Zhuo, Kerui Cao, Xianfu Cheng, Jun Dong, Shengjie Fang, Zhiwei Fei, Xiangyuan Guan, Qipeng Guo, Zhiguang Han, Joseph James, Tianqi Luo, Renyuan Li, Yuhang Li, Yiming Liang, Congnan Liu, Jiaheng Liu, Qian Liu, Ruitong Liu, Tyler Loakman, Xiangxin Meng, Chuang Peng, Tianhao Peng, Jiajun Shi, Mingjie Tang, Boyang Wang, Haowen Wang, Yunli Wang, Fanglin Xu, Zihan Xu, Fei Yuan, Ge Zhang, Jiayi Zhang, Xinhao Zhang, Wangchunshu Zhou, Hualei Zhu, King Zhu, Bryan Dai, Aishan Liu, Zhoujun Li, Chenghua Lin, Tianyu Liu, Chao Peng, Kai Shen, Libo Qin, Shuangyong Song, Zizheng Zhan, Jiajun Zhang, Jie Zhang, Zhaoxiang Zhang, and Bo Zheng. 2025. From code foundation models to agents and applications: A comprehensive survey and practical guide to code intelligence.

Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024. Evaluating and aligning codellms on human preference. *arXiv preprint arXiv:2412.05210*.

Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu, Cening Liu, X Ma, et al. 2026a. Iquest-coder-v1 technical report. *arXiv preprint arXiv:2603.16733*.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, Weicheng Gu, Yaxin Du, Joseph Li, Fanglin Xu, et al. 2026b. Incoder-32b: Code foundation model for industrial scenarios. *arXiv preprint arXiv:2603.16790*.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Tuney Zheng, Fanglin Xu, Weicheng Gu, Lin Jing, Yaxin Du, Joseph Li, et al. 2026c. Incoder-32b-thinking: Industrial code world model for thinking. *arXiv preprint arXiv:2604.03144*.

Junjie Ye, Yuming Yang, Yang Nan, Shuo Li, Qi Zhang, Tao Gui, Xuan-Jing Huang, Peng Wang, Zhongchao Shi, and Jianping Fan. 2025. Analyzing the effects of supervised fine-tuning on model knowledge from token and parameter levels. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 471–513.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?, 2025. *URL https://arxiv. org/abs/2504.13837*.

Wei Zhang, Jack Yang, Renshuai Tao, Lingzheng Chai, Shawn Guo, Jiajun Wu, Xiaoming Chen, Ganqu Cui, Ning Ding, Xander Xu, Hu Wei, and Bowen Zhou. 2025. V-gamegym: Visual game generation for code large language models.

Yitong Zhang, Chengze Li, Ruize Chen, Guowei Yang, Xiaoran Jia, Yijie Ren, and Jia Li. 2026. To see is not to master: Teaching llms to use private libraries for code generation.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Additional Analysis Figures

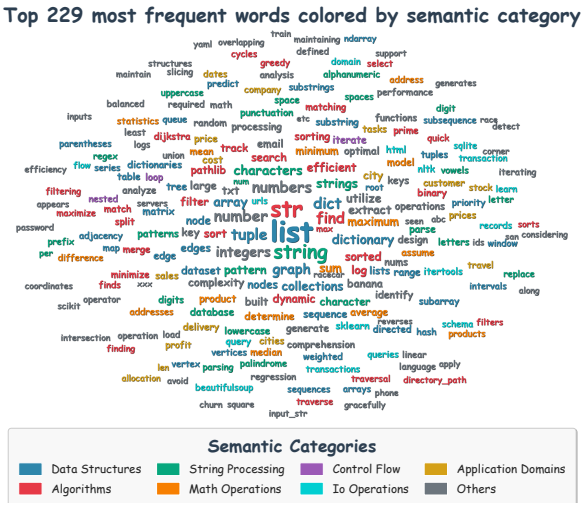

Figure 8: Semantic distribution of 229 frequent terms in generated problems. Word size indicates frequency; colors denote seven semantic categories (see Appendix B.1 for category definitions).

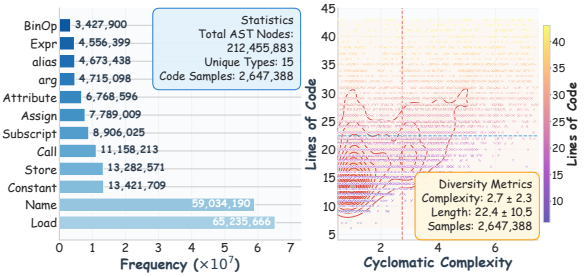

Figure 9: Solution space diversity analysis. Left: AST node type distribution across 2.6M samples totaling 212M nodes, spanning 15 distinct syntactic constructs. Right: Cyclomatic complexity (number of linearly independent paths through the code) versus code length with density contours, showing broad dispersion (complexity: 2.7 ± 2.3, length: 22.4 ± 10.5 lines).

B Metric Definitions

B.1 Problem Diversity Metrics

Semantic Category Coverage. To quantify the breadth of programming concepts in generated problems, we define seven semantic categories:

- Data Structures: Arrays, lists, trees, graphs, heaps, and related terminology (e.g., node, edge).
- Algorithms: Sorting, searching, graph traversal, dynamic programming, and greedy methods.
- String Processing: Substring operations, pattern matching, and tokenization.
- Math Operations: Arithmetic, combinatorics, number theory, and statistics.
- Control Flow: Iteration, branching, recursion, and nested structures.
- I/O Operations: Serialization, encoding, and library/framework usage.
- Application Domains: Business scenarios including e-commerce, finance, and education.

Each extracted term is assigned to the first matching category via exact or substring keyword matching. The distribution is visualized in Figure 8.

Complexity Score. We compute a composite complexity score $C \in [0, 10]$ as the sum of four sub-scores:

$$C = S_{\text{param}} + S_{\text{length}} + S_{\text{algo}} + S_{\text{const}} \quad (7)$$

- Parameter Score $S_{\text{param}} \in [0, 3]$: Number of function parameters multiplied by 0.5, capped at 3.
- Length Score $S_{\text{length}} \in [0, 2]$: $\min(\text{description length}/500, 2.0)$.
- Algorithm Score $S_{\text{algo}} \in [0, 4]$: Weighted sum of matched algorithmic keywords with three-tier weights. High-weight keywords (0.8 each) include dynamic, recursive, backtrack, graph, tree, optimize, shortest, longest, maximum, minimum, subsequence, subarray, permutation, combination, dfs, bfs, dijkstra, binary search. Medium-weight keywords (0.4 each) include sort, search, hash, stack, queue, linked, matrix, two pointer, sliding window, greedy, merge, divide. Low-weight keywords (0.2 each) include sum, count, average, reverse, palindrome, anagram, frequency, duplicate, unique, filter, map.
- Constraint Score $S_{\text{const}} \in [0, 1]$: Each matched constraint keyword (constraint, edge case, boundary, overflow, empty, null, negative, large, efficiency) adds 0.2, capped at 1.

Semantic Coverage Score. We measure conceptual breadth by counting keyword matches across seven semantic categories with differentiated weights:

$$S_{\text{semantic}} = \sum_{c \in \mathcal{C}} w_c \cdot |\mathcal{K}_c \cap \text{tokens}(p)| \quad (8)$$

where $\mathcal{K}_c$ is the keyword set for category c and w_c is the category weight. High-weight categories ($w = 0.3$) are Algorithms and Data Structures. Medium-weight categories ($w = 0.2$) are String Processing and Math Operations. Low-weight categories ($w = 0.1$) are Control Flow, I/O Operations, and Application Domains.

Example keywords for each category:

- Algorithms: sort, search, traverse, recursive, dynamic programming, greedy, backtracking, divide and conquer, binary search, hashing, sliding window, two pointers
- Data Structures: array, list, tree, graph, stack, queue, heap, hash table, linked list, set, dictionary, matrix, trie
- String Processing: parse, match, regex, substring, concatenate, split, replace, format, encode, decode, pattern
- Math Operations: arithmetic, modulo, prime, factorial, fibonacci, gcd, lcm, power, logarithm, probability, statistics, geometry
- Control Flow: loop, iteration, condition, branch, recursion, break, continue, exception, error handling
- I/O Operations: read, write, input, output, file, stream, print, format, serialize, deserialize
- Application Domains: database, network, web, api, game, simulation, encryption, compression, scheduling, optimization

B.2 Solution Quality and Diversity Metrics

Structure Entropy. We measure code structural diversity through the average of four entropy components:

$$H_{\text{structure}} = \frac{1}{4}(H_{\text{ast}} + H_{\text{control}} + H_{\text{cc}} + H_{\text{nest}}) \quad (9)$$

where each component is computed as the Shannon entropy over the corresponding distribution:

- H_{ast}: Entropy over AST node type frequencies (e.g., FunctionDef, If, For, Call, BinOp).
- H_{control}: Entropy over control structure usage (If, For, While statements).
- H_{cc}: Entropy over cyclomatic complexity values, discretized into 10 bins.
- H_{nest}: Entropy over maximum nesting depth values.

Semantic Entropy. We measure implementation approach diversity through the average of two entropy components:

$$H_{\text{semantic}} = \frac{1}{2}(H_{\text{builtin}} + H_{\text{pattern}}) \quad (10)$$

where:

- H_{builtin}: Entropy over built-in function usage frequencies (len, sum, max, min, sorted, map, filter, range, zip, enumerate, etc.).
- H_{pattern}: Entropy over five algorithm pattern categories: recursion (self-referential function calls), list comprehension, generator expressions, iteration (For/While loops), and functional style (map/filter usage).

Code Uniqueness. We compute code uniqueness as the ratio of distinct implementations to total samples:

$$U = \frac{|\{h(c) : c \in \mathcal{C}\}|}{|\mathcal{C}|} \quad (11)$$

where $h(c)$ is a hash function over the code string c and $\mathcal{C}$ is the set of all code samples.

Error-Free Rate. We define the error-free rate as the proportion of samples that execute without runtime errors:

$$R_{\text{error-free}} = 1 - \frac{|\{c \in \mathcal{C} : \text{error}(c) = \text{True}\}|}{|\mathcal{C}|} \quad (12)$$

Preservation Rate. For each metric M, we compute the preservation rate after filtering as:

$$P_M = \frac{M_{\text{filtered}}}{M_{\text{all}}} \times 100\% \quad (13)$$

Values above 90% indicate successful diversity preservation during quality-based filtering.