InCoder-32B-Thinking: Industrial Code World Model for Thinking

Anonymous ACL submission

Abstract

Industrial code tasks, including chip design, GPU optimization, and firmware, often fail for reasons that are invisible in ordinary source code: timing constraints, backend logs, layout data, simulator behavior, and hardware-specific performance trade-offs. Current code models see many programs but few expert correction traces from these settings. We present InCoder-32B-Thinking, a 32B thinking code model trained with Error-driven Chain-of-Thought (ECoT) synthesis and an Industrial Code World Model (ICWM). ECoT converts several rounds of toolchain signals into reasoning traces by preserving the failed code, the observed failure, and the later fix. ICWM learns from execution traces in domains such as Verilog simulation and GPU profiling, then predicts feedback for the data engine, which can build more repair traces without routing every candidate through a real toolchain. The data keeps the uneven reasoning depth of industrial work: short traces for simple cases, long traces for hardware-level debugging. Across 23 tests (14 general and 9 industrial), InCoder-32B-Thinking reaches 81.3% on LiveCodeBench v5, 84.0% CAD-Coder compile success, and 38.0% on KernelBench L2, while remaining competitive on general coding tasks.

1 Introduction

Large language models (LLMs) now solve many routine coding tasks, and recent systems such as DeepSeek-V3.2 (Liu et al., 2025a) and Claude-Sonnet-4.6 (Anthropic, 2026) perform well on software tests (Jimenez et al., 2024; SWE-bench Live Team, 2025; Khandpur et al., 2025; Barres et al., 2025; Luo et al., 2025). The harder cases are less like short Python functions and more like systems work: RTL design, GPU kernels, firmware, compiler work, and CAD scripts. Recent code LLMs (Yang et al., 2026) and industrial tests (Liu et al., 2026; Jin et al., 2025) make this gap visible. A syntactically valid program can still violate a timing rule, exceed a shared-memory limit, generate invalid geometry, or fail once tool feedback arrives. Systems such as OpenAI's o-series (OpenAI, 2025) and DeepSeek-R1 (Guo et al., 2025) attack difficult tasks with more tokens for intermediate reasoning. That helps, but domain code exposes a different issue: the key reasoning often comes from tool data. Engineers do not debug a Verilog module or a Triton kernel from text alone; they read compiler errors, waveform hints, profiler logs, and simulator traces.

This paper combines two ideas that are usually treated separately. Thinking models yield longer deliberation traces. Code world models (Copet et al., 2025) predict toolchain feedback. Both are needed here. CUDA tuning (Ouyang et al., 2025), Verilog generation (Liu et al., 2023c), firmware programming (Englhardt et al., 2024), and compiler optimization (Cummins et al., 2023) all depend on semantics that are poorly represented in web-scale text. Their feedback is also costly: each trial may require compilers, simulators, profile tools, or geometry checks. Current models still struggle on Triton generation (Li et al., 2025c) and Verilog checking (Jin et al., 2025), even when their general code scores are high.

A concrete CUDA failure case and the full overview appear in Appendix B, with notes on what each figure is intended to show.

We introduce InCoder-32B-Thinking, a 32B thinking code model trained from tool-grounded correction traces. The training pipeline has two parts. Error-driven Chain-of-Thought (ECoT) keeps the failed code, the tool error, and the fixed

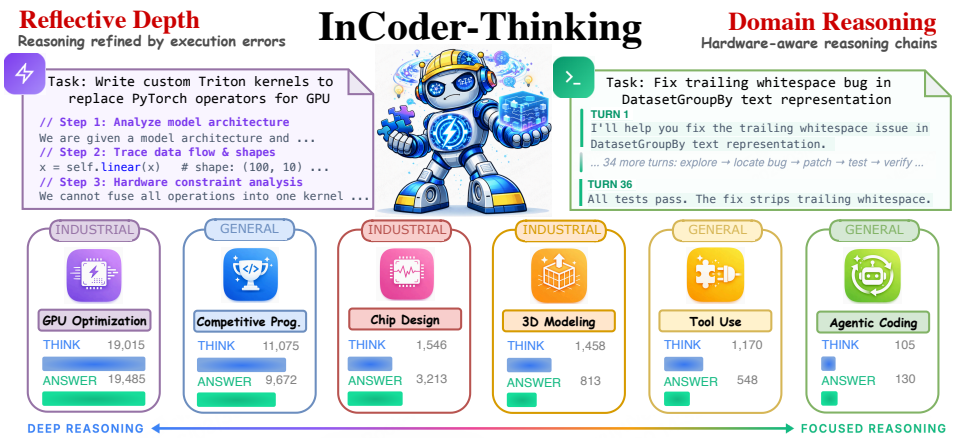

Figure 1: Schematic overview of InCoder-32B-Thinking across general and industrial code tasks.

code in a single trace, then turns that trace into a reasoning example. The Industrial Code World Model (ICWM) uses real execution traces to learn backend signals for new programs. It acts as a learned simulator for Verilog logs, GPU profiles, CAD checks, firmware traces, and build logs. This lets us build many more repair traces while still anchoring them to real toolchain behavior through audits.

We evaluate InCoder-32B-Thinking on 14 general-code and 9 industrial-code tests. It reaches 70.4% on SWE-bench Verified (OpenAI, 2024), 81.3% on LiveCodeBench_v5 (Jain et al., 2024), and 63.9% on BFCL (Patil et al., 2025). Against its instruct counterpart, it keeps similar SWE-bench Verified performance and improves LiveCodeBench by 28.0 percentage points. For those nine domain tests, it is strongest among open LLMs on several chip-design, GPU, CAD, and code-optimization metrics, including CAD-Coder compilation and KernelBench L2. We contribute:

- **Error-driven thinking synthesis.** ECoT turns multi-turn execution feedback into traces that show how an incorrect program is diagnosed and repaired, rather than only showing the final answer.
- **Industrial Code World Model.** It models backend observations from the task state and candidate code. It scales trajectory synthesis for domains where real execution is slow or brittle.
- **A thinking model for industrial code.** InCoder-32B-Thinking uses these traces to improve reasoning on both general code and hardware-aware programming tasks, with the largest gains on tasks that expose rich tool feedback.

2 Error-driven thinking synthesis with an Industrial Code World Model

The data engine starts from real toolchain runs and then uses a learned proxy to scale them. First, we collect task queries, model responses, code edits, and backend feedback from real execution. We train the Industrial Code World Model (ICWM) on those execution turns. After that, ICWM scores feedback for new candidates, so the generator can build more multi-turn correction traces without a real backend call at every step. The pipeline has two phases (Figure 2):

1. **Grounded collection (subsection 2.1 and subsection 2.2).** A generator writes code, runs it through the relevant toolchain or simulator, records structured feedback, and revises the answer for up to K rounds. These trajectories form $\mathcal{D}_{\text{real}}$.
2. **ICWM amplification (subsection 2.3).** The same trajectories train ICWM. During data synthesis, ICWM supplies predicted feedback in place of the real backend. Periodic audits with real runs keep the proxy from drifting.

2.1 Task Seeding and Environment Bundling

We reuse the task seeds from InCoder-32B (Yang et al., 2026), including prompts, unit tests, and execution environments. Each seed is paired with the artifacts needed to run it. Verilog tasks include testbenches and Yosys scripts. STM32 firmware tasks include memory layouts, CMSIS headers, and linker scripts. We denote the packaged input as

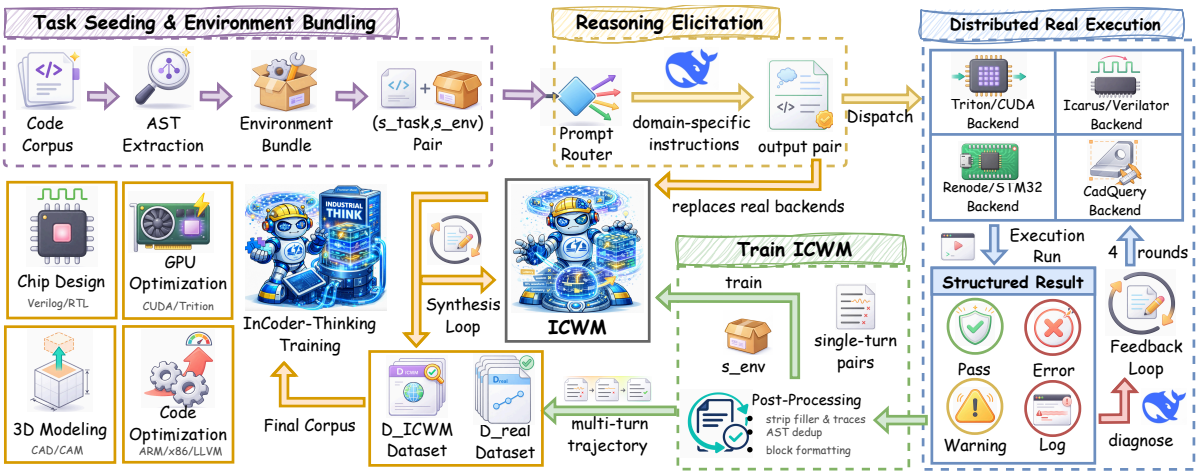

Figure 2: Data engine. Left: task seeds and environment bundles pass through real backends to produce multi-turn trajectories $\mathcal{D}_{\text{real}}$. Right: $\mathcal{D}_{\text{real}}$ trains ICWM, which estimates backend signals for larger-scale synthesis. Real audits recalibrate the proxy.

$\langle s_{\text{task}},\ s_{\text{env}}\rangle$, where s_{env} is the executable context used by both the real backend and ICWM.

2.2 Execution Grounded Trajectory Synthesis

Given a task seed and its environment bundle, a frontier generator produces an initial reasoning trace and candidate code. A lightweight prompt router first reads the environment bundle and chooses domain instructions. GPU tasks list warp divergence and shared-memory limits; RTL tasks list combinational path depth and clock crossings; CAD tasks list wall width and manifold validity; firmware tasks list peripheral order. With these instructions, DeepSeek-V3.2 produces $(r^{(0)}, c^{(0)})$, the first trace and program.

Distributed execution. The candidate code $c^{(0)}$ is sent to the matching backend: Triton/CUDA for GPU kernels, Renode for microcontroller firmware, CadQuery for CAD geometry, and Yosys/Icarus for RTL. Each backend returns an outcome label, such as PASS, COMPILATION_ERROR, or MEMORY_FAULT, together with the diagnostic log and any task-specific output.

Synthesizing reasoning content from multi-turn interaction. When execution fails, the feedback $o^{(k)}$ is fed back to the generator. This feedback can be a compiler message, simulator failure, hardware exception, geometry diff, or profiler output. The generator must explain the failure and produce a revised trace and implementation. We allow up to $K = 4$ correction rounds, giving a trajectory:

$$\tau = \left[\langle s_{\text{init}}, r^{(0)}, c^{(0)}\rangle \rightarrow \langle r^{(1)}, c^{(1)}\rangle \cdots \rightarrow \langle r^{(k)}, c^{(k)}\rangle\right] \tag{1}$$

Here s_{init} is the initial task description plus the task bundle, $r^{(k)}$ is the reasoning at turn k, and $c^{(k)}$ is the matching program. Each arrow marks one round of backend feedback followed by an edit. A trajectory stops when the code passes all checks or reaches the turn limit.

We keep both failed and successful intermediate turns. The failed turns provide the error modes; the later turns provide the repairs. Full trajectories train the code model; execution turns train ICWM.

2.3 ICWM Driven Data Amplification

Real execution is the cleanest supervision signal, but it is expensive and operationally fragile. These tasks require domain toolchains, simulator state, hardware-specific build settings, and timeout control. ICWM is a learned proxy for this signal: it predicts backend observations from the task bundle and candidate code.

Preliminaries. For an industrial coding task, the "world" is the executable environment that decides whether a revision works. The bundle s_{env} includes testbenches, simulators, compiler settings, configs, memory layouts, and domain rules. At turn k, the generator proposes $c^{(k)}$. Running it under s_{env} gives $o^{(k)}$, the observed feedback.

Definition of ICWM. ICWM is a language model over environment-code pairs. Given s_{env} and candidate program c, it predicts feedback from the real backend: the status label, diagnostic text, and any numerical or structured output.

$$\text{ICWM}_\theta : \ (s_{\text{env}},\ c^{(k)}) \longmapsto \hat{o}^{(k)} \tag{2}$$

Table 1: Results on code reasoning, code efficiency, and text-to-SQL benchmarks.

Model	Size	Code Reasoning				Code Efficiency		Text2SQL	
		CruxEval		LiveCodeBench		Mercury		Bird	Spider
		Input-COT	Output-COT	V5	V6	Beyond@1	Pass@1		
6B+ Models									
DeepSeek-Coder-V2-Lite-Instruct	2.4/16B	57.1	56.2	13.2	19.4	76.8	91.4	41.6	72.4
Qwen2.5-Coder-7B-Instruct	7B	66.9	66.0	14.4	18.9	69.9	84.8	53.1	79.8
Seed-Coder-8B-Instruct	8B	62.0	66.6	19.2	22.3	**78.5**	**93.8**	44.7	72.7
Qwen2.5-Coder-14B-Instruct	14B	**75.6**	**79.2**	**22.8**	**24.6**	76.7	88.3	**59.1**	**81.3**
30B+ Models									
Qwen3-Coder-30B-A3B-Instruct	3.3/30.5B	76.9	80.5	43.1	36.0	81.1	95.3	59.0	80.9
DeepSeek-V3.2	37/671B	82.1	94.2	-	83.3	**81.6**	**96.9**	52.6	77.9
Qwen2.5-Coder-32B-Instruct	32B	78.8	84.0	30.5	27.4	79.1	96.1	62.1	**83.9**
Qwen3-235B-A22B-Instruct-2507	22/235B	62.0	89.5	53.9	51.8	80.4	**96.9**	**62.8**	81.1
Qwen3-235B-A22B-Thinking-2507	22/235B	15.2	46.9	80.2	74.1	61.2	70.3	35.2	42.6
Qwen3-Coder-480B-A35B-Instruct	35/480B	87.1	90.4	48.6	53.9	80.2	96.1	61.3	81.2
Kimi-Dev-72B	72B	33.0	64.2	46.1	40.0	59.1	69.5	-	-
Kimi-K2-Instruct-0905	32B/1T	86.8	89.5	52.1	53.7	76.1	90.6	60.4	81.1
Kimi-K2-Thinking	32B/1T	**92.2**	86.2	-	83.1	73.0	85.2	40.6	49.6
KAT-Dev	32B	42.5	65.1	32.9	32.6	75.1	89.1	52.2	77.6
KAT-Dev-72B-Exp	72B	71.4	81.1	13.8	16.0	79.0	94.5	35.2	60.3
GLM-4.7	32/355B	65.6	81.2	-	**84.9**	74.1	86.7	46.5	62.4
InCoder-32B	32B	62.4	73.9	53.3	49.1	71.4	85.6	55.4	79.7
InCoder-32B-Thinking	32B	88.9	**95.5**	**81.3**	77.1	62.4	73.4	47.9	66.7

where $\hat{o}^{(k)}$ includes a status label (PASS, COMPILATION_ERROR, etc.), a diagnostic message, and, when applicable, numeric outputs or diffs. We train the model on every real execution turn in $\mathcal{D}_{\text{real}}$. Tags and templates let one model cover the evaluated industrial domains while keeping separate log types apart.

ICWM-driven synthesis loop. After training, ICWM replaces real backends in the feedback loop described in subsection 2.2:

$$\tau = \left[\langle s_{\text{init}}, r^{(0)}, c^{(0)} \rangle \xrightarrow{\text{ICWM}_\theta} \langle r^{(1)}, c^{(1)} \rangle \cdots \xrightarrow{\text{ICWM}_\theta} \langle r^{(k)}, c^{(k)} \rangle \right] \tag{3}$$

Each arrow is now a predicted observation rather than a real compilation or simulation. The model still revises the code from feedback, but a single ICWM forward pass provides that feedback.

Resulting reasoning corpus. The final training set for reasoning is $\mathcal{D} = \mathcal{D}_{\text{real}} \cup \mathcal{D}_{\text{icwm}}$. Every trace in $\mathcal{D}$ is either produced by real execution or checked against it during audit rounds. This guardrail keeps synthetic reasoning tied to backend behavior rather than free-form self-critique.

3 Evaluation

3.1 Baselines

We compare InCoder-32B-Thinking with open-weight code models and selected proprietary APIs. The two benchmark groups stress different skills, so the baseline set differs between them.

For the general-code group, Table 5, Table 1, and Table 2 report comparisons with DeepSeek-Coder-V2-Lite-Instruct (DeepSeek AI, 2024) and DeepSeek-V3.2 (Liu et al., 2025a); Qwen2.5-Coder-7B, 14B, and 32B (Hui et al., 2024); Qwen3-235B-A22B Instruct and Thinking variants (Yang et al., 2025a); Qwen3-Coder-30B-A3B and 480B-A35B (Qwen, 2025); Seed-Coder-8B (Seed et al., 2025); Kimi-Dev-72B (Yang et al., 2025b); Kimi-K2 variants (Team et al., 2025); the KAT-Dev set (Zhan et al., 2025); GLM-4.7 (AI, 2025); and InCoder-32B. The set covers small and large open models, dense and mixture-of-experts architectures, and models with or without explicit thinking traces.

For the industrial-code group, Table 3 and Table 4 include Qwen3.5-9B, 27B, and 397B-A17B (Qwen Team, 2026b); the Qwen3-Coder set (Qwen, 2025; Qwen Team, 2026a); DeepSeek-V3.2 (Liu et al., 2025a); GLM-4.7 (AI, 2025) and GLM-5 (Zeng et al., 2026); Kimi-K2.5 and Kimi-K2 variants (Team et al., 2026, 2025); MiniMax-M2.5 (MiniMax, 2026); Seed-OSS-36B (Team, 2025a); GPT-OSS-20B and GPT-OSS-120B (Agarwal et al., 2025); and Claude-Sonnet-4.6 (Anthropic, 2026). These models are relevant to hardware-aware generation, program optimization, and other toolchain-bound coding tasks.

3.2 Benchmarks

Figure 6 summarizes the benchmark suite. It groups general-code tasks apart from industrial

Table 2: Results on agentic coding and tool-use benchmarks.

Model	Size	Agentic Coding			General Tool Use				
		Terminal-Bench v1.0	Terminal-Bench v2.0	SWE-bench Verified	Mind2Web	BFCL V3	τ^2-bench Airline	τ^2-bench Retail	τ^2-bench Telecom
6B+ Models									
DeepSeek-Coder-V2-Lite-Instruct	2.4/16B	5.0	-	-	26.7	-	3.5	12.0	-
Qwen2.5-Coder-7B-Instruct	7B	6.3	-	-	38.4	54.2	-	-	-
Seed-Coder-8B-Instruct	8B	7.5	**2.5**	-	38.2	-	**4.3**	**32.0**	-
Qwen2.5-Coder-14B-Instruct	14B	**8.8**	-	-	**42.7**	**59.9**	-	-	-
30B+ Models									
Qwen3-Coder-30B-A3B-Instruct	3.3/30.5B	23.8	23.8	51.9	36.1	63.4	42.0	25.4	25.4
DeepSeek-V3.2	37/671B	23.8	**46.4**	73.1	47.2	68.8	63.8	81.1	**96.2**
Qwen2.5-Coder-32B-Instruct	32B	5.0	4.5	-	32.5	62.3	-	-	-
Qwen3-235B-A22B-Instruct-2507	22/235B	15.0	13.5	45.2	49.0	71.2	50.0	74.6	32.5
Qwen3-235B-A22B-Thinking-2507	22/235B	8.8	3.4	44.6	43.2	**71.9**	58.0	71.9	45.6
Qwen3-Coder-480B-A35B-Instruct	35/480B	37.5	23.6	67.0	54.0	68.7	60.0	77.5	65.8
Kimi-Dev-72B	72B	-	2.3	60.4	-	55.5	21.9	32.0	35.1
Kimi-K2-Instruct-0905	32B/1T	44.5	27.8	69.2	53.4	70.3	56.5	70.6	65.8
Kimi-K2-Thinking	32B/1T	**47.1**	33.7	71.3	55.7	-	-	-	-
KAT-Dev	32B	17.5	10.1	62.4	33.7	64.7	32.0	28.0	35.1
KAT-Dev-72B-Exp	72B	21.3	7.9	74.6	-	-	-	-	-
GLM-4.7	32/355B	36.3	41.0	73.8	53.7	64.8	60.0	70.2	75.4
InCoder-32B	32B	35.0	22.5	**74.8**	**55.8**	61.0	**70.0**	85.1	86.8
InCoder-32B-Thinking	32B	38.8	21.6	70.4	49.1	63.9	62.0	**86.0**	95.6

tasks because the failure modes are different: general tasks mostly test program synthesis, repository repair, tool calls, or SQL generation, while industrial tasks require code that passes domain checks. Appendix B gives a more detailed reading of the plot.

General Code Benchmarks. Following InCoder-32B (Yang et al., 2026), we evaluate general coding with benchmarks for generation, reasoning, speed, text-to-SQL, agentic coding, and tool use. The suite includes EvalPlus (Liu et al., 2023a) with HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), BigCodeBench (Zhuo et al., 2024), FullStackBench (Liu et al., 2024b), CRUXEval (Gu et al., 2024), LiveCodeBench (Jain et al., 2024), Mercury (Du et al., 2024), Spider (Yu et al., 2018), BIRD (Li et al., 2023), Terminal-Bench (Team, 2025c), SWE-bench Verified (OpenAI, 2024), Mind2Web (Deng et al., 2023), BFCL V3 (Patil et al., 2025), and τ^2-bench (Barres et al., 2025). Appendix C gives protocols and metrics for all benchmark families.

Industrial Code Benchmarks. The industrial suite also follows InCoder-32B (Yang et al., 2026). It groups tests by domain. For chip design, we use VeriScope, RealBench (Jin et al., 2025), ArchXBench (Purini et al., 2025), and VeriRepair. For GPU kernels, we use KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025c). For code optimization, we use EmbedCGen and SuperCoder (Wei et al., 2026), and for CAD we use CAD-Coder (Guan et al., 2025). These tests require code to pass compilers, simulators, profile tools, or CAD kernels.

3.3 Main Results

Results on General Code Benchmarks. The main general-code results appear in Table 1 and Table 2; Table 5 in the appendix gives the full generation table. The largest gain is on tasks that require reasoning. InCoder-32B-Thinking scores **81.3** on LiveCodeBench V5, ahead of other open-weight models, including much larger ones. It is also strong on generation and agentic tasks, with the best reported score on τ^2-bench Retail. The trade-off appears on tasks that reward short direct answers, such as Mercury and Text2SQL, where thinking-style models can perform worse.

Results on Industrial Code Benchmarks. The industrial tables show where training helps most. On RealBench module tasks, the model gives the best Func@1 among open LLMs and a competitive Func@5. It also beats the Claude reference on CAD-Coder pass rate and SuperCoder score. Other industrial tests are competitive rather than best on every metric. This pattern matters: gains arise where multi-turn tool signals teach specific fixes, while the top tiers still require strategies beyond more reasoning data.

Table 3: Chip-design results. Gray marks our model.

Model	Size	VeriScope	VeriRepair	RealBench						ArchXBench	
		Score	Fix (%)	System		Module				n	t
				Syn@1	Syn@5	Syn@1	Syn@5	Func@1	Func@5		
6B+ Models											
Qwen3.5-9B	9B	32.0	-	-	-	6.3	15.8	4.3	8.5	1.9	44.3
GPT-OSS-20B	3.6/21B	**73.9**	**86.7**	3.8	17.4	**22.9**	**47.9**	9.8	**21.0**	**3.1**	**53.5**
Qwen3.5-27B	27B	55.7	60.0	**6.2**	**20.1**	17.1	33.8	**10.6**	17.8	2.6	48.3
30B+ Models											
Qwen3-Coder-30B-A3B-Instruct	3.3/30.5B	66.2	76.7	-	-	23.0	35.2	5.2	8.2	2.4	37.3
Seed-OSS-36B-Instruct	36B	67.2	66.7	5.0	21.3	14.3	23.0	11.5	20.3	2.5	43.7
GPT-OSS-120B	5.1/117B	82.2	76.7	5.0	21.3	37.8	64.3	17.5	30.8	3.4	**54.8**
MiniMax-M2.5	10/230B	75.1	66.7	**23.8**	48.5	17.2	38.4	6.9	15.9	2.9	46.0
GLM-4.7	32/355B	81.2	63.3	12.5	24.6	25.4	46.9	11.6	21.2	3.2	51.4
GLM-5	40/744B	**83.2**	**90.0**	2.5	11.2	22.2	43.4	12.2	22.6	3.1	53.2
Kimi-K2.5	32B/1T	73.1	83.3	5.0	17.9	43.7	52.2	23.1	25.7	**3.8**	49.7
Kimi-K2-Instruct	32B/1T	82.4	76.7	6.2	26.2	50.1	70.1	22.2	28.3	2.9	44.9
Kimi-K2-Thinking	32B/1T	73.1	80.0	-	-	27.8	59.4	14.1	28.9	1.5	30.1
DeepSeek-V3.2	37/671B	76.1	77.0	18.8	**55.1**	39.3	52.7	17.2	21.4	3.6	53.9
Qwen3.5-397B-A17B	17/397B	62.5	86.7	11.2	38.1	35.2	59.5	16.4	28.3	3.1	53.5
Qwen3-Coder-480B-A35B-Instruct	35/480B	80.8	76.7	-	-	28.9	39.5	14.8	20.6	3.0	43.9
InCoder-32B	32B	80.7	80.0	10.0	23.7	74.8	**83.3**	62.7	**70.5**	3.4	51.0
InCoder-32B-Thinking	32B	75.4	83.3	12.3	24.5	**75.6**	82.4	**63.1**	69.8	3.1	46.7
Closed-APIs Models											
Claude-Sonnet-4.6	🔒	87.7	83.3	41.2	50.0	69.2	77.7	33.5	37.2	4.4	58.2

Table 4: Results on GPU optimization, code optimization, and 3D modeling. Gray marks our model. †For TritonBench, execution accuracy is reported on the callable subset rather than all attempts.

Model	Size	CAD-Coder		EmbedCGen	SuperCoder		TritonBench				KernelBench		
		Comp.	IoU	Main (%)	Acc. (%)	Spd.	G-call (%)	G-exe† (%)	T-call (%)	T-exe† (%)	L1	L2	L3
6B+ Models													
Qwen3.5-9B	9B	2.0	-	10.0	**36.0**	1.0×	2.7	**100.0**	3.6	**100.0**	-	-	-
GPT-OSS-20B	3.6/21B	2.0	**2.0**	**30.6**	16.0	1.0×	2.2	**100.0**	1.2	**100.0**	**5.1**	**10.0**	2.0
Qwen3.5-27B	27B	**4.0**	0.3	9.6	6.0	**1.9**×	**5.4**	**100.0**	**25.9**	97.7	**5.1**	6.0	**4.0**
30B+ Models													
Qwen3-Coder-30B-A3B-Instruct	3.3/30.5B	-	-	15.4	50.0	1.0×	8.7	**100.0**	24.1	67.5	-	-	-
Seed-OSS-36B-Instruct	36B	2.0	2.0	10.2	8.0	1.1×	1.6	**100.0**	1.8	**100.0**	1.0	2.0	2.0
GPT-OSS-120B	5.1/117B	4.0	1.9	17.8	8.0	1.2×	3.8	85.7	12.7	95.2	6.1	15.0	2.0
MiniMax-M2.5	10/230B	4.0	0.4	22.2	20.0	3.5×	5.4	**100.0**	15.1	**100.0**	7.1	14.0	8.0
GLM-4.7	32/355B	12.0	6.0	89.6	20.0	**8.6**×	3.3	**100.0**	6.0	**100.0**	8.1	19.0	-
GLM-5	40/744B	38.0	18.8	**90.2**	54.0	1.87×	1.6	**100.0**	1.2	**100.0**	16.2	23.0	4.0
Kimi-K2.5	32B/1T	40.0	12.1	81.0	46.0	1.9×	12.5	**100.0**	7.8	**100.0**	13.1	23.0	6.0
Kimi-K2-Instruct	32B/1T	2.0	1.1	69.6	12.0	1.1×	15.8	96.5	13.9	91.3	6.1	-	-
Kimi-K2-Thinking	32B/1T	48.0	20.0	-	24.0	1.2×	17.4	**100.0**	19.9	84.8	9.1	16.0	4.0
DeepSeek-V3.2	37/671B	14.0	4.6	84.4	30.0	1.8×	19.6	**100.0**	18.1	13.3	3.0	-	-
Qwen3.5-397B-A17B	17/397B	36.0	14.2	17.8	34.0	1.2×	7.6	**100.0**	16.3	92.6	4.0	10.0	-
Qwen3-Coder-480B-A35B-Instruct	35/480B	10.0	4.7	9.0	64.0	2.0×	**20.1**	**100.0**	**31.9**	56.6	3.0	6.0	-
InCoder-32B	32B	82.0	**53.5**	35.2	91.0	1.3×	18.5	**100.0**	19.3	93.8	**22.2**	36.0	**14.0**
InCoder-32B-Thinking	32B	**84.0**	48.6	47.9	**93.0**	3.93×	15.2	**100.0**	22.9	89.5	20.2	**38.0**	12.0
Closed-APIs Models													
Claude-Sonnet-4.6	🔒	77.0	32.4	79.0	88.0	4.6×	28.8	98.1	41.6	1.4	11.1	28.0	2.0

4 Analysis

4.1 ICWM Fidelity Analysis

ICWM is useful only if its feedback is close enough to real backend feedback to guide a generator. We test this by holding out 2,000 execution turns per domain from $\mathcal{D}_{\text{real}}$ and comparing ICWM outputs with real backend labels. We report two metrics: **outcome prediction accuracy**, the share of turns whose predicted label matches the backend label, and **trajectory match rate**, the share of full traces whose final pass/fail verdict is the same under real feedback and ICWM feedback.

Figure 3 shows mean outcome accuracy of 96.7% and a mean trajectory match rate of 94.4%. Chip design is easiest for the proxy: it reaches 97.4% outcome accuracy and a 95.8% trajectory match rate, largely because Yosys and Icarus give structured, repeated messages. 3D modeling is harder, at 95.9% and 93.1%, because CadQuery checks can depend on floating-point tolerances and implicit Boolean operations that are not clear from code tokens alone. GPU kernels, code optimization, and embedded systems sit between these cases, with outcome scores from 96.2% to 97.1%.

The 1.6 to 2.8 point drop from outcome score to

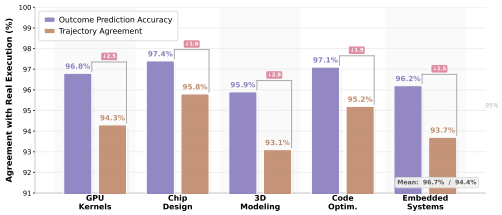

Figure 3: ICWM fidelity: outcome/trace agreement; dashed line marks 95%.

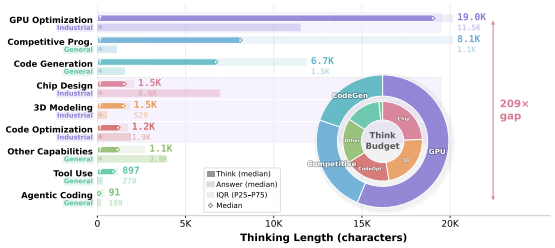

Figure 4: Thinking and answer lengths by task category. Bars show medians; bands show P25–P75; purple marks industrial domains.

trajectory match rate is expected. A single wrong feedback prediction can send the generator down a different repair path. The lowest match rate is still 93.1%, so most ICWM-guided traces reach the same final verdict as real execution. This does not replace final checks, but it is accurate enough for data synthesis with audit rounds.

GPU kernel diagnosis. A Triton fused-attention implementation uses a `tl.zeros([BLOCK_M, BLOCK_N])` tile in shared memory and launches with `num_warps=8`, pushing memory use past the per-SM shared-memory limit once compiler and runtime overhead are included. The real Triton compiler returns MEMORY_FAULT with diagnostic *"shared memory request (49152 B) exceeds per-SM limit"*. ICWM predicts the same MEMORY_FAULT label with a close diagnostic message. In the next turn, the generator reduces `BLOCK_N` from 128 to 64; ICWM predicts PASS, matching real execution. This example matters because the repair is causal, not cosmetic: the model must connect tile shape, warp configuration, and shared-memory pressure.

3D modeling false positive. A CadQuery script constructs a mounting bracket by subtracting a cylindrical bore from a rectangular body. The candidate code places the cylinder axis exactly tangent to one face, creating a degenerate zero-thickness edge. Real CadQuery raises GEOMETRY_ERROR with diagnostic *"BRep check: edge has zero length"*. ICWM classifies this turn as PASS because the code is syntactically valid and the specified dimensions look plausible, producing a false positive. This type of geometric corner case accounts for the wider fidelity gap observed in 3D modeling. During the audit rounds described in subsection 2.3, real-execution spot checks catch these mismatches and add the corrected labels back into ICWM training.

4.2 Adaptive Thinking Depth

Figure 4 plots the length of `<think>` blocks across task categories. The median ranges from 91 characters per step in agentic coding to 19,015 characters in GPU kernel optimization, a 209× spread. We did not set this distribution by hand. It emerges from the correction process in section 2: tasks with dense backend feedback collect longer traces over several repair turns, while simple errors need little explanation.

The pattern matches the backends. For GPU optimization, a repair may involve grid shape, block shape, data movement, shared-memory layout, and warp setup, so this domain needs the longest traces. Chip design has a different shape: the median reason block is only 1.5K characters, but the RTL output is much longer at 6.9K characters. Yosys and Icarus often return concise structural errors, while the needed fix may require a long module. Agentic coding has the shortest per-step thinking because reasoning is spread across many terminal actions.

Because traces in $\mathcal{D}$ come from real or ICWM-simulated feedback, they do not share one template length. The model sees short reasoning for obvious backend errors and longer reasoning when repairs must account for hardware behavior. This is the desired behavior: the reasoning budget follows the feedback cycle rather than a fixed prompt.

4.3 Effects of Thinking Training Data

We next examine how the amount of thinking data affects performance. InCoder-32B was trained with 250M SFT tokens, while our thinking set reaches 540M tokens because correction traces are much longer than final answers. We train models with 180M, 360M, and 540M thinking tokens and test them on the industrial suite. As shown in Figure 5, most metrics improve as the set grows: VeriScope rises from 61.8 to 75.4, and KernelBench L2 rises from 16.0 to 38.0.

A few metrics stay flat, which is informative. TritonBench GPU execution correctness stays at 100 across all three stages, suggesting that the smaller

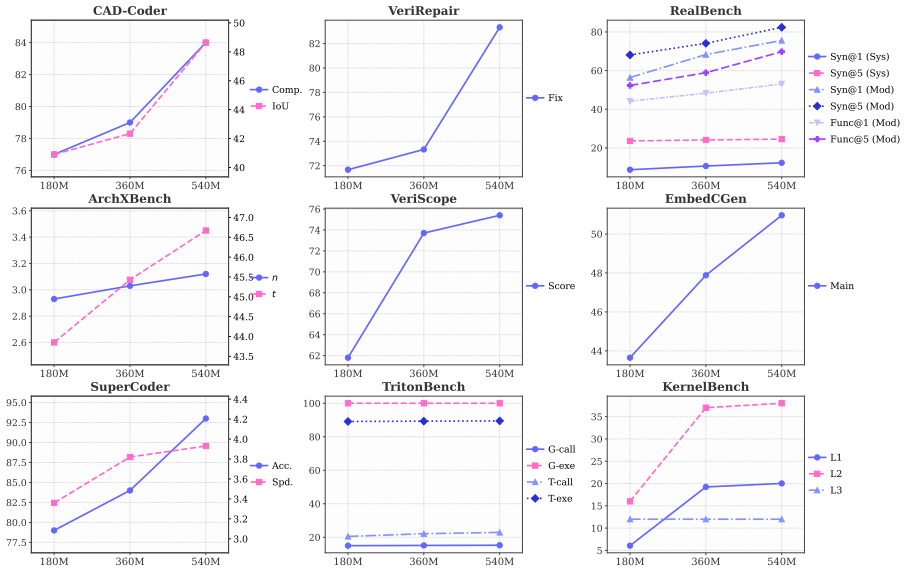

Figure 5: Scaling with thinking data on nine industrial benchmarks.

model already solves the basic run cases covered by this metric. KernelBench L3 also stays at 12.0, which points to a different limit: the hardest tuning tasks may need stronger search or domain tactics rather than more repair traces alone. Overall, more thinking data improves most domain tests, while plateaued metrics show where data volume alone is not enough.

5 Related Work

Industrial Code Intelligence Industrial code uses specialized syntax and backend checks, where small semantic errors can break timing, firmware behavior, geometry, or hardware correctness. Prior work covers RTL generation and repair (Thakur et al., 2023; Liu et al., 2024a; Du et al., 2025), Verilog debugging and formal checks (Wang et al., 2025a; Yubeaton et al., 2025), hardware-reward RL (Zhu et al., 2025; Wang et al., 2025b), RTL benchmarks (Liu et al., 2023b; Jin et al., 2025), GPU kernels (Baronio et al., 2025; Dai et al., 2026; Cao et al., 2026), firmware (Yang et al., 2024; Xu et al., 2025), compiler optimization (Wei et al., 2026; Cummins et al., 2025), and CAD (Guan et al., 2025; Li et al., 2025b). InCoder-32B (Yang et al., 2026) narrows this gap with a shared industrial code model. Building on it, InCoder-32B-Thinking adds reasoning traces for complex industrial code.

Thinking in Large Language Models OpenAI o1 (OpenAI, 2024) showed that RL can train LLMs to spend more compute on reasoning; later systems include o3 (OpenAI, 2025), Gemini 3 (Google DeepMind, 2025), DeepSeek-R1 (Guo et al., 2025), QwQ (Team, 2025b), and Qwen3 (Yang et al., 2025a). Long traces are usually built through RL or distillation: GRPO (Shao et al., 2024) removes PPO's critic (Schulman et al., 2017), distillation transfers chain-of-thought behavior (Deng et al., 2024; Ahmad et al., 2025), and STILL-2 (Min et al., 2024) combines distillation with search. Code LLMs can also learn from tool feedback, as in Self-Refine (Madaan et al., 2023), LeDex (Jiang et al., 2024), and RLEF (Gehring et al., 2024). Structured prompting (Li et al., 2025a), o1-Coder (Zhang et al., 2024), and rStar-Coder (Liu et al., 2025b) adapt thinking to code, but not to domains where backend traces decide correctness. InCoder-32B-Thinking trains on multi-turn execution-grounded repairs and uses ICWM to synthesize more traces with real toolchain audits.

6 Conclusion

InCoder-32B-Thinking is a 32B thinking code model trained from environment-grounded repair traces. ECoT keeps failed attempts, tool feedback, and repairs; ICWM scales the data by predicting toolchain feedback for more traces. Across 23 tests (14 general and 9 industrial), InCoder-32B-Thinking reaches 81.3% on LiveCodeBench V5 and posts strong open-weight results across RTL design, GPU kernels, firmware, code optimization, and CAD. The analysis shows that thinking length follows backend hardness, ICWM matches most real execution verdicts, and more thinking data improves several industrial scores. The remaining proxy errors and plateaued hard tasks motivate stronger search and final real-backend checks.

7 Limitations

Our results depend on the quality of the execution tools used to build the traces. ICWM is accurate enough for synthesis, but it can still be wrong, in particular on geometric edge cases and on traces where one wrong label changes the fix path. For that reason, ICWM feedback should be treated as a data-generation tool, not as a substitute for final backend checks.

The benchmark suite is broad but not exhaustive. It covers several industrial domains, but it does not include every hardware target, compiler stack, CAD kernel, or embedded platform used in practice. The strongest gains also appear on tasks where backend feedback is informative. Tasks that need open-ended search or deep optimization, such as the hardest KernelBench level, may need search loops and domain search methods in addition to more thinking data.

8 Ethical Considerations

Better industrial code models can reduce the cost of writing and debugging hardware-aware software. This is useful for chip design, embedded systems, GPU kernels, and CAD workflows where expert time is scarce.

The same capability also has risks. A model that generates RTL, firmware, or optimized kernels can produce unsafe or unauthorized hardware logic if used without review. We therefore view InCoder-32B-Thinking as an assistant for expert-supervised development. Generated code should still pass the relevant compiler, simulator, testbench, profiler, and safety review before deployment.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*.

Zhipu AI. 2025. Glm-4.7: Advancing the coding capability. https://z.ai/blog/glm-4.7.

Anthropic. 2026. Introducing claude sonnet 4.6.

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

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. tau2-bench: Evaluating conversational agents in a dual-control environment.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, et al. 2026. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*.

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. 2021. Evaluating large language models trained on code.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. 2025. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large language models for compiler optimization. *CoRR*, abs/2309.07062.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pages 141–153.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. 2026. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*.

DeepSeek AI. 2024. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114.

Yuntian Deng, Yejin Choi, and Stuart Shieber. 2024. From explicit cot to implicit cot: Learning to internalize cot step by step. *arXiv preprint arXiv:2405.14838*.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. 2025. Swe-dev: Evaluating and training autonomous feature-driven software development.

Zachary Englhardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak N. Patel, and Vikram Iyer. 2024. Exploring and characterizing large language models for embedded system development and debugging. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems, CHI EA 2024, Honolulu, HI, USA, May 11-16, 2024*, pages 150:1–150:9. ACM.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.

Google DeepMind. 2025. Gemini 3 pro: the frontier of vision ai. `https://blog.google/innovation-and-ai/technology/developers-tools/gemini-3-pro-vision/`.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Yandong Guan, Xilin Wang, Ximing Xing, Jing Zhang, Dong Xu, and Qian Yu. 2025. Cad-coder: Text-to-cad generation with chain-of-thought and geometric reward.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

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. *arXiv preprint arXiv:2403.07974*.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya B Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Ledex: Training llms to better self-debug and explain code. *Advances in Neural Information Processing Systems*, 37:35517–35543.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, et al. 2025. Realbench: Benchmarking verilog generation models with real-world ip designs. *arXiv preprint arXiv:2507.16200*.

Mohammad S Khan, Sankalp Sinha, Talha U Sheikh, Didier Stricker, Sk A Ali, and Muhammad Z Afzal. 2024. Text2cad: Generating sequential cad designs from beginner-to-expert level text prompts. *Advances in Neural Information Processing Systems*, 37:7552–7579.

Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. 2025. SWE-bench Multilingual.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025a. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23.

Jiahao Li, Yusheng Luo, Yunzhong Lou, and Xiangdong Zhou. 2025b. Recad: Reinforcement learning enhanced parametric cad model generation with vision-language models. *arXiv preprint arXiv:2512.06328*.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025c. Tritonbench: Benchmarking large language model capabilities for generating triton operators.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Fei Huang, and Yongbin Li. 2023. Can llm

already serve as a database interface? a big bench for large-scale database grounded text-to-sqls.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023b. VerilogEval: evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Mingjie Liu, Nathaniel Ross Pinckney, Brucek Khailany, and Haoxing Ren. 2023c. Invited paper: Verilogeval: Evaluating large language models for verilog code generation. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*, pages 1–8. IEEE.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024a. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461.

Siyao Liu, Ge Zhang, Boyuan Chen, Jialiang Xue, and Zhendong Su. 2024b. FullStack Bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. 2025b. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*.

Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. 2026. Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents. *IEEE Transactions on Software Engineering*.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. Mcp-universe: Benchmarking large language models with real-world model context protocol servers.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yingqian Min, Zhipeng Chen, Jinhao Jiang, Jie Chen, Jia Deng, Yiwen Hu, Yiru Tang, Jiapeng Wang, Xiaoxue Cheng, Huatong Song, Wayne Xin Zhao, Zheng Liu, Zhongyuan Wang, and Ji-Rong Wen. 2024. Imitate, explore, and self-improve: A reproduction report on slow-thinking reasoning systems.

MiniMax. 2026. Minimax m2.5: Built for real-world productivity. https://www.minimax.io/news/minimax-m25.

OpenAI. 2024. Introducing SWE-bench Verified. Updated February 24, 2025.

OpenAI. 2024. Learning to reason with large language models. https://openai.com/index/learning-to-reason-with-llms/.

OpenAI. 2025. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels?

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 48371–48392. PMLR.

Suresh Purini, Siddhant Garg, Mudit Gaur, Sankalp Bhat, Sohan Mupparapu, and Arun Ravindran. 2025. Archxbench: A complex digital systems benchmark suite for llm driven rtl synthesis. In *Proceedings of the 7th ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*.

Qwen. 2025. Qwen3-coder: Agentic coding in the world.

Qwen Team. 2026a. Qwen3-coder-next technical report.

Qwen Team. 2026b. Qwen3.5: Towards native multimodal agents.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

SWE-bench Live Team. 2025. SWE-bench Live.

ByteDance Seed Team. 2025a. Seed-oss open-source models. https://github.com/ByteDance-Seed/seed-oss.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. 2026. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Qwen Team. 2025b. Qwq-32b: Embracing the power of reinforcement learning.

The Terminal-Bench Team. 2025c. Terminal-bench: A benchmark for ai agents in terminal environments.

Shailja Thakur, Baleegh Ahmad, Hammond Fan, et al. 2023. VeriGen: A large language model for verilog code generation. *arXiv preprint arXiv:2308.00708*.

Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Zhe Jiang, and Nan Guan. 2025a. Veridebug: A unified llm for verilog debugging via contrastive embedding and guided correction. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 61–67. IEEE.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. 2025b. Verireason: Reinforcement learning with testbench feedback for reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.11849*.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. 2026. Supercoder: Assembly program superoptimization with large language models.

Ruiyang Xu, Jialun Cao, Mingyuan Wu, Wenliang Zhong, Yaojie Lu, Ben He, Xianpei Han, Shing-Chi Cheung, and Le Sun. 2025. Embedagent: Benchmarking large language models in embedded system development. *arXiv preprint arXiv:2506.11003*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2024. Embedgenius: Towards automated software development for generic embedded iot systems. *arXiv preprint arXiv:2412.09058*.

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

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, et al. 2025b. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Patrick Yubeaton, Andre Nakkab, Weihua Xiao, Luca Collini, Ramesh Karri, Chinmay Hegde, and Siddharth Garg. 2025. Verithoughts: Enabling automated verilog code generation using reasoning and formal verification. *arXiv preprint arXiv:2505.20302*.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chengxing Xie, Cunxiang Wang, et al. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Zizheng Zhan, Ken Deng, Xiaojiang Zhang, Jinghui Wang, Huaixi Tang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, Wenhao Zhuang, et al. 2025. Kat-coder technical report. *arXiv preprint arXiv:2510.18779*.

Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. 2024. o1-coder: an o1 replication for coding. *arXiv preprint arXiv:2412.00154*.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, et al. 2025. Qimeng-codev-r1: Reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.24183*.

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 Detailed General-Code Results

The main paper focuses on reasoning-heavy and toolchain-bound results because these benchmarks best expose the effect of environment-grounded correction traces. Table 5 gives the complementary code-generation results on EvalPlus, BigCodeBench, and FullStackBench. These results show that InCoder-32B-Thinking preserves broad generation ability while the larger gains appear on tasks that require longer reasoning or backend feedback.

Reading Table 5. The table reports pass rates for short Python synthesis, library-heavy generation, and full-stack programming. InCoder-32B-Thinking is close to the strongest open models on HumanEval, MBPP, and BigCodeBench, while its main advantage appears later in the reasoning and toolchain-bound tables. This pattern supports the intended trade-off: ECoT and ICWM improve difficult feedback-driven tasks without collapsing standard code-generation ability.

B Additional Figure Details

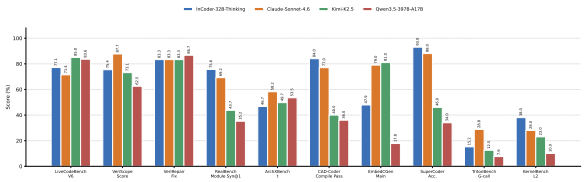

Figure 6: Benchmark overview for InCoder-32B-Thinking across general code and industrial code tasks.

Reading Figure 6. The figure groups the evaluation into two blocks. The general-code block checks whether InCoder-32B-Thinking remains a useful software model after thinking-oriented training: it covers short function synthesis, code reasoning, text-to-SQL, repository repair, terminal tasks, and tool use. The industrial block targets the part that ordinary code benchmarks miss: whether generated programs survive a backend such as a Verilog simulator, a GPU compiler, a firmware emulator, or a CAD kernel. This distinction matters because a model can pass many general code tasks while still failing on industrial code for backend-specific reasons.

The main pattern is that InCoder-32B-Thinking keeps broad coding ability while gaining most on benchmarks where feedback is diagnostic rather than merely binary. LiveCodeBench V5 benefits from longer reasoning over algorithmic constraints. RealBench, CAD-Coder, SuperCoder, and KernelBench benefit from traces where the model sees a failed candidate, the environment's complaint, and a repair. The weaker areas are also informative: Text-to-SQL and some code-efficiency metrics reward concise direct answers, while the hardest KernelBench level requires optimization search beyond the current trace data.

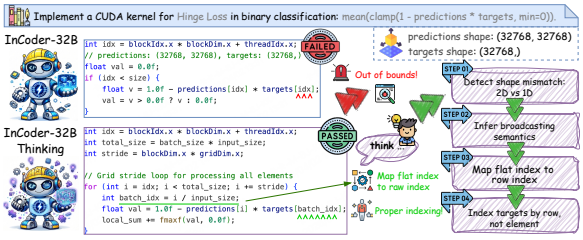

Figure 7: CUDA Hinge Loss example. InCoder-32B-Thinking notices that the prediction tensor is two-dimensional while the target tensor stores one label per row, so it maps each flat element index back to the row index before reading the target. The non-thinking baseline reuses the flat index for both tensors and can read past the target buffer.

Reading Figure 7. The example shows why a thinking trace is useful for industrial code. The task is not only to write a CUDA kernel, but to reason through how the flattened thread index maps onto tensors with different ranks. The prediction tensor has shape `[N, C]`, while the target tensor stores one label per row and is treated as `[N]` or explicitly reshaped to `[N, 1]` for broadcasting. A flat index over `N * C` prediction elements therefore cannot be used directly to index the target array. The correct mapping first recovers the row index, then reads the target for that row. This small indexing detail is exactly the kind of error that causes runtime failures in generated kernels.

The non-thinking baseline treats both arrays as if they shared the same flat indexing scheme. That code can compile and still be wrong, because the target array is shorter than the prediction buffer. InCoder-32B-Thinking instead explains the shape mismatch before writing the final code. This is the behavior that ECoT is designed to teach: use the observed or anticipated backend failure to identify the real cause, then repair the program at the level of semantics rather than changing code superficially.

Table 5: General code generation results.

Model	Size	EvalPlus				BigCodeBench		FullStackBench
		HumanEval	HumanEval+	MBPP	MBPP+	Full	Hard	
6B+ Models								
DeepSeek-Coder-V2-Lite-Instruct	2.4/16B	81.1	75.6	85.2	70.6	37.8	18.9	49.4
Qwen2.5-Coder-7B-Instruct	7B	**87.2**	**81.7**	84.7	72.2	37.8	13.5	42.2
Seed-Coder-8B-Instruct	8B	81.1	75.6	86.2	73.3	44.6	**23.6**	**55.8**
Qwen2.5-Coder-14B-Instruct	14B	62.8	59.8	**88.6**	**77.2**	**47.0**	6.1	53.1
30B+ Models								
Qwen3-Coder-30B-A3B-Instruct	3.3/30.5B	93.9	87.2	90.7	77.2	46.9	27.7	60.9
DeepSeek-V3.2	37/671B	93.9	88.4	93.4	77.2	48.1	27.0	64.9
Qwen2.5-Coder-32B-Instruct	32B	93.3	86.6	90.2	77.8	48.0	24.3	57.4
Qwen3-235B-A22B-Instruct-2507	22/235B	96.3	91.5	92.3	77.8	47.4	25.7	62.7
Qwen3-235B-A22B-Thinking-2507	22/235B	**98.8**	**93.3**	95.5	81.5	44.1	23.0	-
Qwen3-Coder-480B-A35B-Instruct	35/480B	97.6	92.7	94.2	80.2	49.4	27.7	66.4
Kimi-Dev-72B	72B	93.3	86.0	79.6	68.8	45.4	**31.8**	38.6
Kimi-K2-Instruct-0905	32B/1T	94.5	89.6	91.8	74.1	**49.8**	30.4	63.5
Kimi-K2-Thinking	32B/1T	98.2	92.7	**97.4**	**82.3**	46.8	28.4	-
KAT-Dev	32B	90.9	86.6	89.4	76.2	46.2	25.7	58.8
KAT-Dev-72B-Exp	72B	88.4	81.7	85.2	69.3	48.3	26.4	52.9
GLM-4.7	32/355B	87.2	79.9	90.5	75.7	45.7	26.4	**70.2**
InCoder-32B	32B	94.5	89.6	91.8	78.3	**49.8**	31.1	57.1
InCoder-32B-Thinking	32B	95.1	89.6	92.1	78.3	47.4	29.1	60.8

C Benchmark Details

C.1 General Code Benchmarks

Code Generation. **EvalPlus** (Liu et al., 2023a) extends HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with more tests, making short Python solutions harder to pass by accident. **BigCodeBench** (Zhuo et al., 2024) contains 1,140 prompts that require calling and composing common Python libraries. **FullStackBench** (Liu et al., 2024b) covers 16 programming languages and more than 4,000 problems across application domains.

Code Reasoning. **CRUXEval** (Gu et al., 2024) tests execution tracing through input and output prediction on 800 short Python programs. **LiveCodeBench** (Jain et al., 2024) collects competitive-programming problems from LeetCode, AtCoder, and Codeforces with time-stamped splits; we evaluate on V5 and V6.

Code Efficiency. **Mercury** (Du et al., 2024) measures whether generated solutions are both correct and faster than a reference, using Beyond@1 and Pass@1.

Text-to-SQL. **Spider** (Yu et al., 2018) and **BIRD** (Li et al., 2023) ask models to translate natural-language questions into SQL over unseen databases, testing schema linking and joins.

Agentic Coding. **Terminal-Bench** (Team, 2025c) evaluates multi-turn terminal work for software engineering tasks. **SWE-bench Verified** (OpenAI, 2024) uses real GitHub issues that require understanding the repository, writing a patch, and passing tests.

Tool Use. **Mind2Web** (Deng et al., 2023) tests web agents that select elements and actions on real websites. **BFCL V3** (Patil et al., 2025) evaluates function-calling accuracy across API schemas. **τ^2-bench** (Barres et al., 2025) evaluates tool-using dialogue agents in Airline, Retail, and Telecom settings.

C.2 Industrial Code Benchmarks

C.2.1 Chip Design

VeriScope. VeriScope contains 568 Verilog generation problems in five difficulty tiers. Lower tiers cover combinational logic and module composition. Higher tiers include system-level designs, including a dual-core out-of-order RISC-V SoC with cache coherence. Simulation assigns 0 points to code that does not compile, 50 points to code that compiles but fails the functional testbench, and 100 points to code that passes all tests. We report the average score.

RealBench. RealBench (Jin et al., 2025) targets IP-level Verilog rather than small snippets. It is built from four open-source IP cores, including AES, an SD card controller, and the Humming-

birdv2 E203 CPU. The benchmark has 60 module-level subtasks, where missing submodules may use golden references, and 4 system-level subtasks that require implementing the whole hierarchy from a top-level specification. We generate 20 samples per task. Syn@k checks whether at least one of k candidates compiles; Func@k then runs testbench simulation on compilable candidates. In Table 3, we report system-level Syn@1/Syn@5 and module-level Syn@1/Syn@5 plus Func@1/Func@5.

ArchXBench. ArchXBench (Purini et al., 2025) contains 51 digital-system designs across six difficulty levels. The tasks cover cryptography, signal processing, image processing, and machine learning, ranging from arithmetic circuits to AES cores and streaming FFT/DCT pipelines. Each task includes a description, interface specification, and testbench. We sample five candidates per task and report (n, t), where n is the number of syntactically correct candidates and t is the percentage of testbench assertions passed by the best candidate.

VeriRepair. VeriRepair is a Verilog diagnosis and repair benchmark built by injecting realistic bugs into verified implementations. Its taxonomy has 4 major categories and 20 fine-grained error types, including syntax errors, type and structural errors, timing and FSM errors, and semantic logic errors. The dataset contains about 22,000 training samples and 300 test samples. Each sample includes buggy code, error category, error location, a corrected reference, and a testbench. We report Fix%, the fraction of buggy programs repaired successfully under testbench execution.

C.2.2 GPU Kernel Optimization

KernelBench. KernelBench (Ouyang et al., 2025) has 250 PyTorch ML workloads in three levels. Level 1 uses single operators, Level 2 uses operator sequences suitable for fusion, and Level 3 uses full model architectures. Given a PyTorch reference, the model must produce a functionally equivalent faster version. We use fast_p, the fraction of tasks where the generated kernel is correct and at least $p\times$ faster than the PyTorch baseline. We report fast_1 for L1, L2, and L3.

TritonBench. TritonBench (Li et al., 2025c) evaluates Triton operator generation. TritonBench-G contains 184 operators collected from GitHub repositories across five difficulty levels, while TritonBench-T aligns tasks with standard PyTorch operator interfaces. For each track, call accuracy checks whether the code runs without runtime error, and execution accuracy checks whether the output matches the reference among successfully callable cases. We report G-call, G-exe, T-call, and T-exe.

C.2.3 Code Optimization

EmbedCGen. EmbedCGen contains 500 bare-metal embedded C problems for resource-constrained microcontrollers. The tiers range from peripheral control and register operations to multi-peripheral integration, DMA, and state-machine coordination. Generated code must follow HAL conventions and satisfy hard real-time constraints. Evaluation runs code generation, ARM GCC cross-compilation, and Renode simulation in sequence. We report the average pass rate.

SuperCoder. SuperCoder (Wei et al., 2026) formulates assembly superoptimization as language modeling. Given a C program and its `gcc -O3` assembly, the model must produce semantically equivalent assembly that runs faster. The benchmark has 8,072 x86-64 programs, averaging 130 lines, with tests covering 96.2% of lines and 87.3% of branches. We report accuracy under the test suite and speedup over the compiler-optimized baseline.

C.2.4 3D Modeling

CAD-Coder. CAD-Coder (Guan et al., 2025) asks models to produce executable CadQuery scripts, a Python parametric CAD language built on OpenCascade. It is derived from Text2CAD (Khan et al., 2024) and contains 110K verified text–CadQuery–3D model triplets, split into 8K high-quality, 70K medium-quality, and 32K hard cases, plus 1.5K chain-of-thought annotated samples. Compilation success measures whether the script executes and produces valid geometry. IoU measures volumetric overlap between generated and reference voxelizations.