CHIPMATE: TRAINING VERILOG AND REFERENCE-MODEL AGENTS FOR RTL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic RTL generation faces deployment constraints when a golden testbench is unavailable, external LLM APIs cannot be used, and models must support adaptation to proprietary RTL codebases. Self-trained local models address model access and adaptation, but single-turn generation provides no iterative verification workflow. To bridge these gaps, we present ChipMATE, a self-trained multi-agent framework for RTL generation. Inspired by cross-comparison between independently written RTL modules and reference models, ChipMATE pairs a Verilog agent with a separately parameterized Python reference-model agent; the two agents cross-check their outputs without a golden oracle at inference. Training uses correctness oracles. We design a backtrack-based inference workflow to limit error propagation across turns, retaining a new pair only when its measured agreement improves, and a two-stage training pipeline that first strengthens each agent individually, then trains the team jointly using X-GRPO with partner-averaged candidate credit. To support the training, we further build a hybrid data-generation framework that produces 64.4K reference-model samples checked by finite differential simulation. ChipMATE achieves 75.0% and 80.1% final-output pass@1 on VerilogEval v2 with 4B and 9B parameters per agent; each output comes from a complete iterative workflow. We plan to open-source our model weights, inference workflow, and training dataset to facilitate future research.

1 INTRODUCTION

Large Language Models (LLMs) excel at general-purpose programming but struggle with Register-Transfer Level (RTL) code generation, a cornerstone of modern chip design. Recent agentic workflows like MAGE (Zhao et al., 2025) and VerilogCoder Ho et al. (2025) improve RTL quality by orchestrating LLMs through task decomposition and self-correction, with promising results.

Deploying agentic RTL generation in a chip-design organization, however, faces three coupled constraints. **First**, a trusted testbench is not yet available when a module is generated. Verification artifacts may be developed alongside the RTL and can themselves contain bugs, making them unsuitable as golden oracles. **Second**, design data must remain on controlled infrastructure, precluding calls to external LLM APIs. **Third**, the generator must support adaptation to internal RTL codebases, allowing locally available examples to contribute to model improvement. These constraints call for a workflow that is trained and deployed locally and tolerates imperfect verification feedback.

Recent efforts such as QiMeng-CodeV-R1 Zhu et al. (2025) and RTLSeek (Zhang et al., 2026) train open-source LLMs on local servers, removing external-API dependence and enabling training on in-house codebases. Used as single-turn generators, however, these models offer no mechanism to check or correct their own output. Chip-design practice suggests a division of labor: a design engineer implements a module in RTL, while a verification engineer develops an independent behavioral reference in a high-level language (*e.g.*, Python, SystemC, or C++). Neither implementation is treated as infallible; comparing their outputs helps locate bugs and guide fixes. This workflow directly inspires ChipMATE, a multi-agent system in which one agent generates Verilog in the role of a design engineer and another generates a cycle-accurate Python reference model in the role of a verification engineer. The two agents iteratively cross-verify their outputs without seeing each other's code.

However, building such a self-trained multi-agent workflow introduces three key challenges. **(C1) Error propagation without a golden oracle.** Since neither agent generates perfect code, a mismatch

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

between the two agents does not reveal which side is wrong. Naively asking one agent to "correct" the other can compound errors turn by turn, falling below a single-model baseline. Agreement can also reflect a shared misreading of the specification: separate implementations do not imply independent errors. **(C2) Individual capability and collaboration.** Each agent must learn its own task as well as how to use mismatch feedback from a fallible partner. Reference-model generation requires cycle-accurate behavior, reset handling, and fixed-width arithmetic, which general Python proficiency alone does not ensure. **(C3) Training data for reference-model generation.** Existing RTL corpora supervise the design agent but do not supply cycle-accurate Python reference models. Distilling it from API-based LLMs is costly and low-yield: DeepSeek-R1 converts only 17% of the QiMeng-CodeV corpus in a single turn (Section 2.3).

We present ChipMATE, a self-trained multi-agent framework for RTL generation with fully offline inference: training uses reference implementations as correctness oracles, while the deployed workflow needs no golden testbench or trusted reference implementation. Our contributions are:

1. **Cross-verification multi-agent workflow (C1).** A Verilog agent and a Python reference-model agent compare execution traces and repair their own implementations. To limit error propagation, a joint backtracking mechanism accepts a new pair only when its measured agreement improves, which preserves observed agreement rather than ground-truth correctness. Cross-language comparison tools align Verilog and Python outputs, and error-explanation tools translate raw waveform mismatches into natural-language diagnostics.
2. **Two-stage training pipeline (C2).** Stage 1 trains each agent separately with SFT followed by RL on Verilog module and Python reference-model generation. Stage 2 trains the agents jointly within the cross-verification workflow. Our X-GRPO algorithm evaluates each candidate against multiple partner candidates and averages its reward across partners before within-agent advantage normalization, while correctness rewards from training oracles reward repairs that also match the training reference.
3. **Hybrid data-generation framework for reference models (C3).** We combine agentic API-based synthesis with IR-level code conversion to transform existing Verilog datasets into Python reference models, then augment categories where the trained agent struggles. The resulting 64.4K samples pass finite differential simulation against the source Verilog.

On VerilogEval v2, ChipMATE achieves 75.0% and 80.1% final-output pass@1 with 4B and 9B parameters per agent, respectively (Table 1); each output is returned by an iterative workflow with multiple internal samples. Same-framework backend and role-swap controls support the contribution of both trained roles (Section 3.2). We plan to release the model weights, multi-agent workflow, and 64.4K reference-model training samples to facilitate future research.

2 METHODS

As shown in Figure 1, ChipMATE consists of a cross-verification multi-agent workflow (Section 2.1), a two-stage pipeline that trains the agents individually, then jointly (Section 2.2), and a hybrid framework that creates reference-model data from Verilog corpora (Section 2.3).

The two agents have separate parameters and receive the same specification and interface, but never see each other's generated code. Training rewards correctness against known reference implementations. At inference, no golden testbench or reference implementation is available: candidate selection, backtracking, and stopping use only inter-agent agreement, and held-out benchmark tests score the returned Verilog *after* generation. Agreement is thus a selection signal, not a certificate.

2.1 CROSS-VERIFICATION MULTI-AGENT WORKFLOW

Multi-turn cross-verification workflow. As shown in Figure 1(a), our workflow proceeds in three steps. First, the Verilog agent and the Python reference-model agent each independently sample N candidate implementations from the same natural-language specification. Python leverages the base model's general programming capability, while dedicated training adds hardware semantics. Second, candidate j from the Verilog agent is paired with candidate j from the Python agent, yielding N index-wise pairs. The cross-language comparator simulates all pairs on the same fixed suite $\mathcal{S}$ of 1,000 randomly generated stimuli and produces structured mismatch diagnostics, and the pair with

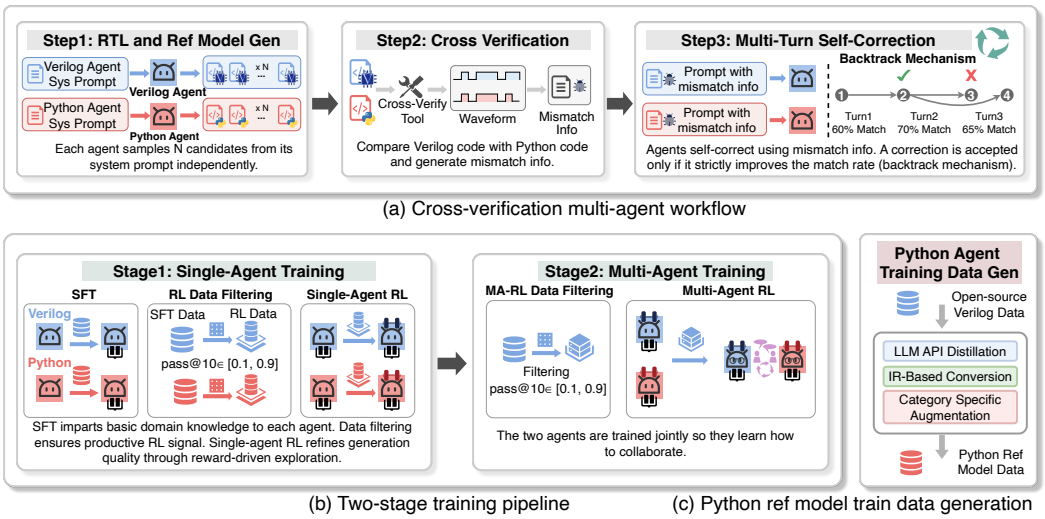

Figure 1: Overview of ChipMATE. (a) Cross-verification with joint backtracking. (b) Role-specific training followed by joint RL. (c) Hybrid reference-model data generation.

the highest match rate becomes the accepted pair. Third, if outputs disagree, each agent receives the mismatch diagnostics and self-corrects over multiple turns, resampling N candidates per turn conditioned on its own previous code. Reusing $\mathcal{S}$ keeps acceptance scores comparable across turns.

Joint backtracking. Let $p_t = (v_t, p_t^{\text{py}})$ be the accepted Verilog/Python pair after turn t, $m(p; \mathcal{S}) \in [0, 1]$ its output match rate on $\mathcal{S}$, and $\widetilde{p}_t$ the highest-match challenger at turn t:

$$p_t = \widetilde{p}_t \;\text{ if }\; m(\widetilde{p}_t; \mathcal{S}) > m(p_{t-1}; \mathcal{S}), \qquad p_t = p_{t-1} \;\text{ otherwise.} \tag{1}$$

Both agents thus retain their previous accepted code unless the challenger strictly improves agreement, making backtracking a joint pair decision. The loop stops at full agreement or after T total turns and returns the Verilog member of the accepted pair. The rule makes the accepted match rate on $\mathcal{S}$ nondecreasing; it does not ensure improved functional correctness or formal equivalence. The default $N = 3$, $T = 3$ permits at most $2NT = 18$ model completions and $NT = 9$ pair simulations per returned RTL design, and early stopping reduces the realized cost.

Comparison and feedback infrastructure. The cross-verification tool compiles the Verilog module via Icarus Verilog, executes the Python reference model, and compares their cycle-by-cycle outputs. Since raw waveform files are difficult for LLMs to parse, a waveform-to-natural-language converter locates the first divergent cycle and packages the surrounding input/output context into a structured description that agents can act upon. The feedback explicitly allows either side to be wrong, so each agent must judge whether its own implementation is at fault. The system prompts (Appendix D) contain the exact module interface or Python class skeleton, combinational- and sequential-logic guidance, and a reasoning-then-code example.

2.2 Two-Stage Training Pipeline

As shown in Figure 1(b), the first stage trains each agent independently for its code-generation task, and the second stage trains both agents jointly to collaborate within the cross-verification workflow.

Stage 1: Single-agent training. We train the Verilog agent and the Python agent separately on their respective datasets using supervised fine-tuning (SFT) followed by RL. GRPO (Shao et al., 2024) samples a group of candidate outputs for each query and derives group-relative advantages from executable correctness rewards. For RL, we retain problems of intermediate post-SFT difficulty to favor groups containing both successful and unsuccessful solutions; the precise sampling statistic and cutoff for this filter remain unspecified. SFT and RL in both stages use full-parameter updates.

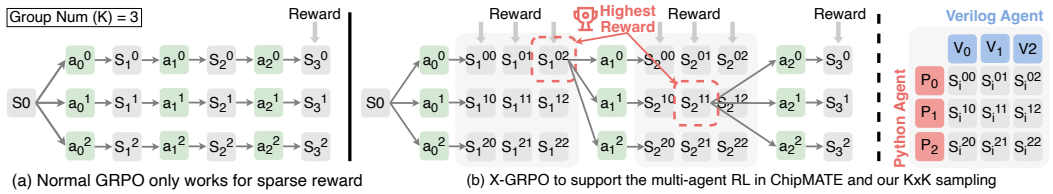

Figure 2: Per-turn candidate grouping. (a) One continuation per prefix leaves no alternatives. (b) X-GRPO scores all cross-role pairs and averages each candidate's reward over partners.

Stage 2: Multi-agent RL. Stage 2 teaches the agents to collaborate: on a mismatch, each agent analyzes the diagnostics and judges whether its own code needs correction. It uses X-GRPO trajectory sampling and a hierarchical reward combining individual correctness with cross-agent agreement.

X-GRPO trajectory sampling. A multi-turn rollout that continues each prefix with only one action leaves later prefixes without within-prefix alternatives for a group-relative per-turn advantage (Figure 2(a)). X-GRPO extends the turn-level grouping of AT-GRPO (Zhao et al., 2026) by sampling multiple candidates per role at every turn and evaluating each against multiple peers.

At turn t, agent $i \in \{V, P\}$ samples K candidate actions $a_t^{(i,c)}$, $c \in \{1, \ldots, K\}$, from its observation $o_t^{(i)}$. An action is a generated candidate response; the observation includes the specification, that agent's code history, and mismatch diagnostics. All K candidates for a role share its observation, and the two roles' observations differ because each agent sees only its own code.

X-GRPO evaluates the Cartesian product of the two candidate sets. With $R_{V,t}^{c,d}$ and $R_{P,t}^{c,d}$ the agent rewards for the pair $(a_t^{(V,c)}, a_t^{(P,d)})$, each candidate receives its mean reward over all K partners:

$$\overline{R}_{V,t}^{c} = \frac{1}{K}\sum_{d=1}^{K} R_{V,t}^{c,d}, \qquad \overline{R}_{P,t}^{d} = \frac{1}{K}\sum_{c=1}^{K} R_{P,t}^{c,d}. \tag{2}$$

The highest-scoring pair supplies the next-turn code histories and mismatch feedback, but all candidates receive credit through Equation 2, separating continuation choice from credit assignment. The Cartesian evaluation needs K^2 pair comparisons but only $2K$ model completions per turn and is used only in Stage 2 training; inference evaluates N index-wise pairs from $2N$ completions per turn.

For each role-specific group g, the candidate rewards are normalized within that group to obtain advantages $A_g^{(i,c)} = (\overline{R}_g^{(i,c)} - \mu_g^{(i)})/\sigma_g^{(i)}$, where $\mu_g^{(i)}$ and $\sigma_g^{(i)} \neq 0$ are the group mean and standard deviation; groups with identical candidate rewards carry no preference signal. With the importance ratio $r_g^{(i,c)} = \pi_{\theta^{(i)}}(a_g^{(i,c)} \mid o_g^{(i)})/\pi_{\theta_{\text{old}}^{(i)}}(a_g^{(i,c)} \mid o_g^{(i)})$ between the trainable and rollout policies for role i, clipping parameter ε, and a minibatch $\mathcal{B}_i$ containing only role i's groups, the clipped surrogate is

$$\mathcal{L}(\theta^{(i)}) = -\mathbb{E}_{g\in\mathcal{B}_i}\left[\frac{1}{K}\sum_{c=1}^{K}\min\left(r_g^{(i,c)}A_g^{(i,c)}, \text{clip}(r_g^{(i,c)}, 1-\varepsilon, 1+\varepsilon)A_g^{(i,c)}\right)\right]. \tag{3}$$

Each policy receives gradients from its own candidates. Cross-agent coupling enters through the pair-dependent rewards, selected continuations, and mismatch feedback.

Hierarchical reward design. Our reward teaches both agents to learn from mismatch information and improve their code through multi-turn iteration. For each evaluated pair, the per-agent reward combines three terms. **(1) Local reward**, $R_{\text{local},i}^{(t)}$, evaluates improvement in the agent's own code using oracle outputs available during training. The underlying gated score is 0 for compilation failure, 0.1 for a runtime error, 0.2 for an I/O-port mismatch, and $0.2 + 0.8c$ for a valid program with test pass fraction $c \in [0, 1]$; a program reaches the next tier only after passing the preceding checks. **(2) Correct-fix bonus**, $R_{\text{fix}}^{(t)}$, is binary: it rewards resolving a previous mismatch only when both agents now produce the correct output at the corresponding stimulus cycle, so agreement on an incorrect output earns nothing. **(3) Team-match reward**, $R_{\text{match}}^{(t)}$, rewards the fraction of matching outputs, a dense signal even without full correctness. The aggregate reward is

$$R_i^{(t)} = \delta_{\text{local}} R_{\text{local},i}^{(t)} + \delta_{\text{fix}} R_{\text{fix}}^{(t)} + \delta_{\text{match}} R_{\text{match}}^{(t)}, \tag{4}$$

with $\delta_{\text{local}} = 10$, $\delta_{\text{fix}} = 0.2$, and $\delta_{\text{match}} = 0.5$. These weights prioritize role-specific correctness while coupling the agents through correct repairs and behavioral agreement. The oracle-dependent terms supervise training only and are unavailable for inference-time selection.

2.3 Hybrid Reference-Model Data Generation

Existing public RTL datasets meet the need for Verilog agent training, while the Python agent requires cycle-accurate reference models. We build a hybrid generation framework combining three complementary pipelines (Figure 3), yielding approximately 64.4K reference-model samples. "Verified" means passing finite differential simulation against the source Verilog, not formal equivalence.

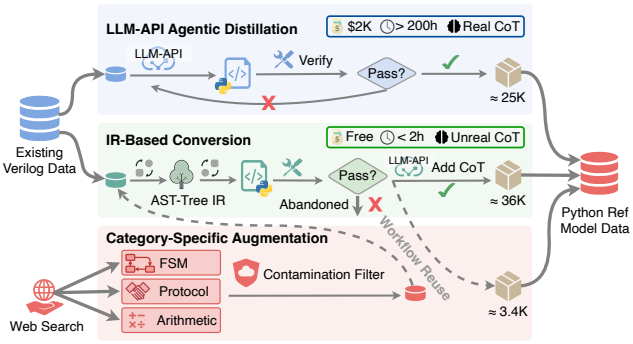

Figure 3: Reference-model data construction: LLM distillation, IR conversion, and category-specific augmentation.

LLM-API agentic distillation. We call a frontier LLM to generate a Python reference model for each Verilog module, then check it against the source Verilog with our cross-language comparator and feed mismatch diagnostics back for iterative self-correction. This pipeline produces samples with reasoning text, but has high cost and low yield: on the 87K-sample QiMeng-CodeV corpus (Zhu et al., 2025), DeepSeek-R1 (Guo et al., 2025) achieves 17% single-turn and 35% two-turn conversion success. Processing the full corpus with four API keys in parallel costs over $2,000 and 200 hours, yielding approximately 25K retained samples. This is an offline expense on public RTL; the deployed agents do not require an API.

IR-based conversion. To extend the corpus, a deterministic pipeline converts the remaining approximately 62K source modules at the abstract syntax tree (AST) level in three stages. **(1) Parsing.** We parse the source Verilog with PyVerilog (Takamaeda-Yamazaki, 2015) and normalize its ports, signals, assignments, and combinational and sequential blocks into a common intermediate representation (IR). **(2) Behavioral lowering.** Combinational logic becomes Python functions, while sequential logic uses a two-phase current/next-state update. Width masks and sign extension preserve the supported bit-vector operations. **(3) Top-module wrapping.** The translated functions are assembled into a `TopModule` class whose port interface mirrors the original module, so our cross-verification tool can test it directly. The supported subset is single-clock, post-reset, fully driven two-state behavior; the converter rejects delays, latches, multi-clock logic, user-defined primitives, and hierarchical references, and does not model general X/Z propagation. Every retained program is checked against the source Verilog by finite simulation. Conversion takes under two hours and yields about 36K more samples. An LLM then appends reasoning text to the translated programs. This post-hoc rationale does not record how the converter derived the code, and executable validation does not establish its faithfulness; in Figure 3, "Real/Unreal CoT" distinguishes jointly generated from post-hoc rationale text, and "Free" excludes rationale-generation and validation costs.

Category-specific augmentation. The first two pipelines yield about 61K samples but leave category gaps. A development evaluation of the 4B model on VerilogEval reveals frequent failures on finite-state machines, multi-cycle protocol blocks (e.g., counters, FIFOs), and bit-level arithmetic. We collect additional public Verilog examples in these categories, convert them through the IR pipeline, and filter for benchmark overlap, adding about 3.4K samples. Overlap screening is imperfect and the categories depend on the development set; Section 3.7 reports a full-corpus contamination audit.

Table 1: Pass@k (%) for RTL generation; **1st** / 2nd / 3rd per column, our rows shaded, sizes per agent. Agents rows are full workflows; pass@5 is oracle-assisted coverage. ‡: reconciliation pending.

Type	Model	Size	VerilogEval v2		RTLLM v2		ChipBench-SC		CVDP cid003		CVDP cid004	
			P@1	P@5	P@1	P@5	P@1	P@5	P@1	P@5	P@1	P@5
Foundation Models	GPT-4o	–	64.1	73.7	56.5‡	70.3	20.0	33.3	49.2	52.4	43.1	45.6
	GPT-5.5	–	84.7	**90.4**	63.2	68.0	30.7	36.7	**53.9**‡	**59.0**	50.0	55.1
	Claude Opus 4.7	–	**86.9**	**90.4**	64.8	68.0	31.3	**46.7**	52.2	57.1	**51.2**	**55.9**
	DeepSeek Coder	236B	68.5	80.8	57.6	70.0	16.7	30.0	34.3‡	47.7	—	—
	DeepSeek V4	1.6T	67.3	80.1	58.8	66.0	18.0	36.7	32.8	45.1	33.8	46.4
	DeepSeek R1	671B	77.5	84.7	64.7‡	75.8	26.7	40.0	39.1	52.3	37.5	51.0
Specialized Models	CodeV-R1 (distill)	7B	65.2	75.2	57.2	71.9	13.3	26.7	37.1	52.3	—	—
	CodeV-R1	7B	68.8	78.2	68.0	**78.2**	30.0	40.0	38.8	54.3	37.1	49.1
Base Models	Qwen3.5-4B	4B	41.7	60.9	34.3‡	49.7	6.7	10.0	22.6	25.0	—	—
	Qwen3.5-9B	9B	48.5	66.6	36.1‡	57.8	13.3	20.0	24.9	32.1	—	—
ChipMATE (Ours)	ChipMATE-Verilog-4B	4B	67.4	71.8	68.0	74.6	26.7	33.3	34.5	49.9	—	—
	ChipMATE-Agents-4B	4B	75.0	76.3	74.6	77.3	33.3	43.3	43.0‡	52.1	—	—
	ChipMATE-Verilog-9B	9B	75.3	77.6	71.9‡	75.8	30.0	36.7	39.3‡	53.7	39.1‡	51.0
	ChipMATE-Agents-9B	9B	80.1	82.4	**75.8**	77.3	**36.7**	43.3	50.1	55.9	46.9	52.3

3 Experiments

3.1 Implementation Details

We apply the two-stage pipeline (Section 2.2) to the Qwen3.5 Qwen Team (2026) base model with 4B and 9B parameters *per agent*, so a pair contains 8B or 18B parameters. SFT uses LLaMA-Factory Zheng et al. (2024) on the 87K-sample QiMeng CodeV-R1 dataset Zhu et al. (2025) for the Verilog agent and on our 64.4K reference-model samples for the Python agent. Single-agent RL uses verl Sheng et al. (2025) with GRPO Shao et al. (2024), and multi-agent RL builds on a modified PettingLLMs Zhao et al. (2026); Appendix B lists hyperparameters and hardware.

Benchmarks and protocol. We evaluate RTL generation on VerilogEval v2 (156 tasks) Pinckney et al. (2025a), RTLLM v2 (50) Liu et al. (2024), ChipBench-SC (30 self-contained tasks) Yu et al. (2026), CVDP cid003 (78), and CVDP cid004 RTL modification (52 evaluable tasks) Pinckney et al. (2025b); Python reference-model generation is evaluated on the first four. We use a 16,384-token limit, temperature 0.6 for SFT-only checkpoints and 1.0 for RL and agentic checkpoints, and ten independent outputs per problem. For agentic systems, each output is the final RTL of one complete workflow with $N = 3$ candidates *per agent per turn* and at most $T = 3$ turns; internal selection uses only inter-agent simulation, never the held-out benchmark tests that score the returned RTL. We report $\text{pass@}k = 1 - \binom{n-c}{k} / \binom{n}{k}$ with $n = 10$ and $k \in \{1, 5\}$ (Chen et al., 2021), where c counts correct returned outputs. Pass@1 is the expected correctness of one returned output, not one model call; pass@5 is oracle-assisted coverage, since deploying it requires picking a correct output.

3.2 End-to-End Results

Table 1 reports final RTL accuracy across the five task sets. The 9B-per-agent pair improves pass@1 over CodeV-R1 by 11.3, 7.8, 6.7, 11.3, and 9.8 percentage points (pp) on VerilogEval, RTLLM, ChipBench-SC, CVDP cid003, and CVDP cid004, respectively. These are system-level differences: the pair uses iterative selection and repair, whereas CodeV-R1 generates one candidate per output, so they do not establish a model-only or compute-matched advantage. Frontier API models remain stronger on some tasks, and higher pass@1 need not imply better pass@5: on RTLLM, CodeV-R1 reaches 78.2 pass@5 versus 77.3 for the trained pair.

Backend and role controls. To separate backend strength from the workflow, we also run each external model in both roles of the identical framework, with identical prompts, N, T, stimuli, backtracking, and stopping (Appendix C.1). On VerilogEval, the framework improves over self-consistency (SC@9) by 0.8–2.1 pp for all four backends; DeepSeek R1 reaches 81.4 in it versus 80.1 for our offline 9B pair. In a 2×2 role swap from CodeV-R1 in both roles (74.1), replacing only the

Table 2: Pass@k (%) for Python reference-model generation; highlighting and ‡ as in Table 1.

Type	Model	Size	VerilogEval v2		RTLLM v2		ChipBench-SC		CVDP cid003	
			P@1	P@5	P@1	P@5	P@1	P@5	P@1	P@5
Foundation Models	GPT-5.5	–	61.5	75.5	48.5‡	55.4	33.0	42.1	51.3	57.7
	DeepSeek Coder	236B	60.1	73.1	42.4	50.0	28.7	40.0	32.4	45.0
	DeepSeek V4	1.6T	59.7	71.8	44.4	54.3	30.7	40.0	35.2‡	46.1
	DeepSeek R1	671B	57.1	70.7	49.6	57.8	28.7	36.7	36.7	47.6
Specialized Models	CodeV-R1 (distill)	7B	40.4	48.5	36.0	46.2	23.3	30.0	34.1	46.1
	CodeV-R1	7B	45.3	56.4	42.7‡	51.8	28.7	33.3	32.4	44.2
Base Models	Qwen3.5-4B	4B	46.8	63.7	38.2	46.1	16.7	23.3	20.2‡	22.8
	Qwen3.5-9B	9B	48.0	64.7	40.1‡	51.8	20.0	26.7	22.4	29.0
ChipMATE-P (Ours)	ChipMATE-Python-4B	4B	77.6	80.1	75.3‡	80.1	41.3	46.7	43.2	55.0
	ChipMATE-Python-9B	9B	**82.4**	**84.7**	**77.3**‡	**81.0**	**43.3**	**50.0**	**53.6**	**58.8**

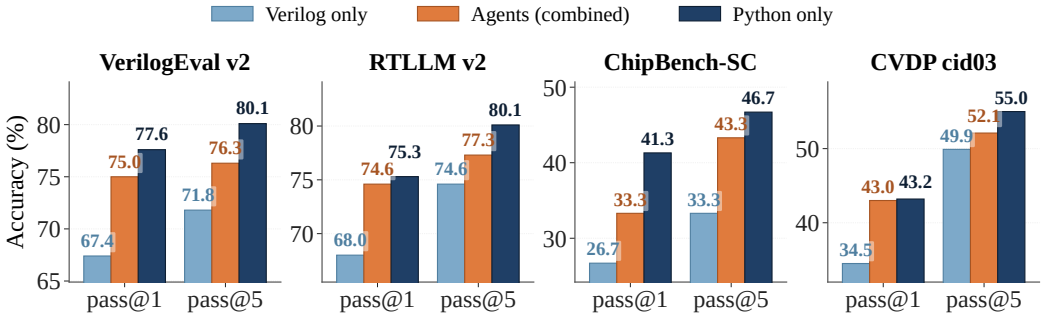

Figure 4: Single-agent Python and Verilog accuracy and the final RTL of the 4B-per-agent workflow.

reference agent yields 77.0, only the Verilog agent 77.8, and both 80.1, so both roles contribute. The trained Verilog agent alone reaches 77.4 under SC@9, so the pair's further 2.7 pp reflects the Python policy, the correction workflow, and joint training.

3.3 Python Reference Model Generation

Table 2 reports Python reference-model generation. The 9B Python agent exceeds the 9B Verilog agent by 7.1, 5.4, 13.3, and 14.3 pp across the four benchmarks. This gap makes reference-guided RTL repair plausible, although Python scores evaluate a different task and do not bound RTL accuracy.

3.4 Python vs. Verilog

Figure 4 compares the single-agent tracks with the final RTL of the workflow; relative to Verilog-only generation, the workflow gains 6.6–8.5 pp at 4B and 3.9–10.8 pp at 9B (Table 1).

A paired 4B VerilogEval analysis finds 16 wrong-to-right and four right-to-wrong transitions relative to Verilog-only generation (net $12/156$; exact McNemar $p = 0.012$). The reference was already correct in 11 repairs and five recovered from a both-wrong start, so the dominant observed effect is reference-guided transfer, with a smaller joint-recovery component. A correct reference does not guarantee a correct implementation: ten of the 39 final failures retain one: five exhaust the turn budget, three evade the stimuli, one gains agreement without a fix, and one never compiles (Appendix C.3).

3.5 Ablation Study

Figure 5 presents the training and inference stages. SFT yields the largest observed gain: 19.2 pp at 4B and 22.0 pp at 9B relative to the corresponding base model. A multi-agent workflow without backtracking falls below the single-agent checkpoint; with the same checkpoints, backtracking raises

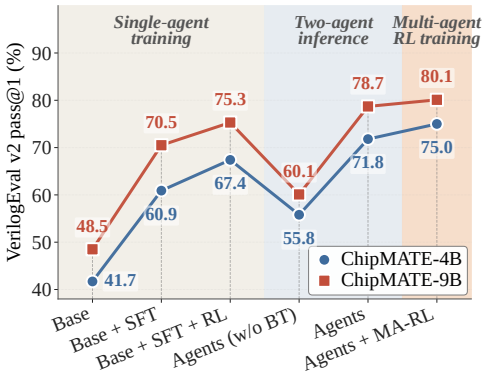

Figure 5: VerilogEval v2 training and inference stages (BT: backtracking). The 9B single-agent point quotes Table 1.

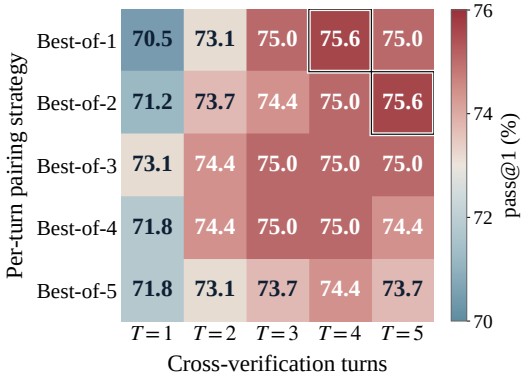

Figure 6: 4B VerilogEval workflow settings. $N{=}3, T{=}3$ quotes the ten-run 75.0; other cells are single runs.

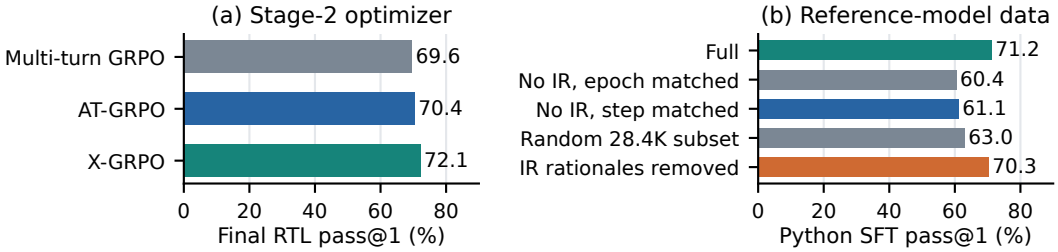

Figure 7: Training controls: (a) Stage-2 optimizers; (b) 4B Python SFT data.

pass@1 from 55.8 to 71.8 at 4B and from 60.1 to 78.7 at 9B, and the jointly trained workflows reach 75.0 and 80.1. The 9B single-agent point reuses the main evaluation, so comparisons to it are descriptive, not paired. This favors controlling harmful revisions before optimizing cooperation.

Training controls. Under identical reward and data, X-GRPO reaches 72.1 on all 156 VerilogEval tasks, versus 70.4 for AT-GRPO and 69.6 for standard multi-turn GRPO (Figure 7(a)), and exceeds AT-GRPO in each of three Stage-2 seeds (mean gain 1.8 pp, standard deviation 0.35 pp); these runs do not equalize pair-simulation budgets, since X-GRPO scores K^2 pairs per turn. Training with R_{local} alone lowers a step-80 checkpoint from 71.9 to 69.7, isolating the added coupling through R_{match} and R_{fix}. For the Python agent (Figure 7(b)), removing the IR-derived subset lowers SFT pass@1 from 71.2 to 61.1 at matched steps, below 63.0 for a random subset of equal size, whereas removing only its post-hoc rationales gives 70.3. The IR-derived code thus adds value beyond extra updates, while dropping its rationales shows no observed benefit; these data controls use a single seed. Appendix C.2 gives their budgets, seeds, and scope.

3.6 Exploration of the Agentic Workflow

Figure 6 sweeps the per-turn samples $N \in \{1, \ldots, 5\}$ and turn limit $T \in \{1, \ldots, 5\}$ for the ChipMATE-4B checkpoint. Accuracy saturates near 75: $N = 1, T = 4$ and $N = 2, T = 5$ each reach 75.6, the former at a lower maximum budget, so we do not claim that the default $N = 3, T = 3$ is cost-optimal; larger 9B and RTLLM settings add 0.3 pp at one third more budget (Appendix C.6). Early termination reduces the measured 9B cost to 9.8 sampled completions, 4.9 simulations, 30.5K generated tokens, and 2.9 minutes per problem. DeepSeek R1 SC@9 generates 95K tokens per problem (117.9K per solved problem versus 38.1K for our pair), although token counts do not establish FLOP equivalence.

3.7 Reliability, Uncertainty, and Scope

Agreement and simulation. Agreement is useful for selection but can preserve shared mistakes. Scored with held-out tests after generation, agreement precision falls from 88.2% on VerilogEval to 62.0% on ChipBench-SC, while false consensus (agreed but incorrect) rises from 10.3% to 20.7% of all runs; two annotators attributed 71% of false consensuses to shared specification misreadings. An optional mode that rechecks agreement on fresh stimuli and abstains below a calibrated threshold raises precision to 95.8%, 94.0%, and 78.6% but auto-accepts only 73%, 67%, and 37% of runs, leaving 3.1–8.0% false consensus (Appendix C.3). The stimuli kill 94.1% of 376 injected mutants, and strengthening the official FSM tests changes each model's pass@1 by at most 0.5 pp (Appendix C.4).

Uncertainty and training-data overlap. Under 95% problem-bootstrap intervals, the 9B pair's gain over CodeV-R1 is significant on VerilogEval ($p = 0.003$) but not individually on RTLLM or ChipBench-SC. A full-corpus overlap audit flags 33 of 156 VerilogEval tasks; on the other 123, the pair still leads CodeV-R1 by 11.7 pp (79.3 versus 67.6; Appendix C.5).

RTL modification and synthesis. On CVDP cid004, the 9B pair reaches 46.9 pass@1 versus 39.1 for its Verilog-only agent and 37.1 for CodeV-R1 (Appendix C.7). On 26 common RTLLM designs synthesized with yosys/Nangate45, its area/depth ratios to golden designs are $1.11/1.05$, versus $1.17/1.12$ for CodeV-R1 and $1.09/1.04$ for DeepSeek R1 (Appendix C.8).

4 Related Work

Agentic RTL generation. AutoChip (Thakur et al., 2023), MAGE (Zhao et al., 2025), and VerilogCoder (Ho et al., 2025) repair HDL with simulator feedback, generated testbenches, or waveform analysis, and AutoVeriFix and AutoVeriFix+ (Tan et al., 2026a;b) already check Verilog against Python reference models. ChipMATE therefore does not claim Python-assisted verification or multi-agent orchestration as new; it trains two separately parameterized, code-blind roles with partner-averaged credit and a dedicated reference-model corpus. AutoVeriFix reports higher RTLLM-v2 pass@5 than ours (83.5% versus 77.3%) under a different backend and selector, and ACE-RTL (Deng et al., 2026) uses a different CVDP cid004 protocol, so we avoid cross-paper rankings.

Specialized RTL models. RTLCoder, AutoVCoder, BetterV, OriGen, and CraftRTL improve RTL generation through domain-specific training, augmentation, or guidance (Liu et al., 2025b; Gao et al., 2024; Pei et al., 2024; Cui et al., 2024; Liu et al., 2025a), and VeriReason, RTLSeek, ChipSeek, and QiMeng-CodeV-R1 apply reinforcement learning (Wang et al., 2026; Zhang et al., 2026; Chen et al., 2026b; Zhu et al., 2025). SiliconMind-V1 also builds a local multi-agent workflow (Chen et al., 2026a); we instead train a separate reference-model role jointly with the Verilog role.

Execution feedback and co-training. CodeT and LEVER select code by execution (Chen et al., 2023; Ni et al., 2023), and Self-Refine, Reflexion, and self-debugging revise outputs from feedback (Madaan et al., 2023; Shinn et al., 2023; Chen et al., 2024), yet agreement on sampled inputs need not imply correctness. CURE and RL Tango co-train generators with testers or verifiers (Wang et al., 2025; Zha et al., 2025), while AT-GRPO and MAGRPO train cooperative LLM agents (Zhao et al., 2026; Liu et al., 2026). X-GRPO differs in credit assignment: it scores all cross-role pairs at a turn and averages each candidate's pair rewards before within-role normalization.

5 Conclusion

We presented ChipMATE, a self-trained multi-agent framework for offline RTL generation in which a Verilog agent and a Python reference-model agent cross-verify their outputs without any golden testbench or cloud-LLM API at inference, while training uses correctness oracles. Its joint backtracking workflow keeps observed agreement nondecreasing across turns, its two-stage pipeline trains both individual capability and collaboration, and its hybrid data-generation framework supplies reference-model training data checked by finite simulation. These results show the potential of self-trained multi-agent workflows for Verilog generation under local deployment and model adaptation.

AI USE STATEMENT

Generative language models were used in constructing the training corpus: they generated reference-model candidates for the distillation pipeline and appended synthetic rationales to IR-derived programs, as described in Section 2.3. Retained programs were checked by finite differential simulation; these checks do not validate the faithfulness of their rationales. AI assistance was also used for manuscript revision, literature and citation checking, and plotting recorded experimental results. The authors are responsible for the final text, references, claims, and artifacts.

REPRODUCIBILITY STATEMENT

Section 2 describes the inference workflow, candidate-credit calculation, and training rewards. Section 3.1 and Appendix B report the training setup and evaluation protocol, and the appendix provides additional results and the agent prompt templates. We plan to release the model checkpoints, reference-model corpus, and training and inference implementation. The reported uncertainty distinguishes repeated inference from Stage-2 training seeds, and the supplementary analyses separate historical benchmark results from later protocol checks.

REFERENCES

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

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. Evaluating Large Language Models Trained on Code, 2021. URL https://arxiv.org/abs/2107.03374. arXiv preprint arXiv:2107.03374.

Mu-Chi Chen, Yu-Hung Kao, Po-Hsuan Huang, Shao-Chun Ho, Hsiang-Yu Tsou, I-Ting Wu, En-Ming Huang, Yu-Kai Hung, Wei-Po Hsin, Cheng Liang, Chia-Heng Tu, Shih-Hao Hung, and H. T. Kung. SiliconMind-V1: Multi-Agent Distillation and Debug-Reasoning Workflows for Verilog Code Generation. In *2026 IEEE 50th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 88–97, 2026a. doi: 10.1109/COMPSAC69091.2026.00023. URL https://doi.org/10.1109/COMPSAC69091.2026.00023.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Cangyuan Li, Xinyang He, Chujie Chen, Mengdi Wang, Haobo Xu, Yinhe Han, Huawei Li, and Ying Wang. ChipSeek: Optimizing Verilog Generation via EDA-Integrated Reinforcement Learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25180–25201. Association for Computational Linguistics, 2026b. doi: 10.18653/v1/2026.acl-long.1154. URL https://aclanthology.org/2026.acl-long.1154/.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024. doi: 10.1145/3676536.3676830. URL https://doi.org/10.1145/3676536.3676830.

Chenhui Deng, Zhongzhi Yu, Guan-Ting Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. ACE-RTL: When Agentic Context Evolution Meets RTL-Specialized LLMs, 2026. URL https://arxiv.org/abs/2602.10218. arXiv preprint arXiv:2602.10218.

Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pp. 162–169, 2024. doi: 10.1109/ICCD63220.2024.00033. URL https://doi.org/10.1109/ICCD63220.2024.00033.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 300–307, 2025. doi: 10.1609/aaai.v39i1.32007. URL https://ojs.aaai.org/index.php/AAAI/article/view/32007.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair. In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e112a4671e8779aa9f640a0e3f81bd26-Abstract-Conference.html.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024. doi: 10.1145/3676536.3697118. URL https://doi.org/10.1145/3676536.3697118.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 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,

2025b. doi: 10.1109/TCAD.2024.3483089. URL https://doi.org/10.1109/TCAD.2024.3483089.

Shuo Liu, Zeyu Liang, Xueguang Lyu, and Christopher Amato. LLM Collaboration with Multi-Agent Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 32150–32158, 2026. doi: 10.1609/aaai.v40i38.40487. URL https://ojs.aaai.org/index.php/AAAI/article/view/40487.

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. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 26106–26128. PMLR, 2023. URL https://proceedings.mlr.press/v202/ni23b.html.

Zehua Pei, Huiling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. BetterV: Controlled Verilog Generation with Discriminative Guidance. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 40145–40153. PMLR, 2024. URL https://proceedings.mlr.press/v235/pei24e.html.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–20, 2025a. doi: 10.1145/3718088. URL https://doi.org/10.1145/3718088.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification, 2025b. URL https://arxiv.org/abs/2506.14074. arXiv preprint arXiv:2506.14074.

Qwen Team. Qwen3.5: Towards Native Multimodal Agents, 2026. URL https://qwen.ai/blog?id=qwen3.5. Official model release.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models, 2024. URL https://arxiv.org/abs/2402.03300. arXiv preprint arXiv:2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Shinya Takamaeda-Yamazaki. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pp. 451–460. Springer International Publishing, 2015. doi: 10.1007/978-3-319-16214-0_42. URL https://doi.org/10.1007/978-3-319-16214-0_42.

Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. AutoVeriFix: Automatically Correcting Errors and Enhancing Functional Correctness in LLM-Generated Verilog Code. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 526–532, 2026a. doi: 10.1109/ASP-DAC66049.2026.11420300. URL https://doi.org/10.1109/ASP-DAC66049.2026.11420300.

Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. AutoVeriFix+: High-Correctness RTL Generation via Trace-Aware Causal Fix and Semantic Redundancy Pruning, 2026b. URL https://arxiv.org/abs/2603.11489. arXiv preprint arXiv:2603.11489.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. AutoChip: Automating HDL Generation Using LLM Feedback, 2023. URL https://arxiv.org/abs/2311.04887. arXiv preprint arXiv:2311.04887.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-Evolving LLM Coder and Unit Tester via Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://arxiv.org/abs/2506.03136.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. VeriReason: Reinforcement Learning with Testbench Feedback for Reasoning-Enhanced Verilog Generation. In *Proceedings of the Great Lakes Symposium on VLSI 2026*, pp. 948–953, 2026. doi: 10.1145/3787109.3815310. URL https://doi.org/10.1145/3787109.3815310.

Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. ChipBench: A Next-Step Benchmark for Evaluating LLM Performance in AI-Aided Chip Design, 2026. URL https://arxiv.org/abs/2601.21448. arXiv preprint arXiv:2601.21448.

Kaiwen Zha, Zhengqi Gao, Maohao Shen, Zhang-Wei Hong, Duane S. Boning, and Dina Katabi. RL Tango: Reinforcing Generator and Verifier Together for Language Reasoning. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://arxiv.org/abs/2505.15034.

Xinyu Zhang, Zhiteng Chao, Yonghao Wang, Bin Sun, Tianyun Ma, Tianmeng Yang, Jianan Mu, Jing Justin Ye, and Huawei Li. RTLSeek: Boosting the LLM-Based RTL Generation with Multi-Stage Diversity-Oriented Reinforcement Learning, 2026. URL https://arxiv.org/abs/2603.27630. arXiv preprint arXiv:2603.27630.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7, 2025. doi: 10.1109/DAC63849.2025.11133191. URL https://doi.org/10.1109/DAC63849.2025.11133191.

Yujie Zhao, Lanxiang Hu, Yang Wang, Minmin Hou, Hao Zhang, Ke Ding, and Jishen Zhao. Stronger-MAS: Multi-Agent Reinforcement Learning for Collaborative LLMs. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/f3da4165893c2465fd7e8df453c41ffa-Abstract-Conference.html.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. QiMeng-CodeV-R1: Reasoning-Enhanced Verilog Generation. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-5160. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/e2f7407b62f152b7fe533fbc077fddb7-Paper-Conference.pdf.

A LIMITATIONS

Agreement and coverage. The two agents can share specification errors despite not seeing each other's generated code. Backtracking preserves the best observed agreement, and finite simulation can miss behavior that distinguishes a correct design from an incorrect one. Agreement precision is particularly limited on ChipBench-SC. Fresh-suite filtering reduces false acceptance at the cost of abstention, but is a separate operating mode and does not certify correctness. Deployment still requires established verification procedures; this system does not replace formal verification, coverage closure, or design review.

Training data and generalization. IR-derived programs are checked by finite differential simulation, and their appended rationales are synthetic explanations rather than faithful records of code derivation. The converter covers a restricted single-clock, two-state subset. The overlap audit supports a clean-subset comparison on VerilogEval but cannot rule out undetected similarity or pretraining exposure. Targeted augmentation was motivated by benchmark-observed failure categories, so that comparison does not establish benchmark-independent development. Our public-benchmark evaluation uses 4B/9B backbones and a limited workflow-parameter search; it does not establish effectiveness on proprietary or production-scale designs, other model families, or tasks outside RTL.

Cost and evidence scope. The workflow performs repeated generation and simulation, so final-output pass@1 is not a compute-matched comparison with single-completion pass@1. Common-framework and self-consistency controls improve comparability, but hosted-model FLOPs and directly comparable latency are unavailable. Confidence intervals remain broad on small benchmarks. Some reported aggregates still require count/protocol reconciliation; corrected-CVDP uncertainty and reliability diagnostics are unavailable. The three training seeds vary Stage 2 from one fixed Stage 1 checkpoint, rather than repeating the full pipeline. Synthesis results cover a common subset of passing designs and report area and logic depth, without signoff timing or power.

B TRAINING AND EVALUATION DETAILS

Training. We obtain ChipMATE by applying the two-stage pipeline (Section 2.2) to the Qwen3.5 Qwen Team (2026) base model with 4B and 9B parameters *per agent*; since the agents have separate parameters, a pair contains 8B or 18B parameters. Stage 1 comprises single-agent SFT followed by single-agent RL, and Stage 2 performs multi-agent RL. For SFT, we use LLaMA-Factory Zheng et al. (2024). The Verilog agent is trained on the 87K-sample dataset released with QiMeng CodeV-R1 Zhu et al. (2025), and the Python agent on the 64.4K reference-model samples generated by our hybrid framework (Section 2.3). Both agents are trained for 6 epochs with a learning rate of 1×10^{-5}, a global batch size of 64, and a context length of 16,384 tokens. For single-agent RL, we use verl Sheng et al. (2025) with GRPO Shao et al. (2024). We set the global batch size and minibatch size to 128, the GRPO group size to 16, and the learning rate to 1×10^{-6}. Training runs for 300 steps with a rollout temperature of 1.0, an instruction length of 2,048, and a response length of 16,384. The reward function is the same local reward used in multi-agent RL. For multi-agent RL, we build on a modified version of PettingLLMs Zhao et al. (2026) with verl as the backbone. Both agents are updated for an additional 200 steps at the same learning rate. The 9B agents are trained on two nodes of 8×H200 (141 GB each), and the 4B agents on a single 8×H100 node.

Evaluation. Separate Python-generation results are not available for CVDP cid004. The default workflow samples $N = 3$ candidates per agent per turn, pairs them index-wise, and runs for at most $T = 3$ turns with a fixed suite of 1,000 stimuli within each run, so one returned RTL may use up to 18 model completions and nine pair simulations. Stage 2 X-GRPO instead evaluates all $K \times K$ candidate pairs per turn ($2K$ model completions, K^2 pair comparisons), whereas inference evaluates N index-wise pairs ($2N$ completions, N comparisons). Pass@5 measures oracle-assisted candidate coverage: using it as a deployment success rate would require identifying a correct result among five outputs. We therefore separate single-pass generation, deployable selection, and the same multi-agent workflow in the controls below.

C Additional Experimental Results

This appendix reports the aggregate results underlying the controls summarized in Section 3. Unless a table states otherwise, accuracy is in percent and differences are in percentage points. Different checkpoints, output tasks, or extraction protocols are kept separate. To use one reported estimate for repeated configurations, entries marked † reuse the main-evaluation value; they are references, not additional control measurements.

C.1 Backend, Role, and Inference-Cost Controls

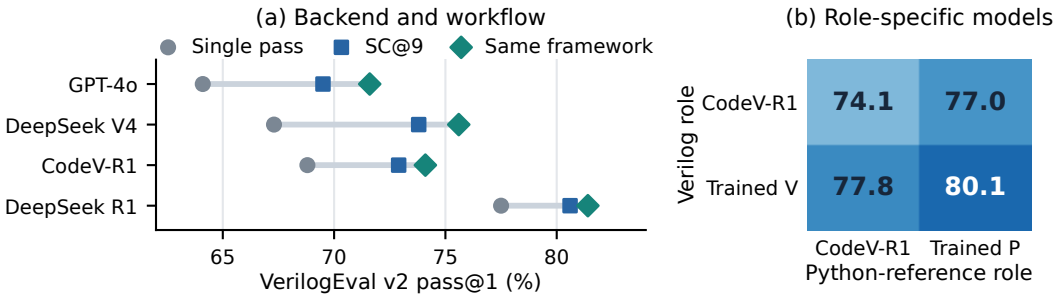

Figure 8: VerilogEval v2 controls. (a) Single pass, SC@9, and the same framework with each backend in both roles. (b) Role swap within that framework (final RTL pass@1, %).

Table 3: VerilogEval v2 selector and backend controls (%), all with $N = 3, T = 3$.

Backend	Single-pass	SC@9	Same framework
GPT-4o	64.1	69.5	71.6
DeepSeek V4	67.3	73.8	75.6
CodeV-R1	68.8	72.9	74.1
DeepSeek R1	77.5	80.6	81.4
Trained 9B pair	—	77.4†	80.1
Claude Opus 4.7	—	88.7	—
GPT-5.5	—	87.5	—

†SC@9 uses the trained Verilog policy alone. CodeV-R1's oracle pass@9 is 80.9; it requires a correctness selector.

For the external-backend controls in Figure 8(a) and Table 3, one model independently samples both roles using the same frozen prompts, index-wise pairing, $N = 3$, $T = 3$, 1,000 stimuli, joint backtracking, and stopping rule. SC@9 samples nine RTL candidates, clusters them by simulated behavior, and returns a representative of the largest cluster; it uses no golden correctness selector. The same-framework controls preserve the system topology and candidate budget, but backend-dependent output length, model cost, and early stopping prevent interpreting them as equal-compute comparisons. The framework improves over SC@9 for all four backends, by 0.8–2.1 pp. DeepSeek R1 reaches 81.4 in the framework, compared with 80.1 for the trained offline 9B pair, and the trained pair exceeds the GPT-4o, V4, and CodeV-R1 implementations of the same workflow. Claude Opus 4.7 and GPT-5.5 reach 88.7 and 87.5 under SC@9. These controls support the utility of the workflow across the tested backends while retaining a role for backend capability. CodeV-R1 was optimized for RTL rather than Python reference modeling; its use in both roles is a topology control, not an independently optimized verifier baseline.

The role swap in Figure 8(b) holds the inference workflow fixed. Replacing the Python policy adds 2.9 pp when paired with CodeV-R1's Verilog policy; replacing the Verilog policy adds 3.7 pp when CodeV-R1 supplies the reference policy. Using both trained agents adds 6.0 pp over CodeV-R1 in both roles. This comparison attributes value to both role-specific checkpoints without isolating joint RL from their earlier role-specific training. The trained Verilog agent alone reaches 77.4 under SC@9; the pair's additional 2.7 pp includes the Python policy, the correction workflow, and joint

training. For reference, CodeV-R1's oracle pass@9 is 80.9, which requires a correctness selector that is unavailable at deployment.

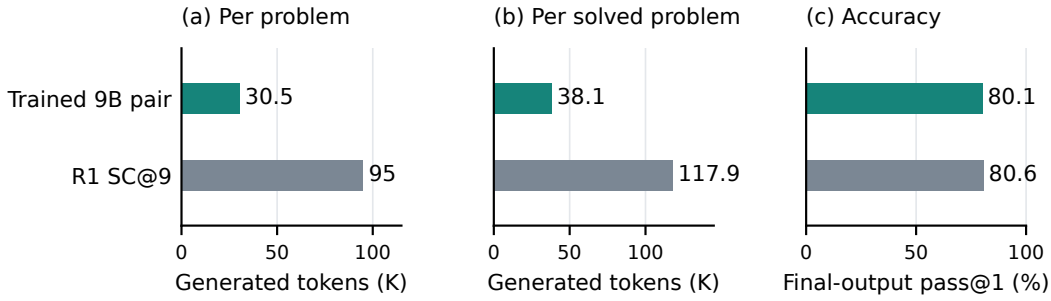

Figure 9: VerilogEval inference cost and accuracy of the trained 9B pair and DeepSeek R1 SC@9.

The trained 9B pair averages 9.8 model completions and 4.9 pair simulations per problem, versus nine model completions for R1 SC@9 (Figure 9). The per-solved token figures divide mean generated tokens per problem by final-output pass@1: $30.5\text{K}/0.801 \approx 38.1\text{K}$ and $95\text{K}/0.806 \approx 117.9\text{K}$. They measure generated-token cost, not input processing, simulator cost, or model FLOPs. The 2.9-minute value is a measured local-system average; R1 pair-simulation counts, directly comparable API latency, and hosted-model FLOP accounting are unavailable.

C.2 Training Controls and the Reference-Model Corpus

Table 4: Optimizer controls (all 156 VerilogEval tasks, same reward and data) and step-80 reward controls; distinct training runs.

Optimizer control	Final RTL pass@1 (%)
Standard multi-turn GRPO	69.6
AT-GRPO	70.4
X-GRPO	72.1
Reward control	**Step-80 pass@1 (%)**
Full reward	71.9
R_{local} only	69.7

Table 5: 4B Python-agent SFT data controls (single seed, same base model).

SFT data	Samples	Updates	Pass@1 (%)
Full corpus	64.4K	4.0K	71.2
No IR, epoch matched	28.4K	1.8K	60.4
No IR, step matched	28.4K	4.0K	61.1
Random size-matched full subset	28.4K	4.0K	63.0
Full code, IR rationales removed	64.4K	4.0K	70.3

The stage summary is shown in Figure 5. The 9B single-agent RL entry quotes the main evaluation's 75.3 for consistent reporting; it is not treated as a new matched ablation measurement. The with/without-backtracking comparison retains the original same-checkpoint control.

The X-GRPO comparison in Figure 7(a) uses identical reward and data across the three optimizers and evaluates all 156 VerilogEval tasks. It does not establish equal training simulation cost: X-GRPO evaluates K^2 candidate pairs per turn. The separate three-seed analysis reruns only Stage 2 from one fixed Stage-1 checkpoint; its mean improvement over AT-GRPO is 1.8 pp (standard deviation 0.35 pp), with a positive difference in every seed. These are Stage-2 repeats, not independent end-to-end retrainings, and they support the tested turn-structured setting rather than all multi-agent RL tasks. The optimizer-control scores are reported separately from the headline results; the step-80

reward control evaluates an intermediate checkpoint. For the reward control, R_{local}-only training retains trajectory-level pair selection and mismatch diagnostics. The 2.2 pp gap therefore measures the additional effect of the explicit cross-agent reward terms R_{match} and R_{fix}, not the entire effect of interaction.

The SFT-only data control in Figure 7(b) uses 4B Python policies initialized from the same base model. The full and step-matched arms use 4.0K updates, whereas the epoch-matched no-IR arm uses 1.8K. Removing the 36K IR-conversion subset lowers accuracy by 10.1 pp at matched steps, and a random equally sized subset of the full corpus reaches 63.0, 1.9 pp above the no-IR subset. Preserving the code but dropping its post-hoc rationale lowers accuracy by 0.9 pp. These controls support retaining the IR-derived code and show no observed improvement from dropping its rationale text. The removed 36K subset is the main IR-conversion stream; the retained augmentation also uses IR conversion. The controls do not establish faithful reasoning: they are single-seed comparisons, and removing rationale tokens also changes token exposure. An earlier distillation-only development checkpoint reached 64.1 versus 52.9 for its base model and 71.2 for full SFT, but used a different schedule; it is context rather than a controlled causal comparison. The distillation procedure produces approximately 25K verified samples, with 17% one-turn yield and 35% yield after repair, at a reported cost exceeding USD 2,000 and 200 hours.

IR-derived Python code is retained only after finite differential simulation against its source Verilog. No separate structural-coverage statistic was computed for these construction-time checks. The added rationale is synthetic, and successful code validation does not make it a faithful trace of how the program was derived. The converter is not formally verified; the scope and excluded language features are described in Section 2.3.

C.3 Agreement Diagnostics, Accuracy Transfer, and Residual Failures

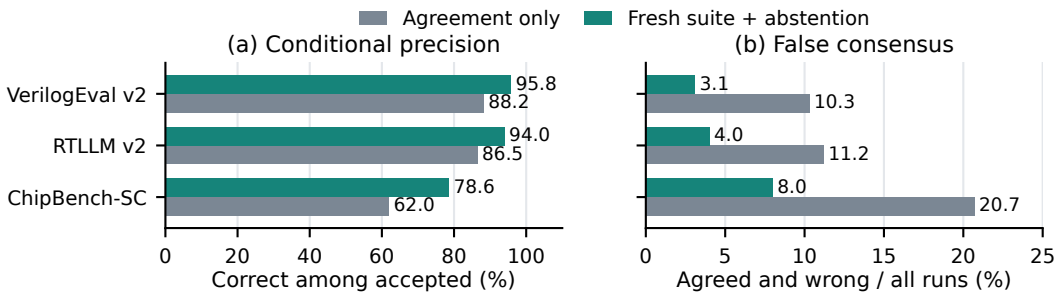

Figure 10: Agreement reliability without and with fresh-suite abstention.

Table 6: Agreement diagnostics (%). Precision conditions on acceptance; the other columns use all runs.

	Agreement only		Fresh suite + abstention		
Benchmark	Precision	False consensus	Precision	Coverage	Residual false consensus
VerilogEval v2	88.2	10.3	95.8	73	3.1
RTLLM v2	86.5	11.2	94.0	67	4.0
ChipBench-SC	62.0	20.7	78.6	37	8.0

We quantify agreement precision, $P(\text{correct} \mid \text{terminal agreement})$, and false consensus, $P(\text{terminal agreement} \wedge \text{incorrect})$, using held-out scoring only after generation. Agreement precision in Figure 10 and Table 6 conditions on the internal termination event; it is distinct from the benchmark-scored correctness of the final returned RTL across all runs. False consensus uses all runs as denominator. Precision falls from 88.2% on VerilogEval to 62.0% on ChipBench-SC, while false consensus rises from 10.3% to 20.7% of all runs. An earlier manual analysis by two annotators classified 71% of false-consensus cases as shared specification misreadings; this taxonomy was not

recomputed after the CVDP score corrections. The policies therefore have correlated errors despite separate parameters and code-blind generation.

The optional filter rechecks agreement on a fresh, disjoint stimulus suite and applies one validation-calibrated abstention threshold, sending the remaining cases to review. It is evaluated separately from the default workflow and does not change the headline pass@1 results. Filtered precision rises to 95.8%, 94.0%, and 78.6% at auto-accept coverage of 73%, 67%, and 37% on VerilogEval, RTLLM, and ChipBench-SC, respectively, and residual false consensus remains 3.1–8.0% of all runs. This tradeoff can support triage, but does not make agreement a correctness certificate or a replacement for downstream verification.

Table 1 compares the combined workflow directly with its Verilog-only policy. In the paired 4B VerilogEval diagnostic, 16 problems switch from wrong to right and four switch from right to wrong; the reported exact McNemar p-value is 0.012. Of the 16 repairs, 11 start with a correct Python interpretation and five recover from both-wrong outputs. Among 20 initial reference-correct/Verilog-incorrect problems, the workflow repairs 11 and misses nine; one initially correct Verilog output subsequently degrades, giving ten final failures with a correct reference. Of 39 total final failures, these ten divide into five turn-budget failures, three missed by the stimuli, one accepted despite a remaining bug, and one without complete compiling RTL. These counts describe a problem-level paired diagnostic and are not a re-estimate of the ten-run main table.

Repair bottleneck: `fsm_ps2`. The reference checks input bit 3 at the done cycle to handle a back-to-back packet. The golden transition is `DONE: next = in[3] ? BYTE2 : BYTE1;`, but the final RTL uses `S3: next_state = S0;`. The generated implementation drops a packet byte, and no sampled RTL passes the held-out test. A correct reference exposes the target behavior but cannot itself generate the required RTL repair.

Improved agreement with a timing bug: `fsm_serial`. The golden output, `assign done = (state == DONE);`, is registered one cycle later than the final candidate's `assign done = (state == STOP && in);`. The Python reference updates `done` after the clock edge, while the accepted RTL revision asserts it one cycle early. Its inter-agent match rate rises to 0.957 without removing that timing error. Monotonic match improvement therefore does not imply monotonic correctness improvement.

C.4 Coverage and Mutation Checks

Table 7: Finite-simulation coverage diagnostics.

Diagnostic	Value
Original mean FSM state-value coverage	78.3%
Original FSM mutant kill rate	95.5%
Original overall kill rate (376 mutants)	94.1%
Improved overall mutant kill rate	95.2%
FSM designs checked for empirical saturation	30
Maximum adaptive simulation length	200K cycles
Additional states beyond empirical saturation	0
Strengthened official observed-state coverage	$> 99.9\%$
Maximum absolute pass@1 change under strengthened official scoring	0.5 pp
4B pass@1: original / improved cross-verifier	75.0 / 75.4
4B conditional false agreement: original / improved	13.9% / 12.8%

On the identified FSM designs, the original stimuli reach 78.3% mean state-value coverage and kill 95.5% of injected FSM mutants; the overall kill rate is 94.1% over 376 mutants (Table 7). Extending simulations to 200K cycles finds no additional states beyond empirical saturation on the 30 identified FSM designs. The coverage checks distinguish the inference-time cross-verifier from the held-out benchmark scoring tests. For the former, active-low reset corrections and constrained-random corner vectors raise the overall kill rate to 95.2%; a separate 4B evaluation with this improved cross-verifier changes final-output pass@1 from 75.0 to 75.4 and conditional false agreement from 13.9% to 12.8%. For the latter, strengthening official FSM scoring tests changes pass@1 by at most 0.5 pp for each

evaluated model. The $> 99.9\%$ quantity refers to *observed-state* coverage in the strengthened scoring test, not a formal reachable-state denominator. Likewise, no new states after empirical saturation at up to 200K cycles and high injected-mutant kill rates do not rule out rare behaviors, unmodeled fault classes, or larger state spaces; finite simulation does not establish exhaustive reachable-state coverage or equivalence.

C.5 Uncertainty and the Final Overlap Audit

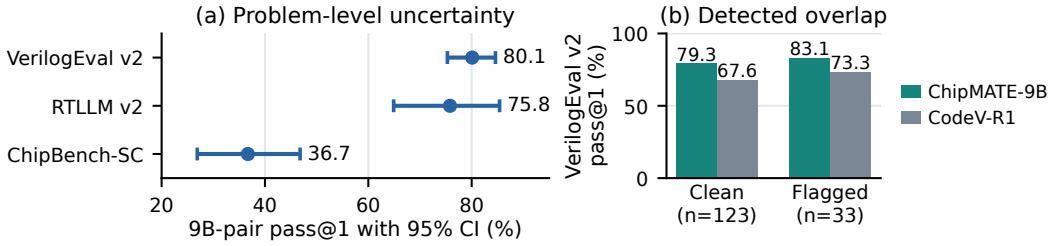

Figure 11: (a) 9B-pair pass@1 with 95% problem-bootstrap intervals. (b) VerilogEval accuracy on clean and overlap-flagged tasks.

Table 8: 9B-pair pass@1 with 95% problem-bootstrap intervals.

Benchmark	Pass@1 (%)	95% interval
VerilogEval v2	80.1	[75.3, 84.6]
RTLLM v2	75.8	[64.9, 85.4]
ChipBench-SC	36.7	[26.9, 46.8]

Table 9: Final overlap audit (AST: subtree-hash Jaccard ≥ 0.8) and clean-subset accuracy.

Benchmark	Token-flagged / total	AST confirmed / flagged
VerilogEval v2	33 / 156	21 / 33
RTLLM v2	4 / 50	3 / 4
ChipBench-SC	4 / 30	3 / 4
CVDP cid003	1 / 78	—
VerilogEval subset	**ChipMATE-9B**	**CodeV-R1**
Clean (123 tasks)	79.3	67.6
Flagged (33 tasks)	83.1	73.3

The confidence intervals in Figure 11(a) and Table 8 resample problems, preserving the distinction between evaluation uncertainty and training randomness. The paired comparison against CodeV-R1 yields $p = 0.003$ on VerilogEval. RTLLM and ChipBench-SC do not individually establish a significant advantage under the reported paired analysis, and their wider intervals warrant treating those differences as directional. Problem resampling does not measure the variation from retraining the complete pipeline, and the three Stage-2 seeds reported above do not replace independent training runs.

The full-corpus audit uses normalized token n-grams with document-frequency filtering, identifier-normalized Verilog ASTs, and manual inspection. The final audit in Table 9 supersedes an earlier provisional 35-flag/121-clean VerilogEval split. Normalized token screening covers the 87K public-source corpus, the 3.4K augmentation source, and the 64.4K reference-model corpus. The final token screen flags 33/156 VerilogEval, 4/50 RTLLM, 4/30 ChipBench-SC, and 1/78 CVDP cid003 tasks, and the reference corpus adds no benchmark-task flags beyond the source union. The AST stage uses MinHash-LSH only to retrieve candidate matches and exact subtree-hash Jaccard to classify them. It performs 703 exact comparisons and reports that 41 token-flagged tasks are parseable. Parse coverage is 234/236 audited benchmark designs and 77,362/87,321 source documents. At threshold 0.8, AST confirms 21/33 VerilogEval, 3/4 RTLLM, and 3/4 ChipBench flags. AST-only recovery adds no tasks

at thresholds 0.75, 0.8, or 0.9 among parseable documents; token screening also covers unparseable documents. The 236-design AST denominator is separate from the four-benchmark token screen; no CVDP AST confirmation count is reported. Only VerilogEval has a reported task-level clean-subset accuracy comparison: on the 123 clean tasks, the trained pair scores 79.3 versus 67.6 for CodeV-R1, a gap of 11.7 pp, and on the 33 flagged tasks the gap is 9.8 pp (83.1 versus 73.3). The detected overlap is therefore insufficient to explain that benchmark's advantage, without asserting zero contamination or complete decontamination of all four benchmarks.

C.6 Budget Sweep and Evaluation Consistency

Table 10: Additional attempt budgets. † quotes Table 1; budgets count both roles.

Model / benchmark	(N, T)	Pass@1	Max. completions	Max. simulations
9B / VerilogEval v2	(3,3)	80.1†	18	9
9B / VerilogEval v2	(4,3)	80.4	24	12
4B / RTLLM v2	(3,3)	74.6	18	9
4B / RTLLM v2	(3,4)	74.9	24	12

In the 4B configuration summary (Figure 6), $N = 1, T = 4$ and $N = 2, T = 5$ each give 75.6, while the default quotes the main evaluation's 75.0. In particular, $N = 1, T = 4$ has a lower maximum budget of eight completions and four pair simulations; these results do not justify a minimum-cost claim for $N = 3, T = 3$. The default's 95% problem-bootstrap interval is $[70.1, 79.9]$; the other grid cells retain single-run measurements. We do not attach a paired confidence interval or significance claim to a comparison that mixes these estimates. The 9B VerilogEval budget table likewise quotes the main evaluation's 80.1 for the default, compared with 80.4 at $N = 4, T = 3$; on 4B RTLLM, the default gives 74.6 versus 74.9 at $N = 3, T = 4$. Both larger settings in Table 10 increase the maximum budget by one third, from 18 to 24 completions and from nine to 12 simulations; this is a descriptive comparison, not evidence of an optimal configuration, and we retain $N = 3, T = 3$ as the evaluated default. On RTLLM, the 9B pair's deployable pass@1 is 75.8 versus 71.2 for CodeV-R1 SC@9, while CodeV-R1 has higher oracle-assisted pass@5 (78.2 versus 77.3). This again separates reliable selection of one output from oracle-reachable candidate coverage.

C.7 CVDP Scope and Backend Configuration

The cid004 results in Table 1 cover 52 benchmark-side-evaluable tasks, with ten independent runs per task. The input supplies an original RTL design and an explicit modification request, allowing both roles to derive the intended modified behavior independently. The 9B pair reaches 46.9 pass@1 versus 39.1 for its Verilog-only agent and 37.1 for CodeV-R1, and also improves oracle-assisted pass@5 over CodeV-R1 (52.3 versus 49.1), while Claude Opus 4.7 and GPT-5.5 remain ahead on both metrics. This supports transfer to an additional RTL output task within our evaluation protocol, rather than a cross-paper ranking or general verification capability. CVDP cid002 partial completion and verification-artifact categories cid012–014 remain outside the current output-task scope.

Thinking configuration. The original R1 row uses a V3-Base-derived reasoning-trained family, whereas the original V4 row uses non-thinking generation. The additional configurations in Figure 12 share one prompt and extractor, five samples per problem, and a 32K token cap, separately from the main 16K-token evaluation; invalid output, compilation failures, and timeouts all count as failures, and no uncertainty intervals were reported. For V4-Flash, enabling thinking raises VerilogEval from 74.9 to 82.7 but lowers RTLLM from 51.2 to 47.2. The benefit is benchmark-dependent under the fixed budget, so the historical R1/V4 gap cannot be attributed solely to intrinsic model quality or scale.

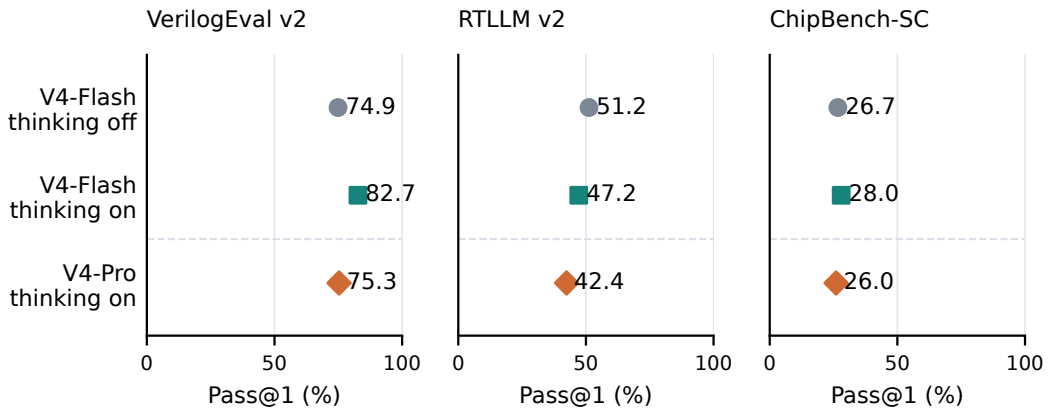

Figure 12: DeepSeek thinking-mode controls (32K-token cap, five samples per task; V4-Pro below the divider).

C.8 Synthesis-Level Physical Quality

Table 11: Synthesis quality on 26 common correct RTLLM designs: ratio to golden RTL (yosys/Nangate45; lower is better).

System	Area / golden	Logic depth / golden
ChipMATE	1.11	1.05
CodeV-R1	1.17	1.12
DeepSeek R1	1.09	1.04

Table 11 evaluates the common 26 RTLLM designs that pass and synthesize for every method, using yosys/Nangate45 and normalizing to golden RTL. The trained pair has area/depth ratios of $1.11/1.05$, compared with $1.17/1.12$ for CodeV-R1 and $1.09/1.04$ for DeepSeek R1, improving over CodeV-R1 on this subset with slightly larger ratios than DeepSeek R1. Conditioning on common correct designs makes physical-quality comparisons interpretable while limiting their coverage. These measurements do not provide power, placed-and-routed timing, or signoff-level PPA results.

D AGENT PROMPT TEMPLATES

We list the system and user-message templates used by the Verilog agent and the Python reference-model agent during single-turn SFT and multi-agent rollouts, together with the retry-and-fix prompt fragments that X-GRPO injects on turns $t > 0$ whenever a V/P mismatch is detected. Both agents share a unified `<think>...</think><answer>...</answer>` reasoning format.

D.1 VERILOG AGENT SYSTEM PROMPT

Verilog agent system prompt

```
You are a helpful assistant.  The assistant first thinks about the
reasoning process in the mind and then provides the user with the
answer.  The reasoning process and answer are enclosed within
<think> </think> and <answer> </answer> tags, respectively, i.e.,
<think> reasoning process here </think><answer> answer here
</answer>.  Now the user asks you to write verilog code.  After
thinking, when you finally reach a conclusion, enclose the final
verilog code in ‘‘‘verilog ‘‘‘ within <answer> </answer> tags.  i.e.,
<answer> ‘‘‘verilog\n module top_module(in, out, ...)  ...  ‘‘‘
</answer>.
```

Example user message. The user message gives the natural-language behaviour and a structured port list (no Verilog code skeleton—only the interface):

Example user message (Verilog agent, VerilogEval-style)

```
I would like you to implement a module named TopModule with the
following interface.  All input and output ports are one bit unless
otherwise specified.
  - input  in  (3 bits)
  - output out (2 bits)
The module should implement a "population count" circuit that counts
the number of '1's in the input vector.
```

D.2 PYTHON REFERENCE-MODEL AGENT SYSTEM PROMPT

The Python agent operates in a *skeleton-completion* mode: the user message contains a natural-language spec *plus* a Python class skeleton that pins the class name, method signature, and exact signal names / bit-widths. The agent must complete the bodies of `__init__` and `eval` without renaming any identifier so that the cross-language verifier can drive the resulting class identically to the Verilog DUT.

Python reference-model agent system prompt

```
You are an expert hardware modeling assistant.  Your task is to
write Python code that serves as a cycle-accurate reference model
for a Verilog hardware module.  Your Python code's outputs must
match the expected outputs of the Verilog circuit exactly.
The user will provide a natural language description of the module's
behavior and a Python code skeleton.  You must complete the provided
skeleton.  Do not rename the class, method, or any signals.  Use the
exact signal names and bit-widths from the skeleton.
GENERAL RULES:
  - Use bitwise operators:  &  |  ^  ~  »  «.  Each eval() call = one
rising clock edge.  Never read clk.
  - Mask all outputs and state variables to their correct bit-width,
e.g. val & ((1 « N) - 1).
```

```
FOR COMBINATIONAL CIRCUITS (no reset, no state):
  - Compute outputs directly from inputs in eval().  No state
variables needed.
FOR SEQUENTIAL CIRCUITS (has reset and/or state):
  - Declare all state variables in __init__(), initialized to 0.
  - In eval():  if reset is asserted, set all state and outputs to 0
and return.
  - Otherwise:  compute next-state from current state (self.*) and
inputs, compute outputs, then update self.* at the end.
First, think step-by-step inside <think> </think> tags.  Then,
provide your final Python code inside <answer> </answer> tags with
"'python "' markers.
```

Example user message (with skeleton). The user prompt injects the spec, a signal table, and a pre-populated `class TopModule` skeleton in which the state variables and the input bit-masks are already laid out; the agent only fills in the sequential / combinational logic and the output expression.

Example user message (Python agent, 3-stage pipelined ALU)

```
Design a pipelined ALU that performs a sequence of arithmetic
operations in three stages.  The ALU should take four 10-bit inputs
(a, b, c, d) and two clock signals (clk1, clk2) and produce a 10-bit
output (F). The operations are as follows:
  1.  Stage 1:  Compute a + b and c - d, and store the value of d.
  2.  Stage 2:  Compute the sum of the results from Stage 1.
  3.  Stage 3:  Compute the product of the result from Stage 2 and
the stored value of d.
The ALU should use pipelining to ensure that each stage operates
independently, and the final result should be available at the
output F. This Python class, named alu_op, has the interface
designed in a port table (signal name, direction, width,
description) for each of a, b, c, d, clk1, clk2, and F.
Complete this skeleton:
"'python
class TopModule:
  def __init__(self):
    self.stage1_d = 0
    self.stage1_diff = 0
    self.stage1_sum = 0
    self.stage2_sum = 0
    self.stage3_product = 0
  def eval(self, inputs:  dict) -> dict:
    a = inputs.get("a", 0) & 0x3FF
    b = inputs.get("b", 0) & 0x3FF
    c = inputs.get("c", 0) & 0x3FF
    d = inputs.get("d", 0) & 0x3FF
    # TODO: implement sequential logic
    return {"F":  ...}
"'
Answer:
```

D.3 Retry-and-Fix Prompt Fragments (X-GRPO Correction Turns)

When the previous turn's V/P pair produced any mismatch on the random-stimuli cross-verification, X-GRPO appends three fragments to the *user message* of the next turn for the chosen best pair: previous code attempts (`{previous_code}`), the diff log (`{previous_error_log}`), and a one-sentence refinement instruction (`{refine_instr}`). On $t = 0$ all three are empty strings, recovering the single-turn behaviour shown above. We display the Verilog-side fragments; the

Python-side is structurally identical (substitute `verilog`→`python`, `module code`→`class TopModule`, `//`→`#`).

(a) Previous-code fragment `{previous_code}`. The two most recent valid attempts (truncated to 1500 chars each) are re-injected so the agent can see its own trajectory:

Previous-code fragment (Verilog side)

```
Your previous Verilog attempts:
Attempt k−1:
"'verilog
<truncated previous code, ≤ 1500 chars>
// ...(code truncated)...
"'
Attempt k:
"'verilog
<truncated previous code, ≤ 1500 chars>
// ...(code truncated)...
"'
```

(b) Verification-error fragment `{previous_error_log}`. A structured diff of the cross-verifier's per-stimulus mismatches (truncated to 2000 chars). Crucially, the message frames the disagreement as *mutual*: the agent is told its peer might be the one at fault, preventing the V agent from over-correcting when the Python reference is actually wrong.

Verification-error fragment (Verilog side)

```
Previous verification error:
Verilog vs Python:  m/N mismatches across T test vectors.
First mismatches (got = your Verilog, exp = peer Python):
  Test 0, signal 'out':  got=42, exp=37
    (inputs:  a=5, b=3, clk=1)
  Test 4, signal 'out':  got=12, exp=8
    (inputs:  a=2, b=6, clk=1)
  ...(up to 5 mismatches shown)...
Check your logic carefully.  Either you or the Python agent is wrong
--
only change your code if you think your previous code is wrong.
```

(c) Refinement instruction `{refine_instr}`. A short, fixed instruction emitted only on $t > 0$ turns:

Refinement instruction

```
(Verilog side)
Please refine your Verilog code to improve correctness and quality.
You MUST output the complete module code in "'verilog"' blocks.
(Python side)
Please refine your Python code based on the mismatch feedback above.
Output complete class TopModule in "'python"' blocks.
```