VERI-SURE: Multi-Agent RTL Code Generation with Temporal Tracing, Slicing and Formal Verification

Jiale Liu
The University of Edinburgh
Jiale.Liu@ed.ac.uk

Taiyu Zhou
University of Macau
yc57996@um.edu.mo

Tianqi Jiang
CUHK, Shenzhen
tianqijiang@link.cuhk.edu.cn

Abstract

In the rapidly evolving field of Electronic Design Automation (EDA), the deployment of Large Language Models (LLMs) for Register-Transfer Level (RTL) code generation has emerged as a promising direction. However, because RTL code describes concurrent hardware behavior with strict cycle-level timing and thus differs fundamentally from software code, LLM-based RTL code generation remains bottlenecked by (i) sparse feedback and limited test coverage of simulation-centric evaluation, (ii) regressions and repair hallucinations introduced by whole-file regeneration during debugging, and (iii) semantic drift as intent is reinterpreted across repair iterations and agent handoffs. In this work, we propose VERI-SURE, a multi-agent framework that establishes a design contract to align agents' intent and uses a patching mechanism guided by static dependency slicing to perform precise, localized repairs. By integrating a multi-branch verification pipeline that couples trace-driven temporal analysis with formal checking, VERI-SURE can generate RTL code with better reliability and robustness. We also introduce VERILOGEVAL-V2-EXT, extending the original benchmark with 53 more industrial-grade design tasks and stratified difficulty levels, and show that VERI-SURE achieves state-of-the-art verified-correct RTL code generation performance, surpassing standalone LLMs and prior agentic systems. Code and dataset are available at GitHub.

1 Introduction

The explosive demand for computing power in the era of artificial intelligence and digital transformation continues to push chip designs to billions of transistors, raising the bar for productivity and correctness in Electronic Design Automation (EDA). In practice, the central artifact in the design flow is the Register-Transfer Level (RTL) description, typically written in Hardware Description Languages (HDLs) such as Verilog and SystemVerilog, which specifies clocked state and the combinational logic between registers, precisely capturing concurrent circuit behavior under strict timing constraints and enabling downstream synthesis and implementation. As a result, writing high-quality, synthesizable RTL remains an indispensable yet costly step in modern chip development.

Large language models (LLMs) have recently made rapid progress in software code generation [16, 12, 21, 15], suggesting a compelling opportunity to generate RTL code directly from natural-language specifications. However, RTL code generation poses requirements that are fundamentally different from those of software, since HDL describes concurrent hardware behavior with strict cycle-level timing, protocol semantics, and hierarchical structural constraints [2]. As a result, general-purpose coding models, which are optimized for autoregressive, sequential text generation, often achieve surface-level syntactic correctness yet fail to reliably preserve temporal intent, handshake protocols, or globally consistent structure [39]. These failures are worsened by the scarcity of high-quality HDL data relative to software corpora, which limits the models' exposure to high-quality RTL patterns and actual verification-driven development practices [24, 51].

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

To bridge this gap, recent methods have strengthened LLM-based RTL code generation through domain adaptation and tool augmentation, including supervised fine-tuning (SFT) [39, 26, 24] and reinforcement learning (RL) [51, 28] on curated HDL corpora, retrieval of reference implementations [9, 23], and tool-in-the-loop prompting with simulation feedback [38, 13, 49]. While these techniques improve compile rates and module-level functional pass rates compared to baselines, they remain insufficient for industrial-scale RTL blocks where correctness depends on subtle temporal corner cases and multi-iteration refinements. In particular, we identify that existing LLM-based pipelines are still bottlenecked by (i) sparse feedback and limited test coverage of simulation-centric evaluation, (ii) regression risk, repair hallucinations, and inefficiency caused by whole-file regeneration during debugging, and (iii) semantic drift as cycle-accurate intent is gradually reinterpreted across repair iterations and agent handoffs. Given that hardware bugs can survive simulation and lead to costly tape-out failures, an effective RTL code generation system should not only be able to generate HDL code but also independently perform verification and debugging to ensure the reliability and correctness of the whole design.

Moreover, progress toward industrial-grade RTL code generation is difficult to measure with existing public benchmarks. Widely used datasets such as VerilogEval-v2 [34] are dominated by short, pedagogical modules and under-represent key industrial needs like advanced arithmetic units, communication protocols, buffer/memory control, Clock Domain Crossing (CDC) related patterns, and pipeline-style designs. They also lack difficulty stratification for diagnosing the failure modes of systems.

To address these challenges, we propose VERI-SURE, a contract-driven multi-agent framework with trace-driven temporal analysis and formal verification. VERI-SURE first converts the natural language specification into a structured design contract that fixes interface semantics and cycle-accurate intent, then uses this contract as the shared source of truth across agents. It then generates candidate RTL code and validates it through simulation. When simulation fails, VERI-SURE performs diagnosis using the temporal analysis and contract-derived formal checks, including assertion checking and Boolean equivalence proofs. Guided by the diagnosis, it applies dependency-slicing-guided patching to perform precise, localized repairs, avoiding unstable whole-file rewrites. Finally, we introduce VERILOGEVAL-V2-EXT, which extends VerilogEval-v2 with 53 industry-grade tasks and a structure-based difficulty level classification to enable finer-grained evaluation. Our main contributions are summarized as follows:

- We design VERI-SURE, a multi-agent framework for RTL code generation that tightly couples LLM code synthesis with an automated generate-verify-debug loop, leveraging simulation, trace-driven temporal analysis, and formal verification.
- We propose a dependency-slicing-guided patching mechanism that leverages static dependency slicing over signal relationships to localize failing behaviors and apply precise, minimal edits, avoiding costly whole-file regeneration and reducing regression risks and repair hallucinations.
- We integrate a multi-branch verification and debugging pipeline that goes beyond simulation by combining trace-driven temporal analysis with formal verification, including assertion-based checking and Boolean equivalence proofs, bringing more reliability and robustness to the debugging loop.
- We introduce VERILOGEVAL-V2-EXT, extending the original benchmark with expanded task coverage and stratified difficulty levels, and demonstrate state-of-the-art verified-correct RTL code generation performance against standalone LLMs and agentic baselines.

2 Related Work

2.1 Automatic RTL Code Generation

Early studies explored open-loop RTL code generation with general-purpose LLMs via zero-shot or few-shot prompting, and domain-adapted models such as ChipNeMo [23] demonstrated the promise of natural-language-to-RTL workflows. These efforts also highlighted a recurring gap that models may produce syntactically plausible Verilog code while violating cycle-accurate behavior, protocol semantics, or synthesizability constraints.

To improve domain alignment, subsequent work moved beyond vanilla prompting toward domain-adapted training and generation scaffolding. Fine-tuning on curated HDL corpora, as demonstrated by VeriGen [39] and RTLLM [26], improves syntax validity and basic module-level functionality. RTL-Coder [24] further filters examples through verification to reduce noisy RTL patterns. BetterV [33] introduces a generative discriminator to steer outputs toward more verifiable implementations, while CodeV-R1 [51] trains a reasoning model with a distill-then-RL pipeline whose rewards come from testbench-based equivalence checking. Orthogonally, AutoVCoder [9] leverages retrieval-augmented generation to condition synthesis on relevant reference designs.

A complementary line of work strengthens RTL code generation by learning from structured supervision. VeriSeek [41] incorporates structure-aware objectives (e.g., abstract syntax tree (AST)-based signals) to encourage hardware-consistent patterns beyond test pass rates, while QiMeng-SALV [48] introduces signal-aware learning to emphasize critical control-signal behavior. VeriLogos [28] combines synthetic augmentation with reinforcement learning to improve both diversity and correctness, and DeepRTL [25] explores curriculum-style training to better align natural-language intent with RTL semantics.

Despite steady progress, these methods rely mostly on simulation-centric supervision and lack the iterative repair central to real-world EDA workflows, so they still struggle with concurrent circuit behavior, strict timing requirements, and rare temporal corner cases.

2.2 Tool-Integrated LLM Systems for HDL

Recent work has moved toward tool-in-the-loop and agentic pipelines that iteratively generate RTL, run simulation, and repair failures. ChatEDA [43] showed that an LLM agent can orchestrate EDA tools through their interfaces, AutoChip [38] closes the loop using simulator feedback, and RTLFixer [40] supports iterative debugging with error-driven edits. These approaches reduce syntax errors and shallow functional bugs. However, their debugging signals are dominated by sparse pass/fail outcomes or textual logs, which provide limited observability for diagnosing timing-dependent failures and localizing root causes in multi-cycle designs.

As tasks scale to larger subsystems and multi-module designs, multi-agent frameworks [22, 20] have been proposed to decompose the workflow into planning, coding, and checking. REvolution [29] frames generation as an evolutionary search with LLM-driven mutations, VerilogCoder [13] coordinates collaborative agents for generation and validation, and MAGE [49] improves candidate quality via diversified sampling and targeted debugging. Despite this progress, most agentic systems remain heavily dependent on test coverage, often apply coarse-grained edits that risk regressions, and can accumulate semantic drift across iterations and agent handoffs.

2.3 Formal Verification and Automated Debugging

Formal verification aims to mathematically prove the correctness of a design against a specification and is thus also useful for catching bugs in RTL code. Assertion-based verification [8] embeds temporal properties in the design, which bounded model checking with Boolean satisfiability (SAT) solvers [4] can check up to a fixed depth. Since hand-written assertions are costly, AssertLLM [45] uses multiple LLMs to generate them from design specifications. For combinational logic, equivalence checking reduces comparing two circuits to proving their miter unsatisfiable [5], which tools such as SymbiYosys [47] can easily automate with satisfiability modulo theories (SMT) solvers like Z3 [6].

To achieve automated debugging, prior work has explored both software and hardware contexts. In software, program slicing [42] narrows a failure to the statements that can affect it, and generate-and-validate repair [18] searches for patches that pass a test suite. In hardware, CirFix [1] repairs Verilog by genetic programming and RTL-Repair [17] synthesizes minimal repairs with an SMT solver, but both draw edits from fixed mutation operators or repair templates. LLM-based debuggers lift this limit: MEIC [44] iterates a debug agent with an LLM scoring agent that triggers rollbacks, UVLLM [14] couples LLM repair with testing under the Universal Verification Methodology (UVM), and LiK [46] localizes bugs from the specification and code alone.

3 The VERI-SURE Framework

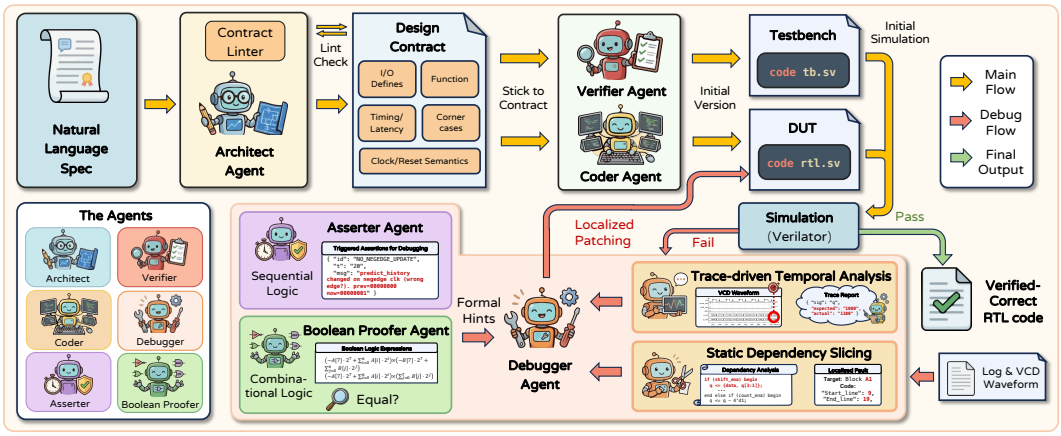


Figure 1: An overview of our VERI-SURE framework.

We propose VERI-SURE, a multi-agent framework that brings a real EDA-style closed loop for development and testing to RTL synthesis from natural language. Rather than treating generation as a one-shot translation problem, VERI-SURE tightly couples code synthesis with automated validation and actionable debugging signals. Unlike prior multi-agent systems that primarily coordinate through natural-language context and thus risk specification drift and coarse, regression-prone rewrites, VERI-SURE uses trace-driven analysis to localize failures to concrete time windows, signals, and violated requirements, enabling targeted fixes and improving robustness on multi-cycle corner cases. It also integrates simulation with formal checks to continuously enforce design requirements and aid debugging.

3.1 Overall Architecture

Figure 1 shows the workflow of VERI-SURE, a single structured workflow in which agents cooperate by passing typed artifacts rather than free-form messages. The process starts with (1) **an Architect agent**, which distills the user prompt into a structured JSON-format design contract, capturing the Device Under Test (DUT) interface (ports, clock/reset conventions), key parameters, and cycle-accurate behavioral requirements. When a reference testbench is available, the Architect cross-checks interface and timing details to reduce ambiguity. A linter then validates the contract, and on lint errors, the Architect is re-invoked with the error report. The linted contract is then passed to (2) **a Verifier agent**, which produces a self-checking testbench whose stimuli and checkers are aligned with the specified requirements, and to (3) **a Coder agent**, which generates synthesizable RTL code. Verilator is then invoked to compile and simulate the code against the testbench. If all checks pass, the design is accepted.

When the compilation or simulation fails, VERI-SURE enters the debugging loop, in which the failing code is passed to (4) **a Debugger agent**. Rather than relying solely on textual error messages, the Debugger calls trace-driven temporal analysis and static dependency slicing tools, which turn the raw failure into a trace report of the failing time window and signals, plus a small set of suspect RTL blocks. To provide a denser diagnostic signal, the Debugger also invokes (5) **an Asserter agent** to generate temporal assertions that expose sequential/protocol violations, and (6) **a Boolean Proofer agent** to check combinational logic against the contract for counterexamples. Their outputs are injected into the Debugger's prompt as hints. The Debugger then patches only the suspect blocks, rolling back any patch that does not improve the failure signature, and iterates until all checks pass or the repair budget is exhausted.

3.2 Contract Design

Natural-language RTL specifications are often underspecified in details like reset polarity, sampling edge, or cycle latency, which easily leads to inconsistent interpretations across agents. VERI-SURE

mitigates this by having the Architect agent compile the prompt into a compact, structured design contract that makes such choices explicit before any code is written.

The resulting contract records the chosen source of truth, a typed module interface, clock/reset and sequential semantics, per-output latency, a precise functional summary with corner cases, a directed test plan, and guidance for the Verifier, Coder, and Debugger. When the prompt underspecifies details, the Architect makes assumptions explicit so they become checkable requirements. A lightweight contract linter then validates the schema and basic consistency, preventing malformed or contradictory contracts from propagating into code generation and verification.

3.3 Tracing, Static Slicing & Patching

When simulation fails, the main challenge is to turn an opaque mismatch count into a minimal, actionable edit without destabilizing unrelated logic. VERI-SURE addresses this with a closed-loop debugging pipeline that couples dynamic evidence (i.e., waveforms) with static structure (i.e., code dependencies), producing a patching task for the Debugger rather than a full-file rewrite, as shown in Figure 2.

Trace-Driven Temporal Analysis. From the simulator log we identify the earliest divergence time and the signals responsible. Let $\mathbf{O}_{\mathrm{DUT}}(t)$ and $\mathbf{O}_{\mathrm{exp}}(t)$ denote the observed and expected output vectors at time t, we define:

$$t_f \;=\; \min\{\, t \mid \mathbf{O}_{\mathrm{DUT}}(t) \neq \mathbf{O}_{\mathrm{exp}}(t) \,\}. \tag{1}$$

We then extract a short waveform window ending at t_f from the Value Change Dump (VCD) file and sample it on the contract-defined clocking scheme. In addition to raw values, the trace reporter runs a lightweight alignment check to detect systematic off-by-one-cycle or wrong-edge bugs by testing small shifts and reporting the best-matching offset as a timing hint. All findings are summarized into a trace report, including failing signals, the failure cycle, and the relevant trace slice.

Static Dependency Slicing. To avoid unfocused edits, the trace slicer restricts attention to the cone of logic that can influence the failing outputs. We parse the RTL into semantic blocks, for example using continuous assignments and always blocks, and compute per-block read/write sets $R(B)$ and $W(B)$. A dependency edge exists when a block reads what another writes:

$$B_j \leftarrow B_i \quad \Leftrightarrow \quad R(B_j) \cap W(B_i) \neq \varnothing. \tag{2}$$

Starting from the failing signals as seeds, we perform a bounded backward traversal to obtain a small suspect set $\mathcal{B}_{\mathrm{sus}}$ with block identifiers and line ranges. This converts "the output mismatched" into "these few blocks are the plausible causes," dramatically shrinking the search space presented to the Debugger.

Localized Patching. The Debugger is only allowed to edit blocks in $\mathcal{B}_{\mathrm{sus}}$ via block-level read/replace operations; the rest of the file is preserved verbatim. Each patch is immediately checked by recompilation and re-simulation, and we apply a simple rollback rule to prevent regressions. Writing $\sigma(\mathrm{RTL}) = (t_f, m)$ for the first-failure time and total mismatch count, we keep a patch only if it improves the failure signature, which means later first failure, or fewer mismatches at the same t_f; otherwise we revert. This loop grounds the repair process in concrete traces while maintaining stability across iterations.

3.4 Formal Verification

Simulation provides concrete counterexamples but offers limited coverage and often weak and sparse diagnostic signals. VERI-SURE therefore augments the debugging loop with a two-branch, contract-driven verification pipeline: an Asserter for sequential/timing obligations and a Boolean Proofer for combinational equivalence. From the design contract $\widehat{C}$, we derive a small set of checkable obligations:

$$\Phi \;=\; \mathcal{P}(\widehat{C}) \;=\; \Phi_{\mathrm{seq}} \cup \Phi_{\mathrm{comb}}, \tag{3}$$

which are then discharged by the two verifiers and translated into structured hints for the Debugger.

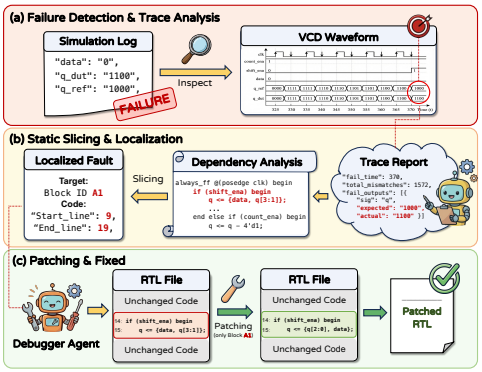


Figure 2: The tracing, static slicing and patching mechanism.

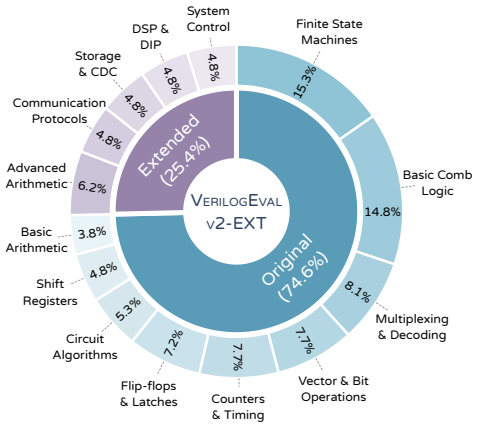


Figure 3: VERILOGEVAL-V2-EXT problem taxonomy.

Asserter for Timing Supervision. Using clock/reset semantics and latency annotations in $\widehat{C}$, the Asserter generates a small set of non-intrusive SystemVerilog assertions and attaches them to the DUT via binding. Assertions run under Verilator and report violations with type, implicated signals, and time, effectively diagnosing issues such as wrong-edge sampling, reset mismatches, and off-by-one-cycle latency.

Boolean Proofer for Combinational Equivalence. For purely combinational obligations, the Boolean Proofer first infers candidate combinational targets from $\widehat{C}$ (where latency $= 0$) and filters out sequentially driven signals via a dependency analysis, avoiding invalid proofs on stateful logic. It then synthesizes a compact reference combinational model from the contract's functional summary and constructs a standard miter that compares DUT and reference under identical inputs. Writing the miter error as

$$e(\mathbf{x}) \;=\; \bigvee_{y \in \mathcal{Y}_{\text{comb}}} \Big(y_{\text{DUT}}(\mathbf{x}) \neq y_{\text{spec}}(\mathbf{x}) \Big), \tag{4}$$

the goal is to prove $\forall \mathbf{x},\ e(\mathbf{x}) = 0$ using SymbiYosys. If the proof fails, the resulting counterexample is distilled into a concrete input assignment.

The formal hints produced by the branches are injected into the Debugger prompt, separating timing faults from pure Boolean faults and improving repair generalization. Since these formal artifacts are generated from LLM-derived specifications, we use them only as diagnostic hints and the final acceptance still requires passing the golden testbench. The reliability of these hints is analyzed in Appendix C.3.

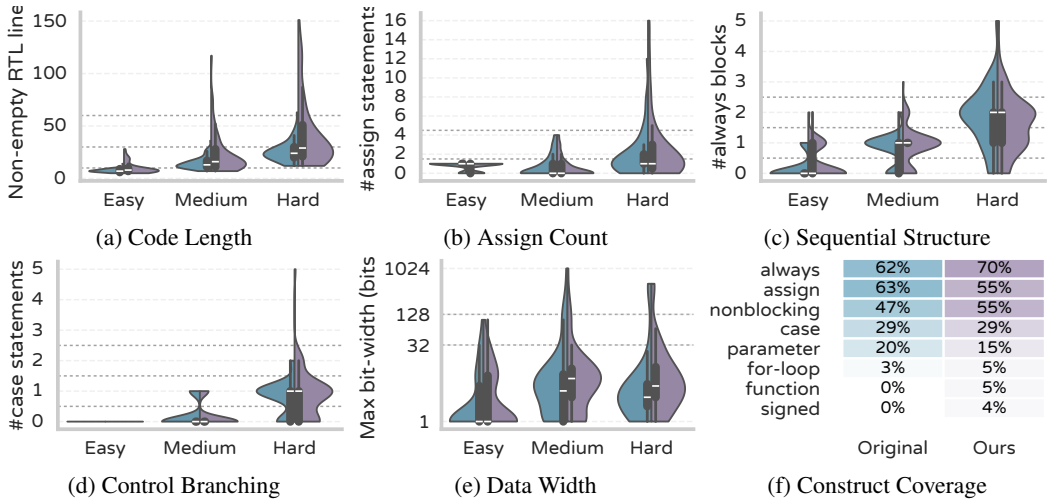
4 The VERILOGEVAL-V2-EXT Benchmark

We introduce VERILOGEVAL-V2-EXT, an expanded benchmark for evaluating LLMs on realistic RTL code generation. It extends VerilogEval-v2 [34] with 53 new, industrial-grade design tasks, increasing the total from 156 to 209 problems. All tasks follow the same evaluation protocol as VerilogEval-v2: each problem provides a natural-language specification, a fully specified module interface, and an executable testbench used for functional checking, enabling direct comparison with prior results.

4.1 Industrial-Grade Extension

VerilogEval-v2 offers strong coverage of basic digital logic problems, but it under-represents modules that dominate real RTL development (e.g., protocol controllers, buffering, and multi-cycle datapaths). As a result, models may achieve high scores while still failing on engineering-critical behaviors such as latency alignment, corner-case handling, and stateful control logic.

To address these gaps, we curated 53 new problems that reflect common Intellectual Property (IP) blocks and system components, including communication protocols and buffering (e.g., Universal



(a) Code Length (b) Assign Count (c) Sequential Structure

(d) Control Branching (e) Data Width (f) Construct Coverage

Figure 4: Dataset complexity statistics comparing VerilogEval-v2 and VERILOGEVAL-V2-EXT (full extended dataset). (a–e) show distributions over problems for different metrics and (f) reports the Verilog construct coverage.

Asynchronous Receiver-Transmitter (UART) interfaces and First-In-First-Out (FIFO) queues), control-dominated designs (e.g., nontrivial Finite State Machines (FSMs)), and richer datapath modules (e.g., advanced arithmetic and DSP-style kernels). Figure 3 summarizes the resulting benchmark composition and details are provided in Appendix A.

4.2 Difficulty Annotations

To support finer-grained analysis beyond an aggregate pass rate, we stratify the 209 problems into Easy/Medium/Hard using a rule-based complexity score derived from observable structural signals, including lines of code, counts of assign/always/case constructs, and datapath width, supplemented with manual review. Full scoring details and thresholds are provided in Appendix A.3. Figure 4 shows that the extension increases structural complexity, especially within the Hard subset (e.g., longer solutions with more sequential structure and wider datapaths, up to 1024 bits). Together, these additions make VERILOGEVAL-V2-EXT a more discriminative and industry-aligned testbed for evaluating end-to-end RTL code generation and debugging.

5 Experiments

5.1 Experimental Setup

Benchmark. We evaluate on our VERILOGEVAL-V2-EXT benchmark and report Pass@1 for syntax and functional correctness. Syntax success means the simulator compiles the generated RTL code without errors. Functional success means the compiled DUT passes the provided testbench. Following MAGE [49], the golden testbench is provided to the Architect and Verifier as optional context and also serves as the final pass condition. "Verified-correct" denotes functional verification in the standard EDA sense, meaning that the design passes the golden testbench.

Baselines. We compare VERI-SURE against 15 standalone LLMs [32, 3, 11, 36, 35, 31, 19, 30, 7, 27, 50, 10, 48, 24, 51, 28], covering both commercial closed-source and recent open-weight models, that generate the DUT in a single attempt. We also include single-agent simulator-feedback baselines that wrap most models with an iterative compile/simulate loop, feeding Verilator logs back for regeneration. Finally, we evaluate representative multi-agent frameworks, MAGE [49] and VerilogCoder [13] with the latest backbone model and report all of the comparisons in Table 1.

Toolchain and Computing. All methods use the same testbenches and verification toolchain. We use Verilator [37] for syntax checking and simulation, chosen for its better support of SystemVerilog assertion-style checks used by our Asserter. For Boolean proof, we use SymbiYosys [47] with a

Z3 [6] backend. Commercial models are queried via official APIs; open-weight models are run locally on NVIDIA A100 80GB GPUs. Iterative methods are capped at most $K=10$ repair iterations.

5.2 Main Results

Table 1: Performance comparison on VERILOGEVAL-V2-EXT (Pass@1, %). **Params** reports total (active) parameters for MoE models, and total for dense models; "w." denotes "with" the specified backbone. Background colors indicate rankings in each column: 1st , 2nd , and 3rd denote the **Global Performance** across all methods. Blue highlights the **Group Best** performance if it is not in the global top-3. We assign these badges for standalone LLMs: **Best Open-Weight**: Top-performing open-weight model. **High Potential**: Largest relative gain when enhanced by agents. **Efficiency King**: Best performance-to-parameter ratio. **Reasoning Expert**: Best performance on the "Hard" subset. **Robust Performer**: Minimal gap between Syntax and Functional scores.

Method	Params	Easy ($n=51$)		Medium ($n=91$)		Hard ($n=67$)		Overall	
		Syn.	Func.	Syn.	Func.	Syn.	Func.	Syn.	Func.
Standalone LLMs									
GPT-5.2	-	100.00	94.12	100.00	79.12	100.00	59.70	100.00	76.56
Claude-4.5-Sonnet	-	100.00	90.20	100.00	74.73	97.01	53.73	99.04	71.77
Gemini-3-Pro	-	100.00	94.12	100.00	85.71	100.00	62.69	100.00	80.38
Qwen3-Max	-	96.08	86.27	93.41	54.95	86.57	37.31	91.87	56.94
Mistral-Medium-3.1	-	100.00	78.43	87.91	52.75	77.61	19.40	87.56	48.33
DeepSeek-3.2	685B (37B)	96.08	80.39	96.70	64.84	85.07	41.79	92.82	61.24
Qwen3-Coder-Plus	480B (35B)	92.16	80.39	91.21	60.44	77.61	29.85	87.08	55.50
LLaMA-4-Maverick	402B (17B)	100.00	86.27	85.71	53.85	76.12	32.84	86.12	55.02
GLM-4.7	358B (32B)	96.08	90.20	86.81	70.33	71.64	44.78	84.21	66.99
Devstral-2	123B	98.04	80.39	91.21	58.24	67.16	25.37	85.17	53.11
Ministral-3-14B	14B	94.12	70.59	71.43	29.67	55.22	11.94	71.77	33.97
QiMeng-SALV	7B	96.08	66.67	95.60	53.85	88.06	22.39	93.30	46.89
RTL-Coder	6.7B	94.12	64.71	83.52	29.67	65.67	5.97	80.38	30.62
CodeV-R1	7B	88.24	74.51	92.31	54.95	64.18	23.88	82.30	49.76
VeriLogos	7B	86.27	49.02	90.11	26.37	71.64	1.49	83.25	23.92
Single Agent Systems *(w. Simulator Feedback & Iterative Fix)*									
w. GPT-5.2	-	100.00	96.08	100.00	81.32	100.00	59.70	100.00	77.99
w. Gemini-3-Pro	-	100.00	96.08	100.00	86.81	100.00	70.15	100.00	83.73
w. Qwen3-Max	-	100.00	94.12	92.31	68.13	89.55	44.78	93.30	66.99
w. Mistral-Medium-3.1	-	98.04	78.43	95.60	56.04	89.55	31.34	94.26	53.59
w. DeepSeek-3.2	685B (37B)	100.00	84.31	97.80	67.03	94.03	44.78	97.13	64.11
w. Qwen3-Coder-Plus	480B (35B)	98.04	80.39	93.41	63.74	88.06	31.34	92.82	57.42
w. LLaMA-4-Maverick	402B (17B)	100.00	92.16	91.21	59.34	79.10	32.84	89.47	58.85
w. GLM-4.7	358B (32B)	98.04	94.12	96.70	81.32	92.54	58.21	95.69	77.03
w. Devstral-2	123B	100.00	78.43	94.51	57.14	88.06	34.33	93.78	55.02
w. Ministral-3-14B	14B	94.12	76.47	79.12	48.35	55.22	19.40	75.12	45.93
w. QiMeng-SALV	7B	98.04	74.51	94.51	51.65	88.06	17.91	93.30	46.41
w. RTL-Coder	6.7B	84.31	56.86	80.22	38.46	70.15	8.96	77.99	33.49
w. CodeV-R1	7B	98.04	84.31	97.80	61.54	85.07	32.84	93.78	57.89
w. VeriLogos	7B	96.08	56.86	86.81	24.18	82.09	4.48	87.56	25.84
Multi Agents Systems									
MAGE (w. GPT-5.2)	-	100.00	96.08	100.00	95.60	98.51	77.61	99.52	89.95
VerilogCoder (w. GPT-5.2)	-	100.00	94.12	100.00	90.11	100.00	68.66	100.00	84.21
VERI-SURE (w. DS-3.2)	685B (37B)	100.00	90.20	98.90	73.63	97.01	53.73	98.56	71.29
VERI-SURE (w. GPT-5.2)	-	**100.00**	**100.00**	**100.00**	**95.60**	**100.00**	**85.07**	**100.00**	**93.30**

Table 1 reports the Pass@1 for syntax and functional correctness on VERILOGEVAL-V2-EXT, further broken down by difficulty. We observe these key things:

Frontier closed models are strong out of the box: Among standalone LLMs, commercial closed-source models deliver consistently high syntax success and strong functional accuracy, especially on Medium and Hard tasks. This suggests that large-scale general pretraining already confers substantial competence in RTL code generation and code-level reasoning, even without hardware-specific fine-tuning.

Table 2: Ablation results of VERI-SURE on VERILOGEVAL-V2-EXT (Func. Pass@1, %). Base model: GPT-5.2. Subscripts denote absolute drop in percentage points vs. the full framework.

Variant	Hard	Overall
GPT-5.2 (Standalone)	$59.70_{\downarrow 25.4}$	$76.56_{\downarrow 16.7}$
+ Simulator Feedback & Fix	$59.70_{\downarrow 25.4}$	$77.99_{\downarrow 15.3}$
VERI-SURE	**85.07**	**93.30**
w/o Design Contract	$82.09_{\downarrow 3.0}$	$90.43_{\downarrow 2.9}$
w/o Tracing only	$76.12_{\downarrow 9.0}$	$88.04_{\downarrow 5.3}$
w/o Slicing only	$77.61_{\downarrow 7.5}$	$89.00_{\downarrow 4.3}$
w/o Patching only	$79.10_{\downarrow 6.0}$	$89.95_{\downarrow 3.4}$
w/o Tracing, Slicing & Patching	$68.66_{\downarrow 16.4}$	$82.30_{\downarrow 11.0}$
w/o Formal Verification	$73.13_{\downarrow 11.9}$	$89.47_{\downarrow 3.8}$

Specialized small models lag behind general large models: In contrast, smaller models that are fine-tuned for RTL code generation do not necessarily outperform state-of-the-art general-purpose large models. While these models often achieve reasonable syntax rates, their functional pass rates are markedly lower on the harder tasks, indicating that parameter scale and broad pretraining remain critical for multi-step temporal reasoning, corner cases, and control-heavy designs. Practically, this gap is difficult to close by fine-tuning alone because training large backbones is compute-intensive.

Feedback improves results, but gains vary by backbone: Simply feeding simulator logs back to the same model improves syntax and functional accuracy for most backbones, but the magnitude is model-dependent. Stronger models tend to benefit modestly, while weaker or less robust models can gain substantially.

Multi-agent approaches are effective and VERI-SURE performs best: Multi-agent frameworks (MAGE and VerilogCoder, controlled to use GPT-5.2) generally outperform both standalone inference and single-agent feedback, highlighting the value of curated tool use and role specialization.VERI-SURE achieves the best overall functional Pass@1 (93.30%) with perfect syntax, and the largest advantage on Hard tasks (85.07% functional). It also boosts the open-weight MoE backbone DeepSeek-3.2 to 71.29% overall functional Pass@1 (+10.05 pp over standalone; +7.18 pp over single-agent feedback).

5.3 Ablation Results

Table 2 isolates the contribution of each major VERI-SURE component. Overall, VERI-SURE achieves 93.30% functional pass rate, improving substantially over GPT-5.2 standalone (76.56%) and the simulator-feedback loop (77.99%), especially on hard problems (85.07% vs. 59.70%). Results also indicate that feedback alone is insufficient for hard RTL tasks, and that most gains come from structured agentic verification and repair.

Generally, removing any major component of VERI-SURE degrades performance. Disabling tracing, slicing, and patching together causes the largest drop, while the finer-grained variants show that all three pieces contribute individually: tracing identifies when and which signals fail, slicing maps those signals to suspect RTL blocks, and patching constrains edits to those blocks. Patching also keeps repairs small: a patch modifies 7.9 lines on average, versus 68.8 for whole-file regeneration (Appendix C.4). For Hard problems, Formal Verification is also important: removing it reduces pass rate from 85.07% to 73.13% (-11.9 pp), indicating that proof-based feedback could help debug deep logical bugs that may not surface under finite simulation.

5.4 Case Study

In Figure 5, we use two cases to illustrate how VERI-SURE converts failing executions into formal-hint-guided repairs. For an 8×8 two's-complement multiplier (top), the Boolean Proofer derives the contract-level Boolean model and proves inequivalence against the generated RTL via a SymbiYosys miter, pinpointing the missing sign-bit contribution of $b[7]$ and enabling a minimal patch to the partial-product logic. For a control-heavy history-register block (bottom), the Asserter injects a clock/reset assertion that triggers at the first violating cycle and reports the implicated signals/values,

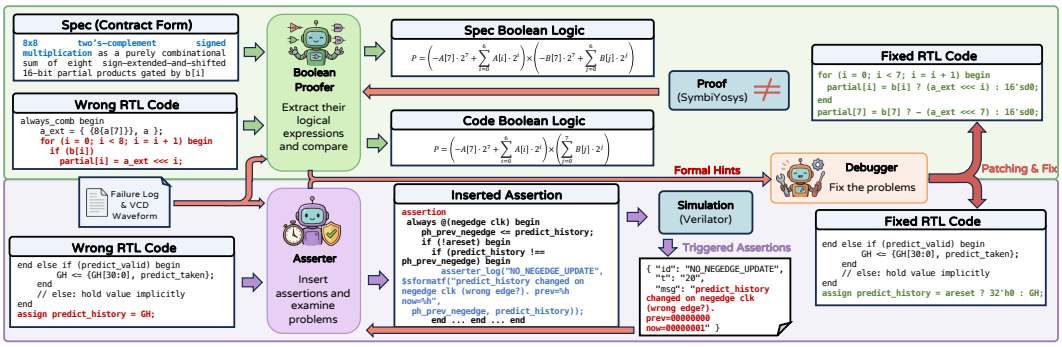


Figure 5: Case study: formal-hint-guided debugging in VERI-SURE. Boolean Proofer and Asserter agents help generate correct RTL code.

guiding the Debugger to fix the edge/reset handling without regenerating the whole module. More detailed success cases can be found in Appendix D and we additionally analyze the remaining failures in Section 5.5 and Appendix C.1.

5.5 Failure Analysis

Under-Specified Descriptions. Among the 14 tasks VERI-SURE fails to solve, one is a tooling artifact, and 10 of the remaining 13 true failures are timing-related, such as FSM transitions with ambiguous timing or misaligned output pulses (Appendix C.1). In most of these cases, VERI-SURE locates the faulty logic correctly, but the repair budget runs out before a correct fix is found, usually because the description does not fully determine the intended behavior (e.g., `Prob117` and `Prob145`) or the agents fail to infer it. Since patches that do not improve the failure signature are rolled back, extra iterations can never make a design worse.

Contract Fidelity. We audited all 209 contracts generated by the Architect and found errors in 12 of them (5.7%), mostly functional contradictions and a few hallucinated constraints, with no interface errors (Appendix C.2). Among the 12 tasks, only 3 fail and they still fail without the contract. The other 9 are solved because the Debugger relies on trace reports from simulating the golden testbench, not on the contract. Contracts are also least precise on hard arithmetic, such as floating-point rounding and subnormal handling, which accounts for the remaining 3 (arithmetic) true failures.

Reliability of Formal Hints. Since the assertions and Boolean reference models are generated from the LLM-derived contract, they inherit its errors and can also fail on their own. In a full-benchmark run, the Boolean Proofer aborted in 4 of its 30 invocations (SMT solver timeout, unsupported constructs, or an UNKNOWN solver result), and generated assertions can miss the temporal condition behind a failure. Therefore, both branches only provide hints to the Debugger, and an acceptance always requires passing the testbench.

6 Conclusion

In this work, we present VERI-SURE, a contract-driven multi-agent framework that closes the RTL development loop by aligning generation with verification and localized repair. By distilling natural-language intent into a shared design contract, VERI-SURE mitigates semantic drift across agents, while its temporal tracing and static slicing mechanisms enable precise patching that avoids whole-file rewrites and reduces regression risk. Beyond sparse simulation-centric feedback, we integrate formal verification to achieve better debugging capability. To measure the progress on realistic RTL tasks, we also introduce VERILOGEVAL-V2-EXT, including 53 more industrial-grade problems and difficulty labels on all tasks. Experiments show that VERI-SURE achieves state-of-the-art verified-correct performance, reaching 93.30% overall functional pass rate and ablations confirm that the gains come primarily from our debugging pipeline. Overall, VERI-SURE demonstrates that the proposed methods can effectively aid LLMs in generating RTL code, moving automated design closer toward silicon-grade correctness.

Acknowledgments and Disclosure of Funding

We thank The University of Edinburgh's ELM service and EDINA team for supporting the API access. This work also used the Isambard-AI National AI Research Resource (AIRR), operated by the University of Bristol and funded by the UK Government's Department for Science, Innovation and Technology (DSIT) via UK Research and Innovation, and by the Science and Technology Facilities Council [ST/AIRR/I-A-I/1023].

References

[1] Hammad Ahmad, Yu Huang, and Westley Weimer. CirFix: Automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 990–1003. ACM, 2022. doi: 10.1145/3503222.3507763.

[2] Mohammad Akyash and Hadi Mardani Kamali. SimEval: Investigating the similarity obstacle in LLM-based hardware code generation. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pages 1002–1007. Association for Computing Machinery, 2025. doi: 10.1145/3658617.3697624.

[3] Anthropic. Introducing Claude Sonnet 4.5, September 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. doi: 10.1007/3-540-49059-0_14.

[5] Daniel Brand. Verification of large synthesized designs. In *Proceedings of the 1993 International Conference on Computer Aided Design (ICCAD)*, pages 534–537. IEEE Computer Society Press, 1993. doi: 10.1109/ICCAD.1993.580110.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[7] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y. Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou,

Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. DeepSeek-V3.2: Pushing the frontier of open large language models, December 2025. URL https://arxiv.org/abs/2512.02556. arXiv:2512.02556.

[8] Harry Foster, Adam Krolnik, and David Lacey. *Assertion-Based Design*. Springer, 2003. doi: 10.1007/978-1-4419-9228-4.

[9] 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)*, pages 162–169. IEEE, 2024. doi: 10.1109/ICCD63220.2024.00033.

[10] GLM-4.5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models, August 2025. URL https://arxiv.org/abs/2508.06471. arXiv:2508.06471.

[11] Google DeepMind. A new era of intelligence with Gemini 3, November 2025. URL https://blog.google/products-and-platforms/products/gemini/gemini-3/.

[12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, January 2024. URL https://arxiv.org/abs/2401.14196. arXiv:2401.14196.

[13] 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, pages 300–307, 2025. doi: 10.1609/aaai.v39i1.32007.

[14] Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, Xinwei Fang, Xi Wang, Nan Guan, and Zhe Jiang. UVLLM: An automated universal RTL verification framework using LLMs. In *62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE, 2025. doi: 10.1109/DAC63849.2025.11435108.

[15] Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Xianzhen Luo, Qiufeng Wang, YuanTao Fan, Qingfu Zhu, Zhaoxiang Zhang, Yang Gao, Jie Fu, Qian Liu, Houyi Li, Ge Zhang, Yuan Qi, Yinghui Xu, Wei Chu, and Zili Wang. OpenCoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1591.

[16] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026. doi: 10.1145/3747588.

[17] Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. RTL-Repair: Fast symbolic repair of hardware design code. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*, pages 867–881. ACM, 2024. doi: 10.1145/3620666.3651346.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38 (1):54–72, 2012. doi: 10.1109/TSE.2011.104.

[19] Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, Alexandre Sablayrolles, Amélie Héliou, Amos You, Andy Ehrenberg, Andy Lo, Anton Eliseev, Antonia Calvi, Avinash Sooriyarachchi, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Clémence Lanfranchi, Corentin Barreau, Cyprien Courtot, Daniele Grattarola, Darius Dabert, Diego de las Casas, Elliot Chane-Sane, Faruk Ahmed, Gabrielle Berrada, Gaëtan Ecrepont, Gauthier Guinet, Georgii Novikov, Guillaume Kunsch, Guillaume Lample, Guillaume Martin, Gunshi Gupta, Jan Ludziejewski, Jason Rute, Joachim Studnia, Jonas Amar, Joséphine Delas, Josselin Somerville Roberts, Karmesh Yadav, Khyathi Chandu, Kush Jain, Laurence Aitchison, Laurent Fainsin, Léonard Blier, Lingxiao Zhao, Louis Martin, Lucile Saulnier, Luyu Gao, Maarten Buyl, Margaret Jennings, Marie Pellat, Mark Prins, Mathieu Poirée, Mathilde Guillaumin, Matthieu Dinot, Matthieu Futeral, Maxime Darrin, Maximilian Augustin, Mia Chiquier, Michel Schimpf, Nathan Grinsztajn, Neha Gupta, Nikhil Raghuraman, Olivier Bousquet, Olivier Duchenne, Patricia Wang, Patrick von Platen, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Quentin Torroba, Romain Sauvestre, Roman Soletskyi, Rupert Menneer, Sagar Vaze, Samuel Barry, Sanchit Gandhi, Siddhant Waghjale, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Teven Le Scao, Théo Cachet, Theo Simon Sorg, Thibaut Lavril, Thiziri Nait Saada, Thomas Chabal, Thomas Foubert, Thomas Robert, Thomas Wang, Tim Lawson, Tom Bewley, Tom Bewley, Tom Edwards, Umar Jamil, Umberto Tomasini, Valeriia Nemychnikova, Van Phung, Vincent Maladière, Virgile Richard, Wassim Bouaziz, Wen-Ding Li, William Marshall, Xinghui Li, Xinyu Yang, Yassine El Ouahidi, Yihan Wang, Yunhao Tang, and Zaccharie Ramzi. Ministral 3, January 2026. URL https://arxiv.org/abs/2601.08584. arXiv:2601.08584.

[20] Jiale Liu and Nanzhe Wang. AutoSurrogate: An LLM-driven multi-agent framework for autonomous construction of deep learning surrogate models in subsurface flow. *Advanced Engineering Informatics*, 76:105058, 2026. doi: 10.1016/j.aei.2026.105058.

[21] Jiale Liu, Dandan Peng, Huan Wang, Chenyu Liu, Yan-Fu Li, and Min Xie. AeroGPT: Leveraging large-scale audio model for aero-engine bearing fault diagnosis. *IEEE Transactions on Cybernetics*, pages 1–14, 2026. doi: 10.1109/TCYB.2026.3668256.

[22] Jiale Liu, Huan Wang, Yue Zhang, Xiaoyu Luo, Jiaxiang Hu, Zhiliang Liu, and Min Xie. InsightX Agent: An LMM-based agentic framework with integrated tools for reliable X-ray

NDT analysis. *IEEE Transactions on Reliability*, 75:1455–1468, 2026. doi: 10.1109/TR.2026.3668946.

[23] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. ChipNeMo: Domain-adapted LLMs for chip design. *arXiv preprint arXiv:2311.00176*, 2023.

[24] 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, 2025. doi: 10.1109/TCAD.2024.3483089.

[25] Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. DeepRTL: Bridging Verilog understanding and generation with a unified representation model. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[26] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design RTL generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE, 2024. doi: 10.1109/ASP-DAC58780.2024.10473904.

[27] Meta AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation, April 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[28] Kyungjun Min, Seonghyeon Park, Hyeonwoo Park, Jinoh Cho, and Seokhyeong Kang. Improving LLM-based Verilog code generation with data augmentation and RL. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE, 2025. doi: 10.23919/DATE64628.2025.10992897.

[29] Kyungjun Min, Kyumin Cho, Junhwan Jang, and Seokhyeong Kang. REvolution: An evolutionary framework for RTL generation driven by large language models. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 282–288. IEEE, 2026. doi: 10.1109/ASP-DAC66049.2026.11420420.

[30] Mistral AI. Introducing: Devstral 2 and Mistral Vibe CLI, December 2025. URL https://mistral.ai/news/devstral-2-vibe-cli.

[31] Mistral AI. Mistral Medium 3.1. Model card (`mistral-medium-2508`), August 2025. URL https://docs.mistral.ai/models/model-cards/mistral-medium-3-1-25-08.

[32] OpenAI. Introducing GPT-5.2, December 2025. URL https://openai.com/index/introducing-gpt-5-2/.

[33] 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*, pages 40145–40153. PMLR, 2024.

[34] 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, 2025. doi: 10.1145/3718088.

[35] Qwen Team. Qwen3-Coder: Agentic coding in the world, July 2025. URL https://qwen.ai/blog?id=qwen3-coder.

[36] Qwen Team. Qwen3-Max: Just scale it, September 2025. URL https://qwen.ai/blog?id=qwen3-max.

[37] Wilson Snyder, Paul Wasson, Duane Galbi, Geza Lore, et al. Verilator, 2026. URL https://verilator.org.

[38] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. AutoChip: Automating HDL generation using LLM feedback. *arXiv preprint arXiv:2311.04887*, 2023.

[39] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. VeriGen: A large language model for Verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024. doi: 10.1145/3643681.

[40] YunDa Tsai, Mingjie Liu, and Haoxing Ren. RTLFixer: Automatically fixing RTL syntax errors with large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024. doi: 10.1145/3649329.3657353.

[41] Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Zhe Jiang, and Nan Guan. Large language model for Verilog generation with code-structure-guided reinforcement learning. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 164–170. IEEE, 2025. doi: 10.1109/ICLAD65226.2025.00025.

[42] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. doi: 10.1109/TSE.1984.5010248.

[43] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. ChatEDA: A large language model powered autonomous agent for EDA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(10):3184–3197, 2024. doi: 10.1109/TCAD.2024.3383347.

[44] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. MEIC: Re-thinking RTL debug automation using LLMs. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9. ACM, 2024. doi: 10.1145/3676536.3676801.

[45] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pages 614–621, 2025. doi: 10.1145/3658617.3697756.

[46] Bingkun Yao, Ning Wang, Jie Zhou, Xi Wang, Hong Gao, Zhe Jiang, and Nan Guan. Location is key: Leveraging LLM for functional bug localization in Verilog design. In *62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE, 2025. doi: 10.1109/DAC63849.2025.11133280.

[47] YosysHQ. SymbiYosys (sby): Front-end for Yosys-based formal verification flows, 2026. URL https://github.com/YosysHQ/sby.

[48] Yang Zhang, Rui Zhang, Jiaming Guo, Lei Huang, Di Huang, Yunpu Zhao, Shuyao Cheng, Pengwei Jin, Chongxiao Li, Zidong Du, Xing Hu, Qi Guo, and Yunji Chen. QiMeng-SALV: Signal-aware learning for Verilog code generation. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-5335.

[49] 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)*, pages 1–7. IEEE, 2025. doi: 10.1109/DAC63849.2025.11133191.

[50] Zhipu AI. GLM-4.7: Advancing the coding capability, December 2025. URL https://z.ai/blog/glm-4.7.

[51] 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, 2025. doi: 10.52202/085713-5160.

Appendices

We supplement some additional technical content here in the appendices. Specifically, Appendix A presents additional details of the VERILOGEVAL-V2-EXT benchmark; Appendix B documents implementation details of the VERI-SURE framework; Appendix C provides failure, design-contract, formal-signal reliability, repairing mechanism, and cost analyses; Appendix D reports three successful case studies; and Appendix E offers a detailed comparison with other related frameworks.

A The VERILOGEVAL-V2-EXT Benchmark

This appendix provides additional details for VERILOGEVAL-V2-EXT, including (i) a full taxonomy and coverage analysis of the original VerilogEval-v2 tasks, (ii) the motivations and design targets of our 53 newly added industrial-grade tasks, and (iii) the rule-based difficulty grading scheme used in the paper.

A.1 Summary of Existing Problems

The original VerilogEval-v2 benchmark contains 156 RTL code generation tasks. While it offers strong coverage of basic digital logic problems, we find it under-represents the types of modules that dominate real RTL development, such as protocol controllers, buffering, and multi-cycle datapaths. As a result, models can achieve high scores by mastering local syntax patterns and short combinational reasoning, while still failing on engineering-critical behaviors like latency alignment, corner-case handling, and stateful control logic. To make this limitation explicit and to identify concrete coverage gaps, we manually reviewed all 156 problems and grouped them into nine categories, summarized in Table 3.

Taxonomy Methodology. Each problem is assigned to the category that most strongly determines its implementation structure (e.g., whether the solution is dominated by combinational datapath wiring, sequential state machines, or algorithmic transformations). When a problem touches multiple concepts, we classify it by the dominant design pattern required to pass the golden testbench.

Key Observations and Limitations. This taxonomy highlights why simulation-only pass rates on VerilogEval-v2 can overestimate progress toward silicon-grade RTL code generation:

- **Over-emphasis on short, textbook patterns:** A large fraction of tasks are small-scale logic exercises (Categories I–III) or repetitive FSM variants (Category VIII).
- **Severe under-coverage of industrial datapaths:** "Basic Arithmetic" comprises only 9/156 tasks, largely limited to simple integer operations.
- **Missing protocol and buffering semantics:** The original set does not systematically test standard communication protocols (UART/SPI/I^2C) or ubiquitous flow-control primitives (FIFOs, skid buffers).
- **Limited exposure to CDC and multi-clock assumptions:** Most tasks assume a single clock domain, ignoring reset-domain reasoning and CDC safety.

A.2 New Problems

To address the coverage gaps above, we introduce VERILOGEVAL-V2-EXT by adding 53 new industrial-grade tasks, bringing the total to 209 problems. The new problems are designed to be industry-aligned, specifically targeting multi-cycle behaviors and synthesizable constructs. We aim to cover the missing areas detailed in Table 4.

A.3 Grading Scheme

To enable a more fine-grained analysis, we stratify the 209 problems into Easy/Medium/Hard using a rule-based structural complexity score computed from the canonical RTL implementation. The complexity of hardware description language (HDL) code can be systematically decomposed into three orthogonal dimensions: (i) structural complexity, which is measured by the number of lines of

Table 3: Full taxonomy of the original VerilogEval-v2 benchmark. We group problems into nine categories to reveal coverage biases and industrial blind spots.

Category	#	Problem IDs	Description & Limitations
I. Basic Logic Gates	16	Prob001, Prob005, Prob007, Prob011, Prob012, Prob013, Prob014, Prob019, Prob026, Prob043, Prob051, Prob052, Prob059, Prob077, Prob087, Prob092	Gate-level/bit-level Boolean logic. Limited depth and limited temporal reasoning.
II. Vectors & Bit Manipulation	14	Prob004, Prob006, Prob015, Prob023, Prob025, Prob032, Prob042, Prob044, Prob055, Prob062, Prob064, Prob070, Prob094, Prob097	Concatenation, slicing, bit-reversal, and parity. Does not cover real numeric formats (fixed-/floating-point).
III. Mux & Decoding	11	Prob017, Prob018, Prob021, Prob022, Prob039, Prob071, Prob076, Prob093, Prob104, Prob112, Prob114	Standard mux/decoder/encoder usage. Mostly textbook patterns; little parameterization.
IV. Basic Arithmetic	9	Prob009, Prob016, Prob024, Prob027, Prob030, Prob033, Prob065, Prob081, Prob123	Mostly add/sub/popcount. Notably lacks multipliers/dividers, saturation, and complex datapaths.
V. Latches & Flip-Flops	14	Prob002, Prob003, Prob008, Prob028, Prob029, Prob031, Prob034, Prob041, Prob046, Prob047, Prob048, Prob049, Prob053, Prob073	Basic sequential primitives (DFF/latch). Almost entirely single-clock; does not test CDC patterns.
VI. Counters, Timers & Edge Det.	15	Prob035, Prob037, Prob038, Prob040, Prob045, Prob054, Prob066, Prob067, Prob068, Prob075, Prob078, Prob080, Prob141, Prob151, Prob156	Counting/timing logic. Typically linear control, limited handshake or flow-control complexity.
VII. Shift Registers	12	Prob060, Prob061, Prob063, Prob082, Prob084, Prob085, Prob086, Prob095, Prob096, Prob105, Prob115, Prob118	Shifting/LFSR/pattern detection. Does not include streaming buffers (FIFO) or backpressure.
VIII. Finite State Machines (FSMs)	42	Prob036, Prob056, Prob072, Prob074, Prob079, Prob088, Prob089, Prob091, Prob099, Prob100, Prob106, Prob107, Prob108, Prob109, Prob110, Prob111, Prob119, Prob120, Prob121, Prob124, Prob127, Prob128, Prob129, Prob133, Prob134, Prob135, Prob136, Prob137, Prob138, Prob139, Prob140, Prob142, Prob143, Prob144, Prob146, Prob147, Prob148, Prob149, Prob150, Prob152, Prob154, Prob155	Largest category. Many are variants of tutorial sequence detectors; few reflect on-chip protocols.
IX. Interpretation & Algorithms	23	Prob010, Prob020, Prob050, Prob057, Prob058, Prob069, Prob083, Prob090, Prob098, Prob101, Prob102, Prob103, Prob113, Prob116, Prob117, Prob122, Prob125, Prob126, Prob130, Prob131, Prob132, Prob145, Prob153	Algorithmic "translation" tasks (e.g., Game of Life, waveform interpretation). Less representative of microarchitecture.

Table 4: Categories of the 53 new tasks in VERILOGEVAL-V2-EXT. The categories are chosen to reflect common IP/SoC RTL development patterns that are sparse or absent in the original VerilogEval-v2 benchmark.

Category	Motivation	Representative RTL requirements
Advanced Arithmetic & Math	Modern AI/DSP chips depend on rich numeric kernels; original is dominated by add/sub tasks.	Multi-operator datapaths, corner-case handling (overflow/saturation), and defined latency.
Standard Communication Protocols	Real chips communicate via standardized serial protocols; absent in original set.	Framing/encoding, CRC/parity, start/stop conditions, and valid-ready handshakes.
Memory, CDC & Flow Control	Buffering is ubiquitous; single-clock primitives are insufficient for system robustness.	FIFO logic, full/empty semantics, backpressure, and safe control signaling.
DSP & Image Processing	Accelerators frequently embed DSP kernels; original tasks ignore structured data flow.	Windowed/streaming computation, pipeline alignment, and boundary handling.
Advanced Timing & System Control	Timing/latency errors are common tape-out bugs; benchmarks need multi-cycle complexity.	Multi-stage pipelines, coordinated enables, complex FSM + datapath interaction.

code (LOC) to reflect the scale and modularity of the design; (ii) data complexity, usually reflected by the bit width for processing data to indicate numerical precision and storage requirements; and (iii) temporal complexity, which is characterized by the diversity of temporal structures and nesting depth to represent the complexity of temporal logic such as clock domains, state machines, and pipelines.

These three dimensions jointly determine the difficulty of understanding, implementing, and verifying the design. During the construction process, these three dimensions are scored, with structural complexity mapped to the macroscopic indicator of lines of code (LOC); data complexity is mainly represented by the maximum bit width (Max Width); and temporal complexity is jointly characterized by the complexity of assignment statements (#assign), the number of blocks (#always), and conditional branches (#case) among other syntactic elements.

Therefore, the complexity score S is the sum of sub-scores derived from five metrics:

$$S = s_{\text{loc}} + s_{\text{assign}} + s_{\text{always}} + s_{\text{case}} + s_{\text{width}} \tag{5}$$

The scoring rules are detailed in Table 5. We map the total score S to difficulty labels:

Easy : $S \leq 1$ **Medium** : $2 \leq S \leq 3$ **Hard** : $S \geq 4$.

Table 5: Scoring rules based on the structural complexity of designs.

Metric	Thresholds	Points
LOC	≤ 10 / 11–30 / 31–60 / > 60	0 / 1 / 2 / 3
#Assign	≤ 1 / 2–4 / ≥ 5	0 / 1 / 2
#Always	0 / 1 / 2 / ≥ 3	0 / 1 / 2 / 3
#Case	0 / 1 / 2 / ≥ 3	0 / 1 / 2 / 3
Max Width	≤ 32 / 33–128 / > 128	0 / 1 / 2

Task Lists. Applying this grading scheme, we obtain an initial difficulty label for each problem. We then manually reviewed the resulting split to ensure the final annotations are consistent with human-perceived implementation and verification difficulty. The final difficulty lists are:

Easy (51 tasks)

Prob001, Prob002, Prob003, Prob004, Prob005, Prob006, Prob007, Prob008, Prob009, Prob010, Prob011, Prob012, Prob013, Prob014, Prob015, Prob016, Prob019, Prob020, Prob022, Prob023, Prob024, Prob025, Prob027, Prob028, Prob029, Prob031, Prob034, Prob042, Prob043, Prob050, Prob055, Prob056, Prob057, Prob062, Prob083, Prob090, Prob098, Prob101, Prob102, Prob103, Prob131, Prob169, Prob188, Prob191, Prob194, Prob195, Prob197, Prob201, Prob202, Prob203, Prob207.

Medium (91 tasks)

Prob017, Prob018, Prob021, Prob026, Prob030, Prob032, Prob033, Prob035, Prob036, Prob037, Prob038, Prob039, Prob040, Prob041, Prob044, Prob045, Prob046, Prob047, Prob048, Prob049, Prob051, Prob052, Prob053, Prob054, Prob059, Prob060, Prob061, Prob063, Prob064, Prob065, Prob066, Prob067, Prob068, Prob069, Prob070, Prob071, Prob072, Prob073, Prob074, Prob075, Prob076, Prob077, Prob080, Prob081, Prob082, Prob084, Prob085, Prob091, Prob092, Prob093, Prob097, Prob099, Prob100, Prob104, Prob105, Prob106, Prob112, Prob113, Prob114, Prob116, Prob117, Prob118, Prob122, Prob123, Prob125, Prob126, Prob130, Prob132, Prob135, Prob141, Prob145, Prob163, Prob165, Prob168, Prob170, Prob171, Prob176, Prob178, Prob182, Prob183, Prob184, Prob186, Prob190, Prob193, Prob196, Prob198, Prob199, Prob200, Prob205, Prob208, Prob209.

Hard (67 tasks)

Prob058, Prob078, Prob079, Prob086, Prob087, Prob088, Prob089, Prob094, Prob095, Prob096, Prob107, Prob108, Prob109, Prob110, Prob111, Prob115, Prob119, Prob120, Prob121, Prob124, Prob127, Prob128, Prob129, Prob133, Prob134, Prob136, Prob137, Prob138, Prob139, Prob140, Prob142, Prob143, Prob144, Prob146, Prob147, Prob148, Prob149, Prob150, Prob151, Prob152, Prob153, Prob154, Prob155, Prob156, Prob157, Prob158, Prob159, Prob160, Prob161, Prob162, Prob164, Prob166, Prob167, Prob172, Prob173, Prob174, Prob175, Prob177, Prob179, Prob180, Prob181, Prob185, Prob187, Prob189, Prob192, Prob204, Prob206.

A.4 Issues with Prob099

During the construction and revision of our VERILOGEVAL-V2-EXT Benchmark, we found an issue with the original Prob099 design from VerilogEval-v2 under our Verilator environment. Specifically, when we attempted to compile and run the Prob099 harness under Verilator, the build failed during elaboration with fatal named-port connection errors, so the simulation never started and no functional comparison could be performed. In the Verilator log, the failure presents as repeated `PINNOTFOUND` diagnostics indicating that the testbench tries to connect ports `Y2` and `Y4` on both the reference instance and the DUT instance, even though those formal port names do not exist on the corresponding module definitions. A representative excerpt is:

```
%Error-PINNOTFOUND: ...Prob099_m2014_q6c_test.sv:74:4: Pin not found: 'Y2'
%Error-PINNOTFOUND: ...Prob099_m2014_q6c_test.sv:75:4: Pin not found: 'Y4'
%Error-PINNOTFOUND: ...Prob099_m2014_q6c_test.sv:80:4: Pin not found: 'Y2'
%Error-PINNOTFOUND: ...Prob099_m2014_q6c_test.sv:81:4: Pin not found: 'Y4'
%Error: Exiting due to 4 error(s)
```

The root cause is an interface mismatch between the testbench and the provided reference/DUT module definitions. The reference module `RefModule` (and the auto-generated `TopModule` wrapper used for validation) exports outputs named `Y1` and `Y3`, but the testbench file instantiates both modules as if they exported `Y2` and `Y4`. Verilator resolves named port connections strictly: when it sees an instantiation such as `.Y2(Y2_ref)`, it searches the callee's port list for a formal named `Y2`; if no such formal exists, it raises `PINNOTFOUND` and terminates. Conceptually, this is consistent with a naming/offset confusion in the problem statement text: the prompt mentions "`Y2` and `Y4` corresponding to `y[1]` and `y[3]`," while the actual reference implementation uses the convention "`Y1` corresponds to the next-state signal for `y[1]` and `Y3` corresponds to the next-state signal for `y[3]`." In other words, the authoritative artifacts (reference module + wrapper) define `Y1`/`Y3`, but the testbench was written against the inconsistent `Y2`/`Y4` naming.

To fix the issue, we made the testbench consistent with the reference/DUT interface and with the benchmark's intended observation points. We renamed the internal signals and updated the named port connections in both instantiations accordingly. We also aligned the testbench's declaration of the state vector to the canonical module interface by changing it from a 1-based range (`logic [6:1] y`) to the conventional 0-based range (`logic [5:0] y`) in both the stimulus generator and the top-level testbench. After these changes, Verilator is able to compile the testbench successfully and execute the simulation, and the DUT (when wrapped to mirror the reference) produces matching `Y1` and `Y3` outputs over the randomized one-hot stimulus set.

B Additional Implementation Details

This appendix documents additional implementation-level details of VERI-SURE that are omitted from the main paper for clarity, including the design contract schema, linting rules, static dependency graph construction for localization, trace alignment/slicing for temporal debugging, and formal miter generation for Boolean equivalence checking.

B.1 Contract Schema

As described in Section 4, each benchmark task provides a fixed module interface and an executable testbench. Nevertheless, the intent behind cycle-accurate behaviors (reset conventions, sampling edge, output latency, corner cases) is frequently under-specified in natural language and can drift across iterations or agent handoffs. VERI-SURE addresses this by introducing a design contract as a structured intermediate representation (IR) shared across agents.

Core Fields. In our implementation, the contract is a JSON object with five core top-level sections:

- `module_name`: (`string`) The top-level DUT module identifier.
- `io`: (`list`) An ordered list of port objects. Each port entry contains:

  - `name`: (`string`) Signal identifier (SystemVerilog identifier constraints).
  - `dir`: (`string`) One of {`input`, `output`, `inout`}.
  - `width`: (`int`) Positive integer bit-width (scalars use `1`).
  - `description`: (`string`) Natural-language semantic summary.
- `clocking`: (`object`) Global clock/reset semantics:
  - `is_sequential`: (`bool`) Whether the design holds clocked state. For purely combinational designs it is `false` and the clock/reset entries are omitted.
  - `clock.name`, `clock.edge`: Clock input port and its active edge (`posedge` or `negedge`); required when `is_sequential` is `true`.
  - `reset.name`, `reset.active`, `reset.type`: Reset input port, its polarity (`high` or `low`), and whether it is `synchronous` or `asynchronous`; omitted when the design has no reset.
- `timing`: (`object`) A mapping from each output to its expected latency, in cycles, relative to contract-defined sampling:
  - `timing.outputs[signal].latency_cycles`: (`int`) non-negative; `0` denotes combinational-by-contract.
- `functional_summary`: (`object`) A compact functional description used to ground verification, including Boolean/combinational obligations:
  - Typically contains a high-level `overview` plus a machine-readable list of `rules` describing input/output relations and corner cases.

Optional Fields. To support richer tasks, the implementation also allows optional keys (ignored if absent), such as `parameters` (typed module parameters) and `test_plan` (directed stimulus/check intent).

Schema Usage. The same contract instance is injected into: (i) the **Coder** prompt (to enforce interface/timing consistency), (ii) the **Verifier** prompt (to generate a self-checking testbench consistent with the contract), (iii) the **Debugger** prompt (as the reference that repairs must preserve, alongside the trace report), (iv) the **Asserter** prompt (to generate temporal assertions for the clock/reset/latency obligations), and (v) the **Boolean Proofer** (to extract combinational obligations for formal equivalence when applicable).

B.2 Contract Linting

To prevent malformed or hallucinated contracts from propagating into RTL code generation and verification, VERI-SURE applies a straightforward linter to every Architect-produced JSON before invoking downstream agents.

Linting Objectives. The linter reports each issue as an *error* or a *warning* and checks three classes of constraints:

1. **Schema validity (structure and typing):** The contract must be a JSON object with a non-empty `module_name` and a non-empty `io` list of port objects. The linter validates types (e.g., `width` is a positive integer, defaulting to `1` when omitted, and `latency_cycles` is a non-negative integer) and rejects invalid enumerations (e.g., `dir` not in {`input`,`output`,`inout`}).
2. **Signal consistency and referential integrity:** The linter ensures that port names are non-empty and unique, and that every output has a `timing` entry with a `latency_cycles` value. A missing entry is reported as a warning, since the latency of that output becomes ambiguous.
3. **Clock/reset integrity:** If `clocking.is_sequential` is `true`, a missing `clocking.clock.name` is an error, and a clock or reset name that is not an input port is reported as a warning. Combinational designs and designs without a reset are therefore valid contracts.

The linter also warns when the per-agent `guidance` block is missing. If the lint report contains errors, the Architect is re-invoked with the report to fix the contract, for up to two revision rounds before generation proceeds. Warnings are recorded in the lint report but do not trigger a revision.

B.3 Static Dependency Graph Construction

VERI-SURE localizes repairs using a static dependency slice over the RTL source (Section 3.3). This slice is computed from a dependency graph built directly from the SystemVerilog AST, rather than fragile regex matching.

Parsing Frontend. We use `tree-sitter-verilog` to parse the generated RTL into an AST and extract semantic units. This provides stable node spans (start/end line ranges) required by our block-level patching mechanism.

Block Granularity. The RTL is decomposed into atomic `RtlBlock` units. A block is any semantic unit that can drive signals:

- **Procedural blocks:** `always`/`always_comb`/`always_ff`/`always_latch`.
- **Continuous assignments:** `assign` statements.
- **Submodule instances:** treated as blocks to account for hierarchical wiring when present.

Each block stores a unique identifier, its source span (line range), its raw text, and statically extracted read/write sets.

Read/Write Extraction. For each block B, we compute:

- $W(B)$ (**write set**): all L-values written by the block (e.g., left-hand sides of `=` or `<=`, including bit/part selects mapped to their base identifiers).
- $R(B)$ (**read set**): all identifiers used as R-values in RHS expressions and control predicates (e.g., `if` conditions, `case` selectors), as well as explicit sensitivity list identifiers when present.

For module instances, $W(B)$ and $R(B)$ are inferred from the port connections when directionality is unambiguous; otherwise, the instance is conservatively treated as reading all connected signals, and writing those connected to nets whose names match the instance's output connections in the AST.

Driver Map and Dependency Relation. We build a global driver map:

$$\mathcal{D}(s) \;=\; \{\, B \mid s \in W(B) \,\}, \tag{6}$$

which maps each signal s to the set of blocks that may drive it. A directed dependency edge exists from B_i to B_j iff B_j reads a signal written by B_i:

$$B_j \leftarrow B_i \quad \Leftrightarrow \quad W(B_i) \cap R(B_j) \neq \varnothing. \tag{7}$$

Backward Dependency Slicing. Given a failing output set $\mathcal{Y}_{\text{fail}}$ identified by simulation/trace analysis, we compute the suspect block set $\mathcal{B}_{\text{sus}}$ using a bounded backward BFS:

$$\mathcal{B}_0 = \bigcup_{y \in \mathcal{Y}_{\text{fail}}} \mathcal{D}(y), \tag{8}$$

$$\mathcal{B}_{d+1} = \mathcal{B}_d \;\cup \bigcup_{B \in \mathcal{B}_d} \bigcup_{r \in R(B)} \mathcal{D}(r), \tag{9}$$

and finally $\mathcal{B}_{\text{sus}} = \mathcal{B}_{d_{\max}}$. We use a small fixed depth bound $d_{\max}$=2, so the slice spans at most three levels of driver blocks, to balance localization precision and completeness: deeper slices increase recall but also inflate the Debugger context window, which can reduce the precision of LLM-guided patches.

The resulting $\mathcal{B}_{\text{sus}}$ is then used to enforce edit locality: the Debugger is only allowed to modify blocks within $\mathcal{B}_{\text{sus}}$, while all other lines are preserved verbatim, reducing regressions and "repair hallucinations".

B.4 Trace Alignment & Slicing

Simulation feedback is often too coarse (pass/fail or a single mismatch) to support reliable multi-cycle repairs. VERI-SURE therefore converts raw simulation artifacts into an LLM-consumable trace report that localizes failures in both time and signal space.

Failure Localization. The Verilator harness captures stdout/stderr and parses for the earliest failure event, including assertion failures emitted by the Asserter or explicit mismatch diagnostics (e.g., "Mismatch at time ..."). This produces the first failing timestamp t_f (or equivalently a failure cycle index under the contract-defined sampling scheme).

Windowed Waveform Extraction. We extract a short window $\mathcal{W}(t_f)$ ending at the failure, consisting of the last L=10 clock edges (posedge and negedge, i.e., five clock cycles) at or before t_f. Using a VCD parser, we sample signal values at every edge in the window, label each sample with its edge type, and record:

1. the expected and observed values of the failing output signals $\mathcal{Y}_{\text{fail}}$,
2. all top-level inputs (for context),
3. up to 20 internal signals identified by the static slice (signals appearing in $R(B)$ or $W(B)$ for $B \in \mathcal{B}_{\text{sus}}$, excluding top-level ports).

Alignment Diagnosis for Off-by-One Timing Bugs. Pipeline and handshake designs frequently fail due to latency mismatches or sampling-edge errors. To provide actionable timing hints, the tracing mechanism performs a lightweight alignment check on each failing output over a longer history than $\mathcal{W}(t_f)$. Let e_k and a_k denote the expected and observed values of the output at the k-th clock edge (posedge or negedge), taken from the last N=200 edges up to t_f. For a shift δ, measured in clock edges, we compute the match rate

$$\text{match}(\delta) \;=\; \frac{1}{|K_\delta|} \sum_{k \in K_\delta} \mathbb{I}[a_k = e_{k-\delta}]\,, \qquad \delta \in \{-1, 0, +1\}, \tag{10}$$

where K_δ is the set of indices k for which both a_k and $e_{k-\delta}$ are available. Under this convention, $\delta = +1$ means that the DUT output lags the expected output by one edge, and $\delta = -1$ means that it leads. We also compute $\text{match}(0)$ separately over posedge and over negedge samples to expose wrong-edge updates. The report includes these rates together with the best-matching shift; for example, a best match at $\delta = +1$ indicates that the output appears one edge late. This signal is complementary to assertion failures generated by the Asserter and is particularly effective for diagnosing systematic latency shifts that may not be obvious from single-cycle mismatches.

Trace Report Format. The final trace report includes: (i) the failure cycle/time, (ii) the set of failing outputs, (iii) the best alignment hint (if any), and (iv) the windowed trace slice. This report is injected into the Debugger prompt together with the localized suspect blocks $\mathcal{B}_{\text{sus}}$, enabling the Debugger to propose minimal edits grounded in concrete temporal evidence.

B.5 Formal Miter Construction

The Boolean Proofer targets purely combinational logic by proving equivalence between (i) the DUT implementation and (ii) a compact specification model synthesized from the contract. This is implemented using an automatically generated miter.

Selecting Targets of Proof. From the contract timing map, we identify candidate combinational outputs:

$$\mathcal{Y}_{\text{comb}} \;=\; \{\, y \in \text{Outputs} \mid \texttt{latency_cycles}(y) = 0 \,\}. \tag{11}$$

To avoid unsound proofs on stateful logic, we further filter $\mathcal{Y}_{\text{comb}}$ by dependency analysis: any output driven by an `always_ff` (or a clocked `always`) in the current RTL is excluded from combinational proving. This conservative rule ensures the miter remains purely combinational.

Model Synthesis and Wrapping. We compile `functional_summary` into a synthesizable SystemVerilog specification module that computes each y_{spec} in $\mathcal{Y}_{\text{comb}}$ from the shared inputs `x`. For simple Boolean expressions, this is emitted as `assign` statements; for piecewise logic, it is emitted as `always_comb` with `if/case`.

We then generate a wrapper that instantiates the DUT and the specification module in parallel, ties their inputs, and asserts equality on selected outputs. A simplified sketch is shown below:

Example

```
module Miter(/* shared inputs */);
  // shared inputs: declared here
  // dut/spec outputs
  logic [W-1:0] y_dut, y_spec;

  dut_top dut (/* .a(a), .b(b), ... */, .y(y_dut));
  SpecModule spec (/* .a(a), .b(b), ... */, .y(y_spec));

  always @* begin
    assert (y_dut == y_spec);
  end
endmodule
```

Running SymbiYosys. The DUT, specification module, and Miter are passed to SymbiYosys [47] using an SMT-based proving flow (Z3 [6] backend in our setup). Since the miter is combinational, the proof reduces to a SAT/SMT validity query over $\forall \mathbf{x}$. If the proof fails, `sby` produces a counterexample assignment $\mathbf{x}^\star$; we parse this model and translate it into a concrete input vector (and, when applicable, a directed stimulus snippet) that is fed back to the Debugger. This provides a precise, minimal witness of the Boolean mismatch and helps disambiguate pure logic faults from timing/latency faults already handled by the trace and assertion branches.

C Additional Experimental Results

C.1 Failure Mode Analysis

We manually inspected the 14 tasks that did not pass after the full VERI-SURE loop. One task was a tooling artifact, leaving 13 true failures. The breakdown in Table 6 shows that the remaining bottleneck is not syntax generation, but long-horizon temporal reasoning and numerically delicate combinational semantics.

Table 6: Failure mode breakdown for non-passing VERI-SURE outputs on VERILOGEVAL-V2-EXT. Percentages are computed over the 13 true failures, excluding one tooling artifact.

Failure type	Count	Typical root cause
Sequential / timing	10 (76.9%)	Wrong FSM transitions, off-by-one latency, reset semantics, wrong-edge updates, or pulse alignment errors. Examples include cases where assertion traces localize a wrong-edge update or a one-cycle pulse shift, but the repair budget is exhausted before a fully correct patch is found.
Hard combinational arithmetic	3 (23.1%)	Numerically delicate arithmetic, especially floating-point normalization, subnormal handling, rounding, or width-sensitive corner cases. These failures usually require more precise numerical specifications than the contract's functional summary provides.
Tooling artifact	1	Non-semantic failure due to checker/tool integration rather than an RTL behavior error; excluded from the true-failure percentages above.

This distribution explains the gap between high syntax pass rates and lower functional pass rates on the hardest tasks. Once syntax and simple combinational logic are solved, the dominant failures arise from temporal corner cases that require consistent reasoning across multiple cycles, or from arithmetic specifications whose corner cases are difficult to infer from natural language alone.

C.2 Design Contract Audit

Since all downstream agents condition on the Architect's contract, we audited whether it faithfully captures the natural-language specification. We manually compared each of the 209 contracts produced in the main GPT-5.2 run against its specification and recorded functional contradictions, constraints not supported by the specification (hallucinations), and interface errors. Table 7 summarizes the results: 12 contracts (5.7%) contain at least one error.

Table 7: Manual audit of the 209 design contracts from the main GPT-5.2 run. The 12 erroneous contracts contain 13 errors in total.

Error type	Count	Typical cases
Functional contradiction	10	K-map derivation, waveform reproduction, and FSM output semantics.
Hallucinated constraint	3	Requirements added by the Architect that the specification does not state.
Interface error	0	–
Erroneous contracts	12 (5.7%)	3 overlap with the 13 true failures in Table 6; the other 9 tasks were solved.

Only 3 of the 12 affected tasks are among the true failures. For the other 9, the Debugger overrides the faulty contract because the trace report is computed from simulation against the golden testbench rather than from the contract. To test whether the 3 overlapping failures are caused by their contract errors, we re-ran them with the contract removed; all 3 still fail, indicating that these tasks are intrinsically hard rather than bottlenecked by the contract. Together with the ablation in Table 2 (w/o Design Contract, -2.9 pp), this suggests that the contract is a net gain and that the trace-driven debugging loop limits the impact of its occasional errors.

C.3 Reliability of LLM-Generated Formal Signals

The Asserter and Boolean Proofer generate formal artifacts from an LLM-derived structured design specification, so VERI-SURE uses them as diagnostic hints for repair and keeps the final acceptance gated by the golden testbench. Erroneous or incomplete formal hints can consume repair budget, but they cannot by themselves certify an incorrect design as correct. We inspected formal-signal behavior on non-passing tasks and summarize the main observations in Table 8.

Table 8: Reliability analysis of LLM-generated formal artifacts. Formal signals improve diagnosis on many tasks, but are not used as acceptance oracles.

Artifact	Useful behavior	Observed limitation
Asserter	Produces concrete timing diagnostics, including wrong-edge updates, reset mismatches, and off-by-one latency violations. These diagnostics often identify the first failing cycle and implicated signals.	Can be incomplete when the generated monitor omits the relevant temporal condition. For example, an SPI release-edge failure can produce simulation mismatches while the generated assertion set reports no trigger.
Boolean Proofer	Provides counterexamples for latency-0 combinational obligations after dependency filtering, helping separate pure Boolean faults from sequential bugs.	Can fail to produce useful guidance when the generated reference model uses unsupported constructs or oversimplifies complex numerical semantics, such as floating-point normalization and subnormal handling.
Acceptance gate	Prevents false acceptance because repaired RTL must still pass the executable golden testbench.	Does not guarantee exhaustive silicon-level correctness; formal artifacts are auxiliary diagnostics rather than complete specifications.

C.4 Repair Statistics

To quantify how much the debugging loop changes a design, we collected repair statistics from the saved results of the main GPT-5.2 run. Of the 209 tasks, 169 (80.9%) pass without any repair. Averaged over all tasks, VERI-SURE applies 1.14 fixes per task, or 5.97 fixes per task that needed repair. A localized patch modifies 7.9 lines (292 characters) on average, whereas regenerating the whole file at every fix rewrites 68.8 lines (1,944 characters). Each patch therefore touches only about 10% of the content a regeneration rewrites, and the rest of the file is preserved verbatim. Locality also improves accuracy: replacing localized patching with whole-file regeneration lowers overall functional Pass@1 from 93.30% to 89.95% (w/o Patching only in Table 2).

C.5 Cost and Token Analysis

Table 9 reports per-task token usage and API cost for GPT-5.2-backed agentic methods on VERILOGEVAL-V2-EXT. All rows use the same backbone to isolate framework-level differences. The cached column reflects prompt-cache discounts for repeated system instructions, tool descriptions, and stable context prefixes. We report both raw and cached costs to separate worst-case API cost from realistic repeated-run deployment cost.

Table 9: Per-task token and API cost analysis with GPT-5.2 as the backbone. Costs are averaged over tasks.

Method	Input tok.	Output tok.	$/task	$/task (cache)
MAGE (10 samples)	16,677	5,508	0.806	0.798
VerilogCoder	25,872	11,718	0.209	0.197
VERI-SURE	62,203	3,653	0.160	0.131

VERI-SURE uses more input tokens because it carries structured diagnostic context, traces, and formal hints across the repair loop. However, localized patching substantially reduces output tokens by avoiding whole-file regeneration. Since output tokens are more expensive than input tokens in the measured API pricing, VERI-SURE is cheaper than other multi-agent frameworks while solving more tasks in the main evaluation.

D Case Studies

D.1 Case Study 1: Boolean Proofer for an 8-bit Signed Multiplier

This case study evaluates the effectiveness of the Boolean Proofer on an 8-bit parallel multiplier (`Prob157`) designed using two's-complement arithmetic. The objective is to implement a purely combinational multiplier that calculates the product of two 8-bit signed integers, a and b, by summing eight sign-extended and shifted 16-bit partial products. This corresponds to the first case in Figure 5.

Problem Description. The initial candidate implementation failed simulation, so the Debugger invoked the Boolean Proofer to check it against the contract-derived reference model. During Boolean equivalence checking, the proofer identified a fundamental mismatch between the implementation logic and the mathematical specification of signed multiplication.

Formal Specification and Counterexample. The Boolean Proofer identified that the implementation treated the multiplier b as an unsigned value, failing to account for the negative weight of the Most Significant Bit (MSB). The logical divergence is summarized below:

- **Erroneous Implementation Logic:**

$$P = \left(-a[7] \cdot 2^7 + \sum_{i=0}^{6} a[i] \cdot 2^i \right) \times \left(\sum_{j=0}^{7} b[j] \cdot 2^j \right)$$

- **Correct Specification Logic:**

$$P = \left(-a[7] \cdot 2^7 + \sum_{i=0}^{6} a[i] \cdot 2^i\right) \times \left(-b[7] \cdot 2^7 + \sum_{j=0}^{6} b[j] \cdot 2^j\right)$$

Root Cause Identification. The error stemmed from a common misconception in signed arithmetic. In a two's-complement multiplier, the MSB of the multiplier, $b[7]$, carries a negative weight (-2^7).

1. **Unsigned Weight Assumption**: The original logic treated all bits of b as positive weights, effectively calculating $A \times B_{\text{unsigned}}$.
2. **Partial Product Sign Error**: For a correct signed product, the 8th partial product must be subtracted (or added as a two's-complement negation) to account for the negative weight:

$$\text{Product} = \sum_{i=0}^{6}(b[i] \cdot (a \ll i)) - (b[7] \cdot (a \ll 7))$$

Code Patching. Following the feedback from the Boolean Proofer, the Verilog implementation was refined. The logic for generating the 8th partial product was adjusted to handle the negative weight of the sign bit as shown in Listing 1.

```
--- buggy_multiplier.sv
+++ corrected_multiplier.sv
@@ -1,8 +1,6 @@
-// BUGGY: All bits treated as positive weights (unsigned multiplication)
-a_ext = { {8{a[7]}}, a };                // Sign-extend a to 16 bits
-for (i = 0; i < 8; i = i + 1) begin      // Loop over all 8 bits
-    if (b[i])                            // If bit is set
-        partial[i] = a_ext <<< i;        // Add positive shifted value
-    else
-        partial[i] = 16'sd0;             // Otherwise add zero
-end
+// CORRECTED: b[7] treated as negative weight (signed multiplication)
+a_ext = { {8{a[7]}}, a };                    // Sign-extend a to 16 bits
+for (i = 0; i < 7; i = i + 1) begin          // Bits 0-6: positive weights
+    partial[i] = b[i] ? (a_ext <<< i) : 16'sd0;
+end
+partial[7] = b[7] ? -(a_ext <<< 7) : 16'sd0;  // Bit 7: negative weight
```

Listing 1: Code correction for Case Study 1.

The patch changes only the eighth partial product, which is now subtracted when $b[7] = 1$. Re-running the Boolean Proofer on the patched RTL found no counterexample against the contract-derived reference model, and the design passed the golden testbench.

D.2 Case Study 2: Assertion Violation in a Global History Register

This case study examines `Prob118`, a 32-bit global branch-history register `GH` with an active-high asynchronous reset `areset`. On a misprediction (`train_mispredicted`), the register is reloaded with `{train_history[30:0], train_taken}`; otherwise, a valid prediction (`predict_valid`) shifts `predict_taken` into it. The register value is exposed through the output `predict_history`. This corresponds to the second case in Figure 5.

Problem Description. The contract specifies a rising-edge clock and an asynchronous active-high reset, so `predict_history` should change only after a rising clock edge or when `areset` is asserted. From these clocking obligations, the Asserter generated a `NO_NEGEDGE_UPDATE` monitor intended to catch updates on the wrong clock edge. Like all Asserter outputs, the monitor is bound to the DUT and evaluated during Verilator simulation rather than by a formal solver.

Generated Assertion and Violation Report. Listing 2 shows the generated monitor. At every falling clock edge outside reset, it compares `predict_history` with the value recorded at the previous falling edge and logs a violation if the two differ.

```
always @(negedge clk) begin
    ph_prev_negedge <= predict_history;
    if (!areset) begin
        if (predict_history !== ph_prev_negedge) begin
            asserter_log("NO_NEGEDGE_UPDATE",
                $sformatf("predict_history changed on negedge clk (wrong
    edge?). prev=%h now=%h",
                          ph_prev_negedge, predict_history));
        end
    end
end
```

Listing 2: Asserter-generated monitor for the negative-edge stability check.

During simulation, the monitor reported its first violation at $t = 20$, where `predict_history` changed from `0x00000000` to `0x00000001`. The report passes the violated property, the failure time, and the implicated signal values to the Debugger as a formal hint.

Root Cause Identification. Starting from `predict_history`, the static slice contains two blocks: the continuous assignment that drives the output and the clocked `always` block that updates `GH`. In the original RTL, the output is a direct copy of the register (`assign predict_history = GH;`), so its reset behavior is only implied by the reset branch of the clocked block. Guided by the Asserter's wrong-edge/reset hint, the Debugger kept the clocked update logic unchanged and targeted the output assignment.

Code Patching. The patch, shown in Listing 3, rewrites only the output assignment so that `predict_history` is forced to zero while `areset` is asserted; the clocked block is preserved verbatim.

```
--- buggy_register.sv
+++ corrected_register.sv
@@ -1,14 +1,14 @@
     always @(posedge clk or posedge areset) begin
         if (areset) begin
             GH <= 32'h0;
         end else begin
             if (train_mispredicted)
                 GH <= {train_history[30:0], train_taken};
             else if (predict_valid)
                 GH <= {GH[30:0], predict_taken};
             // else: hold value implicitly
         end
     end

-    assign predict_history = GH;
+    assign predict_history = areset ? 32'h0 : GH;
 endmodule
```

Listing 3: Code correction for Case Study 2.

The patched module was re-simulated and passed the golden testbench. Because the edit touches a single line, the rest of the design is preserved exactly, which is the behavior the localized patching mechanism is designed to enforce.

D.3 Case Study 3: Automated Repair of Shift Direction Mismatch

This case study illustrates a common logical discrepancy between intended RTL behavior and the actual implementation within a multi-functional sequential circuit in `Prob063`. The target design is a 4-bit register integrating two primary operations: (i) a left-shift with serial input `data`, and (ii) a modulo-16 decrement counter.

Problem Description. A functional mismatch was identified when both control signals, `shift_ena` and `count_ena`, were asserted. Per the specification, `shift_ena` has higher priority; therefore, the register must perform a left shift regardless of `count_ena`.

Trace Analysis & Localization. The failure was captured at simulation timestamp $t = 370$ ns. Inputs are sampled at `negedge clk` and the register updates at the subsequent `posedge clk`. The trace log (summarized in Table 10) records the pre-state $q(t)$ at the sampling edge and the observed/expected next-state values at the following update edge.

As shown in the trace, the design is in state `q(t)=4'b1100` with `shift_ena=1` and `data=0`. Under the required left-shift specification, the next state must be `{q[2:0], data}` = `{100, 0}` = `4'b1000`. However, the DUT produces `4'b0110`, which corresponds to a right-shift style concatenation `{data, q[3:1]}`. The automated debugger localized the failure to the shift transition logic via trace-driven temporal diagnosis, highlighting a bit-ordering error in the concatenation.

Table 10: Failing transaction captured at $t = 370$ ns (inputs sampled at `negedge clk`; next state observed at the following `posedge clk`).

Signal / State	Value	Description
`clk`	0 (`negedge`)	Sampling point for input vectors
`shift_ena`	1	Shift operation enabled (higher priority)
`count_ena`	1	Decrement enabled (masked by `shift_ena`)
`data`	0	Serial input bit for shifting
`q(t)`	1100	Register state before update
$q_{\text{ref}}(t{+}1)$	1000	Expected next state: `{q[2:0], data}`
$q_{\text{dut}}(t{+}1)$	0110	Observed next state: `{data, q[3:1]}`

Root Cause Identification. The localization tool identified the suspect logic within the `always_ff` block. Inspection of the RTL source confirms a directional implementation error: the design specification requires a left shift, but the implementation used a right-shift concatenation, placing `data` at the MSB rather than the LSB.

Code Patching. To resolve the discrepancy, the framework applied a localized patch that re-orders the concatenation to match the left-shift specification (placing `data` at the LSB), as shown in Listing 4.

```
--- buggy_shift.v
+++ corrected_shift.v
@@ -2,7 +2,7 @@
 always_ff @(posedge clk) begin
     if (shift_ena) begin
         // Shift operation
-        q <= {data, q[3:1]};      // Right-shift (BUG)
+        q <= {q[2:0], data};      // Left-shift (FIXED)
     end else if (count_ena) begin
         q <= q - 4'd1;
     end else begin
```

Listing 4: Localized code correction for the shift-direction bug.

Subsequent re-simulation confirmed that the revised logic matches the golden model for all exercised traces, including cases where `shift_ena` and `count_ena` are simultaneously asserted. By using localized patching instead of full-file regeneration, the framework eliminated the mismatch while preserving the stability of the remaining logic.

E Detailed Framework Comparison

This appendix provides a comparison between VERI-SURE and state-of-the-art frameworks, analyzing architectural differences, capability gaps, and performance characteristics.

Table 11 provides a systematic comparison of core architectural features and capabilities across leading frameworks. The table uses the following notation: ✓: Fully supported / High capability, ○: Partially supported / Medium capability, ✗: Not supported / Low capability, – : Not applicable.

Table 11: Comprehensive architectural and capability comparison of RTL generation frameworks.

Feature / Capability	VERI-SURE	MAGE	VerilogCoder	AutoChip	Single-Agent
Core Architecture					
Core Philosophy	**Contract & Verification**	Sampling	Planning & AST	Feedback	Feedback
Agents	**6 Specialized**	4 Specialized	4 Specialized	2 General	1 General
Communication	**Structured Contract**	Shared Context	Shared Context	Shared Context	Single Prompt
Test & Verification					
Testing Approach	**Hybrid (Sim+Formal)**	Simulation-only	Simulation-only	Simulation-only	Simulation-only
Formal Verification	✓	✗	✗	✗	✗
Assertion Generation	✓	✗	✗	✗	✗
Testbench Generation	✓	○	○	✗	✗
Debugging & Repair					
Repair Strategy	**Patching**	Whole-file	Whole-file	Whole-file	Whole-file
Error Localization	**Block-level**	Module-level	Module-level	File-level	File-level
Performance & Efficiency					
Cost Efficiency	Medium	Low	Low	**High**	**High**
Iteration Efficiency	**High**	Medium	Low	Low	Low
Regression Safety	**High**	Medium	Low	Low	Low

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the method, benchmark extension, and empirical claims, and these are supported by the main results in Table 1 and the ablations in Table 2.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5.5 discusses the limitations in the main text, covering under-specified descriptions, contract fidelity, and the reliability of formal hints, including the reliance on the golden testbench for trace reports and final acceptance. The appendix gives further details on failure modes, the contract audit, formal-signal reliability, and cost/token trade-offs in Appendix C.1, Appendix C.2, Appendix C.3, and Appendix C.5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not present theoretical theorems or proof claims; the equations define components of the proposed verification and repair pipeline.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 5.1 specifies the benchmark, baselines, metrics, toolchain, compute setting, and repair-iteration cap, while the appendix provides implementation details for contracts, slicing, tracing, and formal miters. Code and benchmark data are also open-sourced.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The abstract provides a GitHub link for the released code and dataset, and the appendix documents the benchmark construction and framework implementation needed to reproduce the evaluation.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5.1 describes the benchmark, syntax/function metrics, compared baselines, verification toolchain, model access mode, GPU type for open-weight models, and maximum repair iterations. Code and benchmark data are also open-sourced.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Each system is evaluated once under identical settings, because repeating all compared multi-agent systems would incur prohibitive API costs.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper states the use of official APIs, Verilator, SymbiYosys/Z3, and NVIDIA A100 80GB GPUs, and reports token/API cost in Appendix C.5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work uses code-generation benchmarks and generated RTL artifacts, does not involve private data or human subjects, and provides a public GitHub release link.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: The work studies RTL code-generation repair on public benchmarks and does not introduce a deployed system, dataset of sensitive information, or application-specific societal intervention.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release a high-risk pretrained generative model or scraped dataset; the released assets are an RTL benchmark extension and framework code.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Existing benchmarks, tools, and models are cited or named, and their licenses are respected: the VerilogEval-v2 benchmark (MIT), the MAGE (MIT) and VerilogCoder (Apache-2.0) baselines, Verilator (LGPL-3.0 or Artistic-2.0), SymbiYosys (ISC), and Z3 (MIT). Commercial models are accessed through their official APIs under the providers' terms of use, and open-weight models are used under the licenses given on their official model cards.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The paper introduces VERILOGEVAL-V2-EXT and documents its taxonomy, new task categories, difficulty grading, and implementation details in Appendix A; the GitHub repository is linked in the abstract. The code and the benchmark are released under the MIT license, the same license as the original VerilogEval-v2.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The research does not involve crowdsourcing experiments or human-subject data collection.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The research does not involve human subjects, so IRB or equivalent approval is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are central to the proposed multi-agent RTL generation and repair pipeline, and their roles are described throughout Figure 1, Section 5.1, and the framework section.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.