FERMIONS ASI CORP.
AI-powered intelligence for digital chip design
www.fermions.co

W H I T E P A P E R

SAiGE : Autonomous RTL Development

Agentic Verilog Generation with Closed-Loop Debugging

April 2026

Table of Contents

SAiGE : Autonomous RTL Development · Agentic Verilog Generation with Closed-Loop Debugging

Abstract

This paper describes the SAiGE Agent — part of the SAiGE platform, which is an autonomous RTL development system that generates, verifies, and debugs synthesizable Verilog modules from natural-language specifications. The system combines a structured design workflow, domain-specific guidance injected at tool-call boundaries, a curated resource library, and persistent state management to transform a general-purpose coding agent into a capable hardware design assistant within SAiGE.

SAiGE is an AI-native orchestration platform for digital chip design. The SAiGE Agent presented in this paper represents its RTL development capability.

Evaluated on the RTLLM 2.0 benchmark (50 Verilog designs), SAiGE achieves a 100% functional pass rate with Claude Sonnet 4.6 and generalizes across multiple LLMs — 92% with DeepSeek V3.2, 90% with Claude Haiku 4.5, and 84% with Qwen3 Coder.

1. Introduction

Hardware design automation has traditionally relied on template-based generators or formal synthesis from high-level specifications. Large language models capable of producing syntactically correct HDL code have introduced a new possibility: agentic RTL development within platforms like SAiGE, where an LLM autonomously writes, compiles, simulates, and debugs hardware designs in an iterative loop.

However, LLM-based code generation for hardware faces several practical challenges. Generated modules often diverge from testbench expectations in port naming or signal widths. Models struggle to interpret simulator output without structured guidance. Debug loops can stall when the agent repeatedly applies the same unsuccessful fix. And complex designs may exhaust the model's context window before reaching a solution.

We address these challenges with a domain-aware architecture within SAiGE built on four pillars: a structured workflow that guides the agent through specification intake, implementation, and verification; targeted guidance delivered at tool-call boundaries rather than in a static system prompt; a resource library encoding RTL design patterns and verification methodology; and persistent state that survives context window compaction. Together, these allow the system to receive the right information at the right time without prompt bloat.

Within the SAiGE platform, this capability operates in the Spec → RTL → Verification loop, enabling closed-loop design execution before downstream physical design.

1.1 Key Ideas

1. **Domain-aware agent architecture:** A structured workflow, targeted guidance, and curated resources transform a general-purpose coding agent into a capable RTL development system
2. **Closed-loop evaluation:** Each design is evaluated through a full autonomous session with compilation, simulation, and iterative debugging — not single-shot generation
3. **Cross-model generalization:** The same agent framework improves results across four different LLM families

2. Background

2.1 LLM-Based RTL Generation

The RTLLM benchmark (Lu et al., 2024) established a standard evaluation for LLM-generated Verilog, initially comprising 30 designs with provided testbenches. Reported functional pass rates for GPT-4 in a single-shot setting range from 37–58%. The benchmark was later expanded to 50 designs in RTLLM v2.0 (Liu et al., 2024), which has become the standard suite for evaluating agentic RTL systems.

Several approaches have been explored to improve on single-shot generation. Domain-specific fine-tuned models such as RTLCoder and ScaleRTL have pushed pass rates toward 83–91%. Agentic systems like REvolution apply iterative feedback with evolutionary search strategies, reaching 88% on RTLLM v2.0. Prompt engineering approaches such as COMBA-PROMPT have achieved up to 90% on related benchmarks.

2.2 Agentic Code Generation

The software engineering community has demonstrated that tool-using agents can substantially outperform single-shot generation by iterating on compiler and test feedback. SAiGE applies this paradigm to hardware design, where the feedback loop includes HDL linting, simulation with self-checking testbenches, and structured diagnostic output.

3. System Architecture

3.1 Overview

The SAiGE Agent operates within an agentic coding framework that provides a tool-calling interface (file read/write, shell execution), hook execution at lifecycle points, context window management, and headless execution for automated benchmarking.

On top of this, SAiGE adds four layers of domain specialization:

Workflow orchestration. A phased pipeline guides the SAiGE Agent from specification intake through implementation and verification. The workflow defines expected phases (read spec, extract interface, implement, compile, simulate, debug) and nudges the agent toward the next appropriate action based on its current state.

Targeted guidance at tool-call boundaries. Rather than encoding all domain knowledge in a system prompt, SAiGE injects feedback at lifecycle points — after file writes, after simulation runs, and before task completion. This includes automatic linting of HDL files, structured parsing of simulation output into pass/fail diagnostics, and escalating debug suggestions when the agent appears stuck. A completion gate prevents models from declaring success when tests are still failing.

Resource library. A curated set of coding standards, design patterns, and verification methodology documents are made available to the SAiGE Agent as needed. These cover common pitfalls in RTL design — clock domain crossings, signed arithmetic, reset handling — and provide reference implementations the agent can consult.

State persistence. Before context window compaction, SAiGE snapshots the current design state to disk — file inventory, simulation results, error history — so the SAiGE Agent can continue effectively after losing conversation history.

3.2 Workflow

For the RTLLM benchmark, the SAiGE Agent follows a fast-path workflow for single-module designs:

4. Read the specification and provided testbench
5. Extract the expected module interface (port names, widths, directions)
6. Implement the module
7. Compile and iterate on lint errors
8. Run simulation against the provided testbench
9. If tests fail, diagnose and fix — repeating until all tests pass or the timeout is reached

4. Evaluation

4.1 Benchmark

We evaluate the SAiGE Agent on 50 Verilog designs from the RTLLM 2.0 benchmark, spanning four categories:

Category	Count	Examples
Arithmetic	19	Adders, multipliers, dividers, fixed-point
Control	6	FSMs, sequence detectors, counters
Memory	5	Async FIFO, LIFO, shift registers, barrel shifter
Miscellaneous	20	Clock dividers, signal generators, ALU, width converters

Each design includes a self-checking Verilog testbench. The agent must implement a module that compiles cleanly and passes all testbench assertions within a 1200-second timeout.

4.2 Results

Primary Results (Sonnet 4.6)

Metric	Value
Syntax Pass Rate	50/50 (100%)
Functional Pass Rate	50/50 (100%)
Average Runtime	104s per design

All 50 designs pass across all four categories. By the time the agent runs the simulator binary for the first time, it has already read the testbench, written the RTL module, and resolved any lint errors — so the first simulation is a measure of RTL generation quality after informed implementation, not blind zero-shot generation. Even so, only 20/50 designs pass on this first simulation attempt; the remaining 30 are recovered through the closed-loop debug loop. Designs that passed first simulation required an average of 6.7 LLM API calls in total, while those that needed debugging required 11.8 calls — confirming the debug loop performs genuine iterative repair. The most time-consuming designs involve signed arithmetic (radix2_div, 314s), dual-clock domain crossing (asyn_fifo, 584s), and fractional clock division (freq_divbyfrac, 584s).

Cross-Model Results

Model	Context Window	Syntax	Functional	Avg Time
Sonnet 4.6	200K	50/50 (100%)	50/50 (100%)	104s
DeepSeek V3.2	164K	45/50 (90%)	46/50 (92%)	596s
Haiku 4.5	200K	49/50 (98%)	45/50 (90%)	132s
Qwen3 Coder Next	131K	44/50 (88%)	42/50 (84%)	653s

The results show a clear capability gradient. Control designs (FSMs, counters) are solved by all models. The challenging designs involve multi-cycle timing precision — fractional clock dividers, dual-clock FIFOs, signed division, and pipelined multipliers — where even small off-by-one errors cause complete test failure.

4.3 Comparison with Published Results

System	Approach	Benchmark	Functional
GPT-4 (Lu et al., 2024)	Single-shot	RTLLM v1.0	50%
RTLCoder-DeepSeek 7B	Fine-tuned	RTLLM v1.1	>47%
ScaleRTL-32B	Reasoning	RTLLM v1.1	82.8%
REvolution (DeepSeek-V3)	Agentic	RTLLM v2.0	88%
COMBA-PROMPT + GPT-4o Mini	Prompt eng.	RTLLM-guided	90%
SAiGE Agent (Sonnet 4.6)	**Agentic**	**RTLLM v2.0**	**100%**
SAiGE Agent (DeepSeek V3.2)	Agentic	RTLLM v2.0	92%
SAiGE Agent (Haiku 4.5)	Agentic	RTLLM v2.0	90%
SAiGE Agent (Qwen3 Coder)	Agentic	RTLLM v2.0	84%

Cross-version comparisons should be made carefully, as RTLLM v2.0 includes harder designs than the earlier versions. On the same v2.0 benchmark, our result with Sonnet 4.6 represents the highest reported functional pass rate to date, improving on the previously best-reported agentic result (REvolution at 88%) by 12 percentage points.

Notably, even our weaker model configurations — Haiku 4.5 (90%) and DeepSeek V3.2 (92%) — exceed prior state-of-the-art, and Qwen3 Coder (84%) matches REvolution's result with Llama-3.3-70B.

This demonstrates that SAiGE provides a model-agnostic intelligence layer, enabling consistent performance improvements across different LLM backends.

5. Discussion

5.1 Why Domain-Aware Guidance Helps

Encoding all domain knowledge in a static system prompt has practical limitations: instructions lose salience over long conversations, detailed prompts compete with code and tool outputs for context space, and fixed guidance cannot adapt to the agent's evolving state. The SAiGE architecture addresses these limitations by delivering relevant information throughout the session — at the points where it is most useful — rather than front-loading it. The overall effect is that the agent stays on track across complex, multi-step designs without paying a significant upfront context cost.

5.2 The Value of Iterative Feedback

The gap between single-shot generation (~50% for GPT-4) and our agentic results highlights the importance of closed-loop development. The first-simulation proxy metric makes this concrete: even after the agent has studied the testbench and implemented the module (passing lint), only 40% of Sonnet sessions pass on the first simulator execution. Designs that pass first simulation require an average of 6.7 LLM API calls total, while those requiring debugging average 11.8 calls — the debug loop is performing genuine iterative repair, not merely confirming already-correct code. Across all four models, the closed-loop architecture consistently recovers designs that fail on first simulation, with the recovery gap (first-sim → final pass rate) ranging from 20 to 48 percentage points depending on model capability.

Compiler feedback, simulation diagnostics, and structured guidance all contribute to helping the agent converge on correct implementations that would be unlikely in a single attempt. The combination of iterative feedback with domain-aware guidance is particularly effective for hardware design tasks where small errors — a wrong shift direction, an off-by-one in a timing counter — cause complete test failure with no partial credit.

5.3 Model Differences

All models handle straightforward designs reliably. The remaining failures tend to involve designs requiring precise multi-cycle timing — clock dividers, domain crossings, and pipelined arithmetic — where small errors cascade into complete test failures. Weaker models also require more iterations to converge, resulting in longer runtimes.

5.4 Infrastructure Considerations

Supporting multiple model families on a common evaluation platform required practical engineering around API routing, context window configuration, and output token budgets.

6. Limitations and Future Work

- The current evaluation focuses on single-module designs with provided testbenches. Multi-module integration and testbench generation have not been evaluated at the same scale.
- The benchmark is Verilog-only. SystemVerilog support exists in the agent but is not benchmarked.
- Verification is simulation-based. Formal property checking could catch corner cases missed by testbenches.
- Future directions include synthesis-in-the-loop evaluation (Yosys/Vivado for PPA feedback), multi-module benchmarks, and evaluation of testbench generation quality.

7. Conclusion

We have described a domain-aware agent architecture for RTL development that achieves, to our knowledge, the highest reported functional pass rate on the RTLLM v2.0 benchmark — 100% with Claude Sonnet 4.6 — and generalizes across model families with 92% (DeepSeek V3.2), 90% (Haiku 4.5), and 84% (Qwen3 Coder), each of which meets or exceeds prior published results.

The SAiGE Agent, a core component of the SAiGE platform, demonstrates how AI-native orchestration enables autonomous RTL development. It combines structured workflow orchestration, just-in-time guidance at tool-call boundaries, curated domain resources, and persistent state management to bridge the gap between general-purpose coding agents and the demands of hardware design.

The cross-model results suggest that this architecture provides a general-purpose improvement across LLMs, with the remaining performance gap between models driven primarily by capability differences on timing-sensitive designs.

Reference:

Deng, C., Tsai, Y., Liu, G., Yu, Z., & Ren, H. (2025). *ScaleRTL: Scaling LLMs with Reasoning Data and Test-Time Compute for Accurate RTL Code Generation* (pp. 1–9). https://doi.org/10.1109/mlcad65511.2025.11189212

Liu, S., Fang, W., Lu, Y., Wang, J., Zhang, Q., Zhang, H., & Xie, Z. (2024). 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. https://doi.org/10.1109/tcad.2024.3483089

Liu, S., Lu, Y., Fang, W., Li, M., & Xie, Z. (2024). *OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation* (pp. 1–9). https://doi.org/10.1145/3676536.3697118

Lu, Y., Liu, S., Zhang, Q., & Xie, Z. (2024). *RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model* (pp. 722–727). https://doi.org/10.1109/asp-dac58780.2024.10473904

Min, K., Cho, K., Jang, J., & Kang, S. (2026). *REvolution: An Evolutionary Framework for RTL Generation driven by Large Language Models* (pp. 282–288). https://doi.org/10.1109/asp-dac66049.2026.11420420

Nguyen, V., Do, D., Nguyen, N., & Le, D. (2025). *COMBA-PROMPT: Comprehensive Benchmark and Augmentation for Verilog Generation Leveraging Dual-LLM Prompting*. https://doi.org/10.36227/techrxiv.175339240.04076578/v2

Notice

This document is provided for informational purposes only and shall not be regarded as a warranty of any functionality, condition, or quality of any product or system. Fermions ASI Corp. (“Fermions”) makes no representations or warranties, expressed or implied, as to the accuracy, completeness, or reliability of the information contained in this document and assumes no responsibility for any errors or omissions.

This document does not constitute a commitment to develop, release, or deliver any product, feature, code, or functionality described herein. Fermions reserves the right to make changes, corrections, enhancements, or modifications to the content of this document at any time without notice.

The information presented is intended for general guidance only. Users are responsible for independently evaluating the applicability of this information for their specific use cases and for performing all necessary validation, testing, and verification.

Fermions shall not be liable for any direct, indirect, incidental, special, or consequential damages arising from the use of, or reliance on, this document or the information contained herein.

No license, express or implied, is granted under any intellectual property rights of Fermions. References to third-party products, services, or technologies do not constitute endorsement or recommendation and may require separate permissions or licenses from their respective owners.

Reproduction, distribution, or use of this document, in whole or in part, is permitted only with prior written consent from Fermions and must be reproduced without modification and with all notices included.

Important Disclaimer

THIS DOCUMENT IS PROVIDED “AS IS.” FERMIONS MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Copyright

Fermions — AI-powered intelligence for digital chip design

contact@fermions.co | *www.fermions.co*