Improving MAGE for Verilog RTL Generation with Agentic Routing and Memory Persistence

JunPyung Kim Mark Sui Bryan Zhu Dennis Zang
Department of Electrical and Computer Engineering
University of California San Diego
La Jolla, California, United States
{juk063,tisui,brzhu,dzang}@ucsd.edu

Abstract

This project studies how to extend MAGE [1], a multi-agent engine for LLM-based RTL code generation, with two debugging mechanisms: agentic failure routing and per-task memory persistence. Agentic routing classifies a failed Verilog attempt into syntax, interface, logic, or ambiguous-runnable failures and sends the attempt to a route-specific repair prompt. Memory persistence records the debugging trajectory inside each benchmark task, including candidate RTL checkpoints, syntax and simulation outcomes, mismatch counts, and the best state observed so far. On VerilogEval-V2 with `gpt-5.4-nano`, temperature 0.85, and a candidate budget of 5, the primary routed rerun produced 148 valid paired tasks. The MAGE baseline passed 137/148 tasks (92.6%), while agentic routing passed 136/148 (91.9%). Routing reduced estimated API cost from $1.452 to $1.395 and runtime from 4670 s to 4487 s, but the paired accuracy difference was not statistically meaningful. In a separate full 156-task memory-persistence run, pass count improved from 138/156 to 141/156. Overall, routing behaved mainly as an efficiency control, while memory persistence gave a small raw accuracy gain by making the editor aware of prior attempts and regressions.

1 Introduction

Large language models can generate short Verilog modules from natural-language specifications, but correct RTL generation remains difficult because the code must satisfy the required module interface, compile under HDL tools, and match the reference behavior cycle by cycle. A generated module that looks plausible may still fail because of a port-width mismatch, a parser issue, a reset-timing mistake, or a subtle truth-table error. Benchmarks such as VerilogEval make these failures observable with executable tests, but they also show that a single pass of text generation is usually not enough for reliable hardware generation. MAGE addresses this problem with a multi-agent workflow that separates testbench generation, RTL generation, simulation review, judging, and RTL editing.

Our project modifies MAGE in two directions. First, the agentic-routing module changes the repair loop so that failures are classified before editing. Instead of using a single generic repair instruction for every failed simulation, the system chooses a repair route and adds route-specific constraints to the RTL editor. This is meant to reduce unnecessary rewrites: syntax errors should be fixed before logic is redesigned, and interface warnings should not be ignored when the top-level module boundary is invalid.

Second, the memory-persistence module keeps the record of the debugging process inside each task. Without memory, an editor mainly sees the latest failed state and can repeat edits that already failed, overwrite a nearly correct candidate, or lose track of whether a change improved the mismatch count. The memory module stores checkpoints and exposes a compact summary of the best and recent

Preprint.

attempts so that the next edit can be conditioned on the local trajectory rather than on the most recent log alone.

The main research question is whether lightweight control around MAGE's existing agents can improve efficiency or accuracy without replacing the underlying generation model. Our results are split across the two extensions. Routing is slightly cheaper and faster in the primary rerun, but it does not outperform the baseline in pass rate. Memory persistence improves the raw full-set pass count, suggesting that preserving task-local debugging history is more directly useful for correctness than a coarse failure label alone.

2 Related Work

Early work such as DAVE framed English-to-Verilog generation as a machine-learning problem for translating natural-language intent into RTL [3]. VerilogEval later provided a practical benchmark for text-to-RTL generation because each problem includes a specification and an executable reference test [2]. This benchmark exposes failure modes that are common in generated hardware code: syntax errors, wrong module boundaries, sequential timing bugs, incorrect combinational simplifications, and incomplete finite-state machines. Our experiments use VerilogEval-V2 with Icarus Verilog as the compile and simulation toolchain [10].

Several recent RTL-generation systems use verification feedback to repair generated designs. VeriAssist and AutoVCoder both use iterative generation, checking, and self-correction, where tool output is fed back to an LLM for refinement [4, 5]. Other work directly studies how EDA tool feedback can improve LLM-generated Verilog [8]. These systems motivate our use of structured simulation logs, mismatch counts, and compiler messages, but they do not specifically test whether the repair prompt should be specialized by failure type or whether the editor should retain a persistent record of failed edits.

Multi-agent RTL systems add role separation to reduce context switching. VerilogCoder uses planning and waveform-oriented reasoning, while AIvril distributes generation and verification responsibilities across specialized agents [6, 7]. MAGE is the closest baseline for this project. It decomposes RTL generation into cooperating agents: a testbench generator, an RTL generator, a simulation reviewer, a judge, and an RTL editor [1]. This makes it more structured than a single self-repair loop, but it can still spend tokens on broad repair prompts even when the failure has a narrow cause. MAGE also motivates state checkpointing, although our memory module is narrower and more implementation-focused: it records per-task candidate outcomes and exposes the best and recent checkpoints to subsequent repair steps.

HDLFORGE studies adaptive model escalation for efficient Verilog generation [9]. Our project is related in spirit because both approaches try to use stronger or more expensive reasoning only when needed. However, we keep the model fixed in the main experiments and instead change the control layer around MAGE. Agentic routing changes which prompt is used after failure, while memory persistence changes what debugging history is available to the editor.

3 Method

3.1 Baseline MAGE Workflow

The baseline arm uses the existing MAGE top-level agent. For each benchmark instance, MAGE generates a testbench and interface, generates an RTL candidate, runs syntax and simulation checks, samples up to five RTL candidates, and invokes the RTL editor when candidate generation alone does not pass. In our experiments, the golden VerilogEval testbench is treated as the authoritative oracle, so testbench repair is disabled and the final pass/fail result comes from the benchmark simulation.

The important baseline property is that repair is handled through MAGE's normal judge/editor path. The judge sees the failed simulation log, failed RTL, failed testbench, and problem specification, then the editor applies actions such as replacing a matched block or replacing the whole RTL file. This gives the baseline a flexible repair loop, but the editor prompt is less tightly constrained by the type of failure. It also means that most of the usable debugging evidence comes from the current candidate and current log, rather than from an explicit memory of all candidates tried during the task.

3.2 Agentic Failure Routing

The routed arm enables `TopAgent.enable_failure_routing`. When the initial simulation fails, the top agent bypasses the normal simulation-judge decision and calls a log-driven failure classifier. The classifier assigns one of four routes:

- `syntax`: syntax check failures, parser errors, undeclared signals, invalid module items, invalid assignments, or elaboration failures.
- `interface`: missing modules, wrong port names, wrong port counts, port-width mismatches, padding/pruning warnings, or incompatible top-level boundaries.
- `logic`: runnable simulations with mismatched outputs, explicit mismatch summaries, or timeout-style functional failures.
- `ambiguous_runnable`: failures that ran far enough to produce tool output but did not match the syntax, interface, or logic patterns cleanly.

The selected route is written to `route_history.jsonl` and passed into the RTL editor. The editor normalizes aliases such as `syntax_repair` and injects route-specific goals into the repair prompt. Syntax repair asks for the smallest compile-unblocking change while preserving the interface. Interface repair focuses on module names, port names, port widths, and directions. Logic repair assumes syntax and interface are mostly correct and asks the model to use mismatch logs to localize behavioral bugs. Ambiguous-runnable failures reuse the logic prompt because they usually need behavioral debugging plus raw tool-log context.

3.3 Action Acceptance Guards

The routed editor does not blindly accept every proposed edit. After each edit action, it reruns syntax and simulation checks and records the result in `debug_history.jsonl`. A syntax-route edit is accepted if it advances to an interface or logic failure, changes the syntax-error signature, or passes simulation. An interface-route edit is accepted if it advances to logic, changes the interface-error signature, or passes. A logic-route edit is rejected if it introduces a syntax or interface blocker, and it is also rejected when the total mismatch count increases. These guards are simple, but they make the repair loop measurable: the log records whether a proposed edit made route-specific progress.

3.4 Memory Persistence

The memory-persistence arm adds a task-local debug memory. After candidate generation, syntax checking, simulation review, or editor repair, the system records a checkpoint with the RTL state, syntax result, simulation result, mismatch count, first mismatch time when available, output-level mismatch summaries, a short simulation-log excerpt, and the edit action associated with that state. These checkpoints are exported as `debug_memory.json` files inside each task folder, which makes the run auditable after the experiment and also gives the agent structured information during the run.

The memory module assigns a simple cost to each checkpoint so that states can be ranked. Passing simulations have the lowest cost. Runnable failures are ordered mainly by mismatch count, because a candidate with fewer mismatches is usually closer to the reference behavior than one with more mismatches. Non-executable states, such as syntax failures or interface blockers, receive a high cost because they do not provide reliable behavioral evidence. At later repair steps, the editor can receive a compact memory summary containing the best checkpoint and recent checkpoints, including whether the last attempted edits improved, preserved, or worsened the result.

This memory is different from the routing history. Routing history records a predicted failure class such as `syntax` or `logic`; memory records the actual debugging trajectory. It answers questions such as which RTL states were attempted, whether a proposed edit regressed the mismatch count, and which candidate has performed best so far. The memory is also local to a single benchmark task. It does not retrieve examples from other VerilogEval tasks and does not require a long-term vector database or external storage service. In this project, routing and memory were evaluated as separate extensions, but they are complementary: routing decides what type of repair prompt to use, while memory decides what prior evidence should be shown to that repair prompt.

Table 1: Primary paired rerun on VerilogEval-V2, candidate budget 5.

System	Tasks	Passes	Pass@1	Input tok.	Output tok.	Cost	Runtime
MAGE baseline	148	137	92.6%	2.418M	0.775M	$1.452	4670 s
Agentic routing	148	136	91.9%	2.515M	0.714M	$1.395	4487 s
Delta	–	-1	-0.7 pp	+4.0%	-7.9%	-3.9%	-3.9%

Table 2: Paired pass/fail outcomes for the primary rerun.

Outcome	Count
Both pass	133
Baseline only passes	4
Agentic routing only passes	3
Both fail	8

4 Results & Analysis

4.1 Experimental Setup

We evaluated on `verilog_eval_v2` using `gpt-5.4-nano` through the OpenAI provider. For the routing experiment, both arms used temperature 0.85, top-p 0.95, one repeat per task, maximum output length 4096, five RTL candidates, one selected candidate for editing, three editor trials, two simulation retries, and golden testbenches. Estimated API cost used $0.20 per million input tokens and $1.25 per million output tokens. The primary routing result is the June 3 rerun, which reuses the completed baseline records and adds missing baseline tasks so that 148 tasks have valid paired outcomes. The memory-persistence result was run separately on the full 156-task attempted set, so we report it as a raw full-set comparison rather than as a paired statistical test against the routing run.

Table 1 shows that agentic routing did not improve pass rate. It passed one fewer task than the baseline, a difference of only 0.7 percentage points. However, the routed system reduced total output tokens by 7.9%, total estimated cost by 3.9%, and runtime by 3.9%. The input-token count increased by 4.0%, which is expected because route-specific prompts and extra route metadata add context. The cost still decreased because output tokens are priced higher than input tokens under the experiment's pricing formula.

The paired view in Table 2 shows that most tasks are unaffected by routing: 133 tasks pass in both systems and 8 fail in both systems. Only seven tasks are discordant. A two-sided exact McNemar test on the discordant set gives $p = 1.0$, so the observed pass-rate difference should be treated as noise rather than evidence of an accuracy gain or loss.

Table 3 explains why routing did not improve accuracy. The router sent many failures to syntax and logic paths, but only five recorded route-specific repairs ended in a passing simulation, and all five were on the logic route. Syntax and interface routes were useful for constraining the editor, but they did not directly produce final successes in this rerun. The ambiguous-runnable route was especially weak: these failures often reflected tool or testbench execution issues that the current classifier could not localize.

4.2 Qualitative Outcomes

Agentic routing helped on `Prob156_review2015_fancytimer`. The initial routed simulation had 79,832 mismatched samples out of 200,000, including timing mismatches on `count`, `counting`, and `done`. The router selected the logic path, and a full-file logic repair passed simulation. This is the kind of failure the method is designed for: the top-level interface was valid, and the repair needed to reason about FSM timing.

The method also exposed a failure mode on `Prob062_bugs_mux2`. The log included a port-width warning, so the router selected the interface path. The editor then focused on a 1-bit/8-bit boundary and made local edits that changed the mismatch signature but did not solve the actual benchmark behavior. The baseline passed this task. This suggests that warnings are not always the root cause;

Table 3: Agentic-routing behavior from the primary rerun. Route counts include initial route-history records and editor debug-history records, so they do not have to sum to the number of tasks.

Route	Route count	Successful repairs	Failed repairs
`syntax`	80	0	60
`logic`	56	5	27
`ambiguous_runnable`	29	0	21
`interface`	4	0	3

Table 4: Memory-persistence results on the full 156-task VerilogEval-V2 attempted set.

System	Tasks	Passes	Raw pass rate	Δ vs. baseline	Runtime / artifact
MAGE baseline	156	138	88.5%	–	–
Memory persistence	156	141	90.4%	+3 / +1.9 pp	2:23:21; 156 memories

interface hints should be combined with behavioral evidence before constraining the editor too strongly.

On `Prob070_ece241_2013_q2`, both systems failed. The routed system selected the logic path and the editor proposed truth-table/K-map changes, but the acceptance guard rejected edits that increased the mismatch count from 49 to 75 or 106. This guard prevented clear regressions, but it could not infer the correct treatment of don't-care cases. This example shows that route-specific repair needs stronger semantic information than a total mismatch count.

4.3 Replication Run

An earlier May 30 run produced 139 valid paired tasks. In that run, the baseline passed 130/139 tasks (93.5%) and the routed system passed 125/139 (89.9%). Cost was nearly unchanged: \$1.334 for baseline versus \$1.339 for routed. The paired outcomes were 122 both-pass, 8 baseline-only, 3 routed-only, and 6 both-fail, with exact McNemar $p = 0.2266$. We treat the June 3 rerun as the primary result because it adds missing baseline tasks and covers 148 valid pairs, but the earlier run reinforces the same conclusion: routing did not produce a statistically supported pass-rate gain.

4.4 Memory Persistence Results

We tested whether MAGE improves when the editor can store previous RTL attempts during the same debugging run. Each checkpoint stores the RTL state, syntax result, simulation result, mismatch count, and whether the edit improved or worsened the result. The goal is not cross-task retrieval; it is to prevent the current task from losing useful local evidence.

Table 4 reports the full 156-task memory experiment. The original MAGE baseline passed 138/156 tasks, while the memory-persistence arm passed 141/156 tasks. This is a gain of three tasks, or 1.9 percentage points in raw pass rate. The run also produced 156 memory artifacts, one per attempted task, so the final result can be inspected at the checkpoint level rather than only through a task-level pass/fail record.

The main benefit is that the agent no longer debugs only from the latest error. For example, an exported memory trace for `Prob093_ece241_2014_q3` records several runnable candidates with 38 mismatches on `mux_in`, then records a later candidate with zero mismatches as the best checkpoint. This kind of trace is useful because it distinguishes repeated non-improvements from genuine progress. In a normal single-state loop, the editor may only see the most recent failure and can rediscover the same bad direction. With memory, the prompt can include the best known state and the recent failed states, giving the model a concrete reason to preserve working behavior or avoid a previously harmful edit.

The result should still be interpreted carefully. The memory run was not a combined routing-plus-memory experiment, and the raw comparison does not isolate every source of variance between runs. However, it supports the design intuition that RTL debugging benefits from persistent state. Hardware bugs often require several local hypotheses before the correct timing or combinational condition is

found; preserving the trajectory makes those hypotheses visible instead of treating every repair as a fresh attempt.

5 Limitations and Future Work

The routing labels are intentionally simple, but this simplicity is also the main limitation. Syntax, interface, and logic failures often overlap. A width warning can appear beside a real behavioral bug, and a logic error can create symptoms that look like an interface issue. In our primary rerun, the interface and ambiguous-runnable routes were especially weak, and only the logic route produced recorded successful repairs. Future routing should combine compiler messages with behavioral evidence before committing to a narrow prompt, and it should allow mixed labels when the root cause is uncertain.

The memory module is also deliberately lightweight. It ranks checkpoints using pass/fail status and mismatch count, which are useful but incomplete signals. A lower mismatch count is not always semantically closer to the correct RTL, especially for finite-state machines where a one-cycle offset can dominate the score. Richer memory could store waveform snippets, AST-level edit summaries, invariant hypotheses, or clock-cycle-localized mismatch windows. The most natural next experiment is to combine routing and memory in one arm: the router would select the repair mode, and the memory summary would tell the editor which prior states and hypotheses should constrain that repair.

6 Conclusion

This project extended MAGE with two lightweight debugging controls. Agentic routing makes the repair loop more explicit by recording failure routes, injecting route-specific editor goals, and rejecting edits that do not make measurable progress. On the primary VerilogEval-V2 rerun, it reduced estimated cost and runtime by 3.9%, but it did not improve pass rate: the routed system passed 136/148 tasks compared with 137/148 for the baseline.

Memory persistence addressed a different weakness: the lack of task-local debugging history. By storing candidate checkpoints, mismatch summaries, and best-so-far states, the memory arm improved the full-set raw result from 138/156 to 141/156 tasks. The overall takeaway is that failure-aware control can reduce overhead, but correctness improvements depend more on preserving and using concrete debugging evidence. A combined routing-plus-memory system is the most promising next step, because it would let MAGE choose an appropriate repair mode while also grounding that repair in the actual trajectory of previous RTL attempts.

7 Code & GitHub

The code for this project is available at: `https://github.com/junepaykim/MAGE.git`

The agentic-routing implementation is primarily located in `src/mage/agent.py`, `src/mage/rtl_editor.py`, and `src/mage/sim_judge.py`. The experiment driver used for the MAGE baseline versus agentic-routing comparison is `experiments/run_agentic_routing_experiment.py`. The result artifacts used in this report are stored under `experiment_results/gpt54nano_mage_vs_routed_20260530_152236` and `experiment_results/agentic_routing_rerun_20260603_182054`.

Memory-persistence implementation is in Mark's fork: `https://github.com/marksui/MAGE/tree/memory-only`
The main memory-persistence code is located in `src/mage/debug_memory.py`, with integration changes in `src/mage/agent.py`, `src/mage/sim_judge.py`, and `src/mage/rtl_editor.py`.

The memory result artifacts are reported as `memory_only_156/record.json` at `github.com/marksui/MAGE/tree/main/memory_only_156`. Each memory run also exports per-task `debug_memory.json` files, which record checkpoint history, best RTL state, mismatch information, and edit outcomes for debugging analysis.

8 Team Member Contribution

JunPyung Kim led the agentic-routing direction, including the initial idea proposal, debug-agent implementation, experiment design and execution, and the first draft of the report.

Bryan Zhu implemented the router-agent component for agentic routing, helped prepare the presentation materials, and supported team alignment throughout the project.

Mark Sui developed the memory-persistence module, ran the corresponding memory-persistence experiments, and assisted with report drafting.

Dennis Zang: No substantial contribution was documented.

A Teaching Evaluation Submission

A.1 JunPyung Kim

Figure 1 shows the teaching-evaluation submission confirmation for JunPyung Kim.

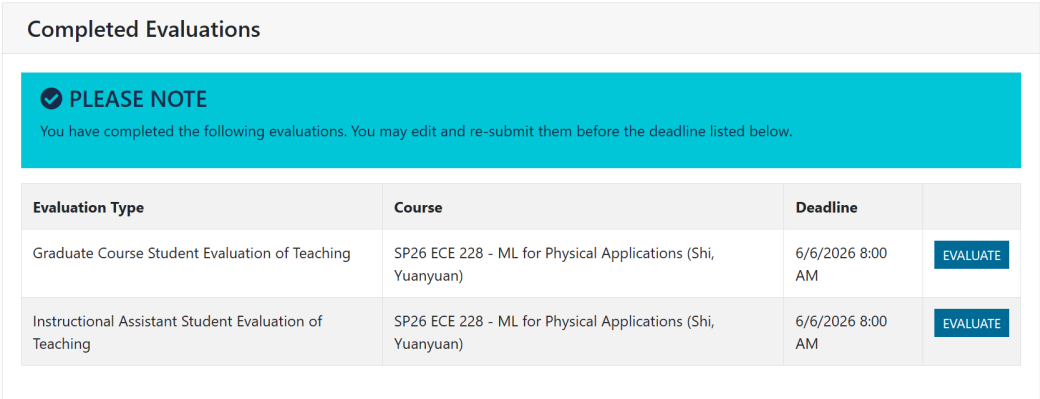

Figure 1: Teaching-evaluation submission confirmation for JunPyung Kim.

References

[1] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. arXiv preprint arXiv:2412.07822, 2024.

[2] M. Liu, N. Pinckney, B. Khailany, and H. Ren. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv preprint arXiv:2309.07544, 2023.

[3] H. Pearce, B. Tan, and R. Karri. DAVE: Deriving Automatically Verilog from English. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 27–32, 2020.

[4] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao. Towards LLM-Powered Verilog RTL Assistant: Self-Verification and Self-Correction. arXiv preprint arXiv:2406.00115, 2024.

[5] M. Gao, J. Zhao, Z. Lin, W. Ding, X. Hou, Y. Feng, C. Li, and M. Guo. AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs. arXiv preprint arXiv:2407.18333, 2024.

[6] C.-T. Ho, H. Ren, and B. Khailany. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and AST-based Waveform Tracing Tool. arXiv preprint arXiv:2408.08927, 2024.

[7] M. ul Islam, H. Sami, P.-E. Gaillardon, and V. Tenace. AIvril: AI-Driven RTL Generation With Verification In-The-Loop. arXiv preprint arXiv:2409.11411, 2024.

[8] J. Blocklove, S. Thakur, B. Tan, H. Pearce, S. Garg, and R. Karri. Automatically Improving LLM-based Verilog Generation using EDA Tool Feedback. arXiv preprint arXiv:2411.11856, 2024.

[9] A. Abdollahi, S. Shokoufa, N. Ashrafi, M. Kamal, and M. Pedram. HDLFORGE: A Two-Stage Multi-Agent Framework for Efficient Verilog Code Generation with Adaptive Model Escalation. arXiv preprint arXiv:2603.04646, 2026.

[10] Icarus Verilog Project. Icarus Verilog Documentation. `https://steveicarus.github.io/iverilog/`, accessed 2026-05-01.