RTL-BENCHLS: A LARGE-SCALE BENCHMARK FOR AGENTIC RTL REASONING AND IMPLEMENTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic frameworks are advancing rapidly, driving growing interest in agents for automated hardware design. Implementing hardware in RTL (i.e., RTL generation) is a key task in VLSI circuit design. However, *existing RTL benchmarks are too easy for cutting-edge agentic frameworks.* Specifically, we observe three gaps in the RTL benchmarks. *(1) Limited task scope.* Benchmarks are limited in module-level scope. Such module-level scope is insufficient to evaluate agentic frameworks. The EDA community urgently requires a qualified project-level RTL benchmark. *(2) Narrow task type.* Most benchmarks only focus on a narrow task scope. A single narrow task type is not enough to thoroughly evaluate agents on complicated RTL tasks. *(3) Small benchmark scale and design size.* Existing benchmarks are usually in small scale with small design size. To address these limitations, we introduce **RTL-BenchLS**, a large-scale benchmark of 17K+ cases with 12 tasks. We contribute in three aspects. *(1) New task scope.* We expand the scope from *module level* to *project level* with realistic project-level designs. Experiments show that our module-level cases are sufficiently challenging, and project-level tasks expose obvious limitations of current agentic frameworks. *(2) Expanded task type.* We cover three important types to evaluate agents' abilities from different perspectives, including generation, completion, and debugging. *(3) Enlarged benchmark scale and design size.* We collect significantly large scale of cases covering larger design size. To further increase the benchmark scale, we define a new task *protocol* named the *round-trip* protocol, which eliminates the need to manually craft qualified specifications and testbenches. As a result, we can significantly expand the benchmark scale using collected RTL designs. The *round-trip* tasks are also significantly challenging for the current LLMs. This new benchmark, at unprecedented scale, will help evaluate academic work and industrial solutions and improve LLM/agent solutions for hardware RTL design. Our benchmark will be released at `https://anonymous.4open.science/r/RTL-BenchLS-A0AB`.

1 INTRODUCTION

The rapid advancement of agentic frameworks has spurred growing interest in agents for automated hardware design (Pearce et al., 2020; Thakur et al., 2024; Chang et al., 2023; Thakur et al., 2023; Liu et al., 2024; Cui et al., 2024; Ho et al., 2024; Pei et al., 2024; Gao et al., 2024; Zhao et al., 2024). Industry is also placing increasing emphasis on automated RTL implementation. For example, Cadence recently introduced an RTL Generation Agent within its ChipStack AI Super Agent.[1] Implementing hardware in RTL (i.e., RTL generation) is a key task in VLSI circuit design. All VLSI circuits, including CPUs, GPUs, and AI accelerators, are implemented in RTL code (e.g., Verilog or VHDL code). However, *existing RTL benchmarks are too easy for cutting-edge agentic frameworks* (Kang et al., 2024; Huang et al., 2025; Zhu et al., 2025; Pinckney et al., 2025; Liu et al., 2025b). For example, MAGE (Zhao et al., 2024) and VerilogCoder (Ho et al., 2024) achieve functional pass rates above 90% on the RTL benchmark VerilogEval (Liu et al., 2023; Pinckney et al., 2024). Additionally, some failures are caused by flaws in the cases themselves (Wang et al., 2026). As a result, current benchmarks leave limited room to distinguish and further improve agentic frameworks.

[1] Cadence, "RTL Generation and Early PPA Optimization," 2026.

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

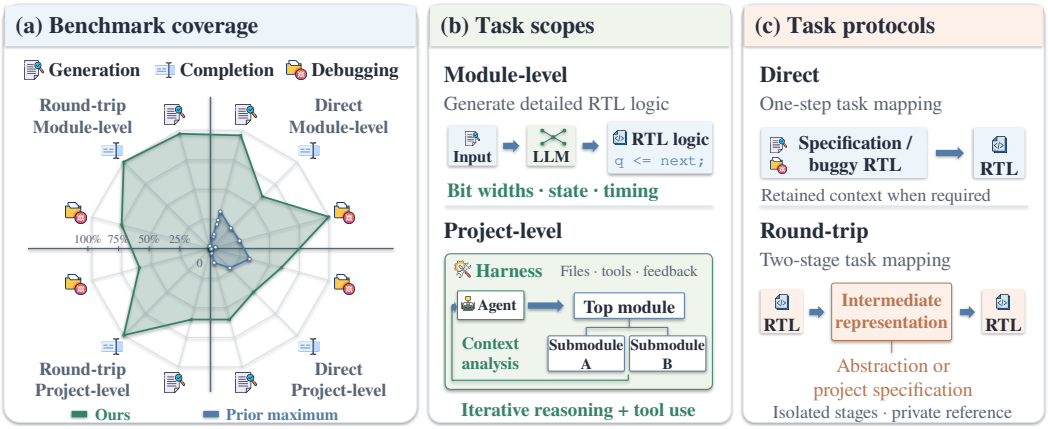


Figure 1: Overview of RTL-BenchLS. *(a) Benchmark coverage.* Our benchmark provides significantly larger scale and more tasks than existing benchmarks. *(b) Task scope.* In addition to traditional module-level tasks, we introduce project-level RTL tasks. (c) *Task protocols.* In additional to the traditional *direct* task, we propose a new *round-trip* task.

We observe three gaps in existing benchmarks on LLMs for RTL design. **(1) Limited task scope.** Widely used benchmarks such as VerilogEval (Liu et al., 2023; Pinckney et al., 2024) and RTLLM (Lu et al., 2024; Liu et al., 2025a) primarily assess RTL generation on module-level scope. Such module-level designs do not represent realistic industrial scenarios. As a result, the benchmarks are insufficient to evaluate cutting-edge agents on more realistic tasks. The EDA community urgently needs a qualified RTL benchmark at the *project level*. **(2) Narrow task type.** Existing benchmarks mainly focus on a single type of task, such as generation (Lu et al., 2024; Liu et al., 2023; Pinckney et al., 2024) or completion (Allam et al., 2024; Jin et al., 2025) task. A single narrow task type is not enough to thoroughly evaluate agents on complicated RTL tasks. **(3) Small benchmark scale and design size.** Existing benchmarks are small in scale and usually contain small designs. For example, the most representative benchmarks, VerilogEval (Liu et al., 2023) and RTLLM 2.0 (Lu et al., 2024), contain 156 and 50 designs, respectively. CVDP provides 783 cases with larger designs, but it has fewer than 200 cases per task.

To address the limitations, we introduce **RTL-BenchLS**, a large-scale RTL benchmark with 17K+ cases and 12 tasks. We contribute in three aspects, each targeting a specific limitation.

(1) New task scope. We expand the task scope from module level to project level with realistic project-level designs. Our benchmark includes two task scopes. Figure 1(b) depicts the two scopes and the distinct evaluation focus. The traditional *module-level* task scopes evaluate LLMs on detailed RTL logic within one module. We introduce the *project-level* scope, which requires an agent to understand project context, call tools, and finish the task iteratively. Experiments show that our cases at the existing module-level scope are sufficiently challenging, and project-level scope tasks help expose limitations of agentic frameworks.

(2) Expanded task type. To evaluate agents' ability thoroughly from different perspectives, we cover three important types, including generation, completion, and debugging. Specifically, *generation* creates RTL from a specification. *Completion* fills missing logic or modules using the specification and surrounding context. *Debugging* locates and corrects faulty RTL or reported issues. These task types require different reasoning and generation abilities of agents.

(3) A new task protocol for scaling up. To further increase the benchmark scale, we propose a new task **protocol** called the **round-trip** task. Figure 1(c) depicts the scenario for two task protocols. We define traditional RTL tasks as the *direct* protocol, where the task input is a supplied specification, RTL context, or issue report. In the *Round-trip* protocol, the agent must perform a two-stage task. In the first stage, the agent generates an intermediate representation of RTL code. In the second step, the agent takes the intermediate representation as input to recover or debug the RTL code. The *round-trip* protocol eliminates the need to manually craft qualified specifications and testbenchs. As a result, the benchmark scale can be significantly expanded from collected RTL designs. Experiments show that tasks in the *round-trip protocol* are significantly more challenging than traditional tasks with *direct protocol*.

Table 1: Comparison with representative RTL benchmarks on task coverage and scale.

Protocol	Scope	Type	VerilogEval (2023)	RTLLM 2.0 (2025a)	CVDP (2025)	RealBench (2025)	Location is Key (2024)	HWE-Bench (2026)	Ours
Direct	Module-level	Generation	156	50	78	60			**705**
		Completion			94				**1,296**
		Debugging			36		102		**1,564**
	Project-level	Generation			37	4			**50**
		Completion			26				**110**
		Debugging			11			417	**827**
Round-trip	Module-level	Generation							**9,696**
		Completion							**1,669**
		Debugging							**1,648**
	Project-level	Generation							**72**
		Completion							**618**
		Debugging							**596**

Enlarged benchmark scale and design size. We collect a significantly more cases covering larger design sizes. Figure 1(a) provides an overview of the benchmark. We contribute more cases on existing tasks and expand to new tasks. Specifically, we construct 12 tasks based on the three dimensions: two task *scopes*, three task *types*, and two task *protocols*. This new benchmark, at unprecedented scale, will help evaluate academic work and industrial solutions and improve LLM/agent solutions for hardware RTL design.

2 Related Work

Table 1 compares *RTL-BenchLS* with representative RTL benchmarks on different tasks. In the table, tasks are organized based on three dimensions: *protocol*, *scope*, and *type*. The numbers show the number of cases for each task. *RTL-BenchLS* contributes more tasks with larger scale and design size. Figure 2 compares *RTL-BenchLS* with representative benchmarks. Our benchmark contributes more cases across the design sizes. Next, we discuss the benchmark comparison at *module-level scope* and *project-level scope*.

Module-level scope. The most widely used RTL generation benchmarks primarily evaluate tasks at *module-level scope* and they are mainly limited in benchmark scale and design size. Specifically, VerilogEval (Liu et al., 2023; Pinckney et al., 2024) and RTLLM 2.0 (Liu et al., 2025a) contain 156 and 50 designs, respectively, and are mostly simple. CVDP contributes larger designs with more tasks, but remains limited in benchmark scale, with fewer than 200 cases per task. *RTL-BenchLS* provides more and larger cases across the tasks. Some methodologies, like CodeV-R1 (Zhu et al., 2025) and SpecLoop (Chang et al., 2026), use *round-trip reasoning* for dataset synthesis or specification refinement. We evaluate *round-trip* as a task protocol using formal verification as an evaluation method. Specifically, formal verification uses mathematical methods to check whether generated RTL matches a reference design's behavior. The *round-trip protocol* together with the formal verification-based methodology significantly expands the benchmark scale.

Project-level scope. Pioneering works have made considerable efforts on project-level tasks and are mainly limited at benchmark scale. CVDP (Pinckney et al., 2025) provides 166 agentic problems, with 37 specification-to-project and 26 module-reuse cases. Spec2RTL-Agent (Yu et al., 2025) explores specification-to-project generation based on three cases. RealBench (Jin et al., 2025) provides two project-level cases. HWE-Bench (Cui et al., 2026) offers 417 repository issue-fixing cases based on 6 GitHub repositories. RTL-BenchLS provides significantly more benchmark cases (1k+) from 50+ GitHub and OpenCores' projects.

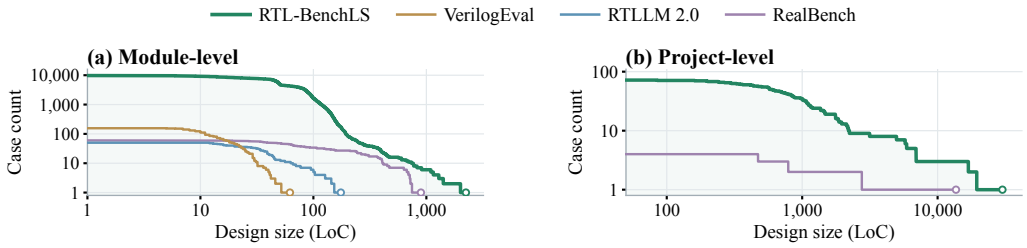


Figure 2: Comparison with representative benchmarks at one benchmark scale and design size.

3 RTL-BenchLS In Detail

3.1 Benchmark Tasks

We define each task by its task type, task scope, and task protocol.

Task type. We define three task types to evaluate different capabilities of agents. **Type 1:** *Generation* requires implementing RTL from a specification, including the required logic and interfaces. **Type 2:** *Completion* requires recovering missing logic or modules from the available specification and retained RTL context, while preserving compatibility with the surrounding design. In existing work (Liu et al., 2023; Lu et al., 2024; Pinckney et al., 2025), completion tasks are mainly defined as implementing the RTL logic based on the specification and an additional module header. In our work, we broaden the definition to fill-in-the-middle based on the surrounding context: LLMs may also be required to implement a missing code block or module in a project. **Type 3:** *Debugging* requires correcting an implementation to satisfy its specification or resolve a reported issue. We collect debugging cases in two ways: *(1) mutant-insertion*-based method for bug creation and *(2) real issue-fixing pairs* from realistic repository maintenance history.

Task scope. We separate two task scopes. *(1) Module-level* tasks evaluate the LLM backbone's ability on detailed RTL logic in a one-shot generation manner. *(2) Project-level* tasks target the agents' abilities in context-based reasoning and project development and maintenance, which requires iterative reasoning and tool calling. Agents may have to inspect the available project context, edit RTL, and invoke tools in isolated workspaces.

Task protocol. Existing benchmarks mainly rely on the **direct protocol**, which provides specifications, RTL context, or issue reports as input, and requires agents to generate desired output. In this protocol, input is predefined and fixed. In this work, we introduce the *round-trip* protocol. In the **round-trip protocol**, the agent generates an intermediate representation of the RTL design, which serves as input for the next reasoning step. For example, the agent may generate, complete, or debug RTL based on the intermediate representation and surrounding RTL context in the separate second step. We set two rules for the intermediate representation. *(1) Leakage rules:* the agent can not leak hardware code, like Verilog or VHDL. We use a script to detect code leakage. *(2) Code-based length limits:* the agent's intermediate representation should not exceed a length proportional to the RTL code. We use a *length limit ratio* r to control the length. This forces the agent to describe design functionality concisely while keeping the key semantics. This limitation also prevents the agent from describing overly detailed information, such as high-level operations, in natural language. This round-trip protocol is inherently harder because the agent must generate a qualified intermediate representation within a limited length and must understand it well enough in the second step. Additionally, the round-trip protocol increases the number of usable designs for benchmarking without requiring an original specification–RTL pair.

We derive 12 tasks from these three dimensions in Table 1. Direct Project-level debugging has three subtypes: mutant debugging, nonfunctional issue fixing, and functional issue fixing. Their scores are averaged in Table 2 and reported separately in Appendix C. Task names follow *task protocol* + *task scope* + *task type*. Appendix B lists the detailed task definitions. *Mutant debugging* requires the agent to debug a mutant of reference RTL based on the original specification. For real GitHub issue-fixing, we separate the fixings into *nonfunctional issue fixing* and *functional issue fixing*. We separate these two issue-fixing types to address a remaining limitation of HWE-Bench (Cui et al., 2026), which does not strictly validate that the commit makes functional modifications detectable by simulation. To address these limitations, we introduce formal verification tools to separate *functional*

from *nonfunctional* commits. Our investigation shows that some commits are functionally equivalent to the original code but still make meaningful modifications, such as removing redundant hardware logic and optimizing performance. We still retain these commits as *nonfunctional issue fixing*, using commit difference checks against the reference commit for evaluation.

Evaluation process. We utilize different evaluation metrics for tasks. For *module-level* generation and completion tasks, we report functional pass rate, following the metrics used in prior work (Lu et al., 2024; Liu et al., 2023). For *project-level* tasks, we utilize testpoint-based evaluation metrics. Project-level tasks are significantly harder for agents to implement fully correctly. But the agents can still make meaningful progress. We use the official testbenches and formal verification on key submodules to construct meaningful testpoints to measure progress in project generation. The metric is thus the passed testpoints / total testpoints for a project. For *design debugging*, we set *easy* and *hard* modes. *Easy* mode provides the bug location, and the agent is only required to fix the bugs. For *hard* mode, the agent has to identify the bug and fix it.

3.2 Benchmark Data Source

Data composition. We collect data from a broad range of sources for *module-level* and *project-level* tasks. *(1) On the module level. For the Direct protocol,* we collect 700+ qualified designs. *For the Round-trip protocol,* we collect 9k+ RTL designs, including 900+ from 25 open-source GitHub repositories and OpenCores IPs[2] and 9000+ from four public RTL datasets (Cui et al., 2024; Gao et al., 2024; Allam et al., 2025; Zhang et al., 2024). We apply strict filtering to the module-level RTL designs to remove duplicates and designs that fail compilation or formal-harness validation. The collection covers both combinational and sequential RTL, including logic, arithmetic, storage, finite-state control, interfaces, interconnects, processors, memory hierarchies, DSP and ML accelerators, security IP, and application-specific hardware.

(2) On the project level. We collect 72 projects in total from high-quality OpenCores IP and highly starred GitHub repositories, including repository issues and fix histories. These projects cover complete processor, SoC, communication, memory, interconnect, cryptographic, video, and accelerator systems. *For the direct tasks*, we curate 50 qualified projects that contain manually created project specifications.

Benchmark organization. Table 1 summarizes the case counts for each task. *(1) On the module level.* We organize cases in two ways. *Source-wise groups* organize RTL designs by source, indexed as S1–S5, to identify where the designs come from. Specifically, S1 contains repository designs, while S2–S5 correspond to the four public RTL datasets. We also divide cases into 20 slices to support evaluation in smaller batches. Each slice is balanced, with a matched proportion and design-size distribution to the full benchmark distribution. *(2) On the project level.* We organize cases by their source project. Appendix A.2 provides the detailed list of project-level cases.

4 RTL-BenchLS Generation Process

Figure 3 summarizes four stages for collecting qualified RTL designs and constructing test cases. We use automated scripts and agentic frameworks to automate repeated collection and primary validation. We then manually verify the collected cases.

❶ **Collect and extract designs.** We collect RTL, available specifications, and verification assets from repositories and datasets. During this process, we control source diversity and design size by removing duplicate designs and balancing lines of code (LoC) in the dataset. Two types of data cases are collected during this step: *specification-RTL paired* cases and *single RTL* cases. For *specification-RTL paired* cases, we further conduct qualification verification, especially of the specification against the reference RTL code. The *single RTL* designs are prepared for *round-trip* tasks.

❷ **Qualify designs and specifications.** In this stage, we first filter out the designs that are synthesizable and functionally verifiable. Specifically, designs must either pass the existing manually written testbench or be verifiable by formal verification tools. Then, we create the mutants from reference RTL code and validate that the testbench or formal scripts can successfully detect the

[2]OpenCores (`https://opencores.org/`) is a well-recognized community for reusable open-source IP. Its projects provide RTL sources and project metadata, such as design specifications and testbenches.

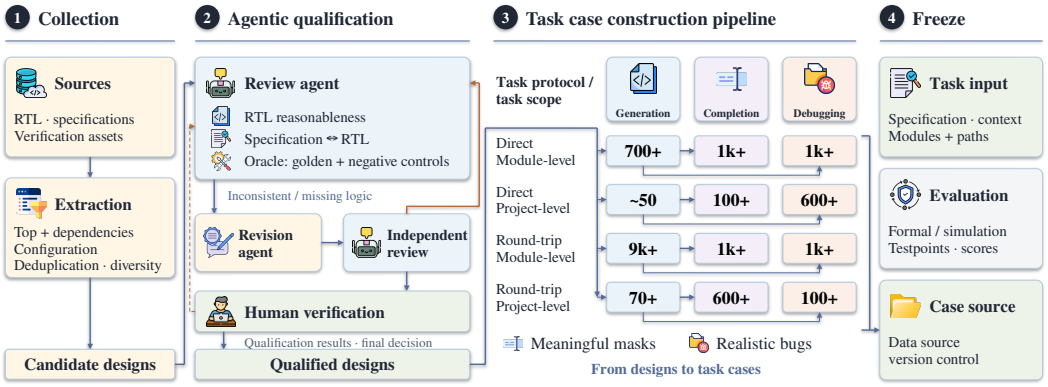


Figure 3: Data curation workflow in RTL-BenchLS, with four stages from design collection to frozen task cases. Task-case counts are rounded.

mutant bugs. After the design qualification step, we further conduct agentic validation on specification quality. Motivated by RTL-BenchMT (Wang et al., 2026), we design specification agents that review RTL behavior, check specifications against RTL, and diagnose inconsistent or missing requirements. Revisions undergo independent review, followed by final human verification. We only keep the specification-RTL pairs that pass the validation process.

❸ **Construct task cases.** Based on the qualified cases, we first construct **type 1: design generation** tasks. The *single RTL* designs are utilized to construct *round-trip* tasks. The *specification-RTL pairs* are utilized for *direct* tasks. For *design generation* tasks, the input is a design specification, and the required output is an RTL implementation. For *module-level* generation tasks, only one verification against the targeted top-module is conducted. For *project-level* design implementation tasks, the evaluation involves multiple testbenches or formal verification against important sub-modules within the project. We define each verification step for the project as a "*testpoint*." The project-level design generation task is significantly challenging, but the passed testpoints still effectively reveal the progress of the agent on the project implementation. *Type 2: design completion* and *Type 3: design debugging* are mainly derived from *Type 1: design generation* cases. Briefly, we construct *design completion* tasks by masking an important block within the module or an important module within a project. We construct *design debugging* tasks by creating mutants for the reference module or project. Detailed task formulations are included in Appendix A.

❹ **Admit and freeze cases.** This is a human-lead stage, where we double check the task formulation, evaluation pipeline, specification qualities. Only manually verified cases are accepted in our final benchmark dataset.

5 Experiments

5.1 Experimental Setup

Models and harness. We evaluate both the commercial models and open-source models on our benchmark cases. For commercial models, we select GPT-5.6 Sol, Claude Sonnet 4.5, and Claude Haiku 4.5. For open-source models, we select DeepSeek V3.2, Mistral Large, Qwen3.5-397B-A17B, and Llama 3.3 70B. *Project-level* tasks require an agentic harness. For commercial models, we use Codex CLI for GPT-5.6 and 5.4, and Claude Code for Claude Sonnet-4.5 and Haiku-4.5. For open-source models, including Mistral, Qwen, DeepSeek, and Llama, we use the open-source harness framework OpenHands.

Evaluation and verification. The metrics for different tasks are introduced in **evaluation process** of Section 3.1 Our functional verification relies on formal verification and testbench-based simulation. For designs with manually written testbenches (from GitHub repositories or OpenCores projects), we reuse the qualified testbench for simulation. For designs without testbenches, we utilize the formal verification tools to verify the generated RTL against the reference RTL code. For tasks at *round-trip protocol*, we set *length limit ratio* $r = 0.5$. Appendix C provides more detailed explorations for round-trip module-level tasks.

Evaluation subsets on round-trip tasks. Due to the large scale of the data cases and expensive API cost, for round-trip module-level tasks, we use slices 01, 05, 10, 15, and 20 for evaluation. In Appendix D, we show that each slice is balanced and has a similar data distribution.

5.2 Benchmarking Results

In this section, we mainly provide overall benchmarking results. Appendix C provides more detailed results and analysis. Table 2 reports functional scores for 12 combinations of task protocol, task scope, and task type. The three Direct Project-level debugging subtypes are averaged. *Direct Module-level generation*, a long-standing standard task, shows the highest performance among the models. GPT-5.6 Sol demonstrates the strongest ability across tasks, with strong generalization across scopes and agentic capabilities for project-level hardware implementation. Generally, the *project-level* and *round-trip* tasks are harder for LLM agents.

Table 2: Main results organized by task protocol, task scope, and task type.

Task protocol	Task scope	Task type	GPT 5.6 Sol	GPT 5.4	Sonnet 4.5	Haiku 4.5	DeepSeek V3.2	Mistral Large	Qwen 3.5	Llama 3.3
Direct	Module-level	Generation	74.2	54.1	54.6	38.6	46.4	26.1	39.1	19.7
		Completion	53.3	40.0	**67.2**	30.6	30.6	16.7	31.7	11.1
		Debugging	63.2	52.3	60.0	**70.0**	56.7	2.3	16.7	13.8
	Project-level	Generation	20.8	13.0	21.7	15.6	14.4	6.5	8.3	3.1
		Completion	37.5	25.0	33.3	25.0	36.1	16.7	16.9	1.7
		Debugging	34.5	34.3	37.6	32.5	28.2	23.0	16.7	10.0
Round-trip	Module-level	Generation	44.0	35.0	39.9	35.7	33.0	24.5	33.2	26.0
		Completion	29.1	22.4	21.5	22.8	22.7	18.0	22.9	23.4
		Debugging	74.1	63.6	70.6	63.4	57.8	33.6	55.2	41.3
	Project-level	Generation	27.5	6.2	10.0	2.4	1.3	1.6	3.7	2.3
		Completion	26.2	15.0	18.8	11.4	18.9	4.3	11.2	1.7
		Debugging	32.5	23.8	36.2	27.5	22.9	10.3	16.2	11.9

Task scope and task type. On *module level*, the *design completion* task is generally harder than other task types. For example, Direct and Round-trip Module-level completion show lower performance across models. This suggests the agent is less capable of context understanding and reasoning than of implementing a hardware design directly from a given specification. On the *project level*, the generation tasks are significantly harder than other task types. For Direct and Round-trip Project-level generation, we report testpoint pass rates rather than a "full pass rate." The testpoint pass rates are more informative than the full pass rate because many models' full pass rates are close to 0. Even the testpoint pass rates are clearly lower than other task types across models. This demonstrates the current agents' limitation on project-level tasks. Appendix C.1 further elaborates on the LLMs' limited ability on project-level tasks based on the token consumption statistics. **Task protocol.** The *Round-trip* protocol on *generation* and *completion* task types usually scores below their *direct* counterparts. This suggests the agent is less capable of RTL design summarization than directly generating hardware RTL code. The table reports the task performances under the code-based ratio $r = 0.5$. In the Appendix Table C.2, we further analyze the influence of increasing ratio on performance for round-trip tasks.

5.3 Benchmark Comparison

In this section, we compare *RTL-BenchLS* with existing module-level benchmarks along *task protocol* and *case difficulty*. Table 3 shows performance comparisons across benchmark cases and protocols. Rows with the same benchmark report the performance under the same benchmark cases with different protocols (*Direct* v.s. *Round-trip*). We compare cases in *Direct Module-level generation* of RTL-BenchLS with RTLLM (Lu et al., 2024) and VerilogEval (Liu et al., 2023).

For the **task-protocol** comparison, *Round-trip* pass rates are lower than *Direct* pass rates among models within each benchmark. This phenomenon demonstrates that our proposed *round-trip* tasks are more challenging than the standard *direct* protocol. For the **case-difficulty** comparison, *Direct* scores among models on RTL-BenchLS cases are below the corresponding RTLLM and VerilogEval cases. This suggests our benchmark cases are more challenging under the same task protocol. Additionally,

we include new evaluation results for downstream RTL-specific coders (Zhu et al., 2025) fine-tuned on the RTL training dataset. The RTL-specific coders also perform worse on RTL-BenchLS. For round-trip tasks, the RTL-specific coders' near-zero performance implies that these downstream models cannot generalize to round-trip tasks involving RTL reasoning and summarization.

Table 3: Comparison with module-level benchmarks on cases and task design.

Benchmark	Task protocol	GPT-5.4	Sonnet	Haiku	DeepSeek	Qwen	Llama	CodeV-R1 (RL)	QiMeng-SALV
RTLLM	Direct	74.2	68.2	74.0	62.3	56.7	49.8	68.0	62.0
	Round-trip	54.9	36.5	42.7	29.7	11.2	13.4	0.1	0.0
VerilogEval	Direct	82.1	80.1	79.5	70.5	58.3	53.2	68.8	57.1
	Round-trip	64.1	62.2	50.0	58.3	30.1	12.2	0.2	0.1
RTL-BenchLS	Direct	54.1	54.6	38.6	42.9	39.1	19.7	45.4	42.0
	Round-trip	39.7	40.3	26.8	33.6	27.3	12.5	1.6	1.3

6 Discussion

In this section, we further analyze the benchmarking results across design size and examine the failure reasons for each task.

6.1 Benchmarking Results across Design Size

Figure 4 summarizes performance across design sizes for each task. *Design generation* generally becomes harder as designs grow. Performance on *project-level generation* tasks, including the *direct* and *round-trip* protocols, declines as LoC increases. For *module-level* tasks, all models show relatively high performance on designs with LoC less than 100, and performance drops quickly as design size increases beyond 200. Compared with generation tasks, design size has less influence on completion and debugging tasks.

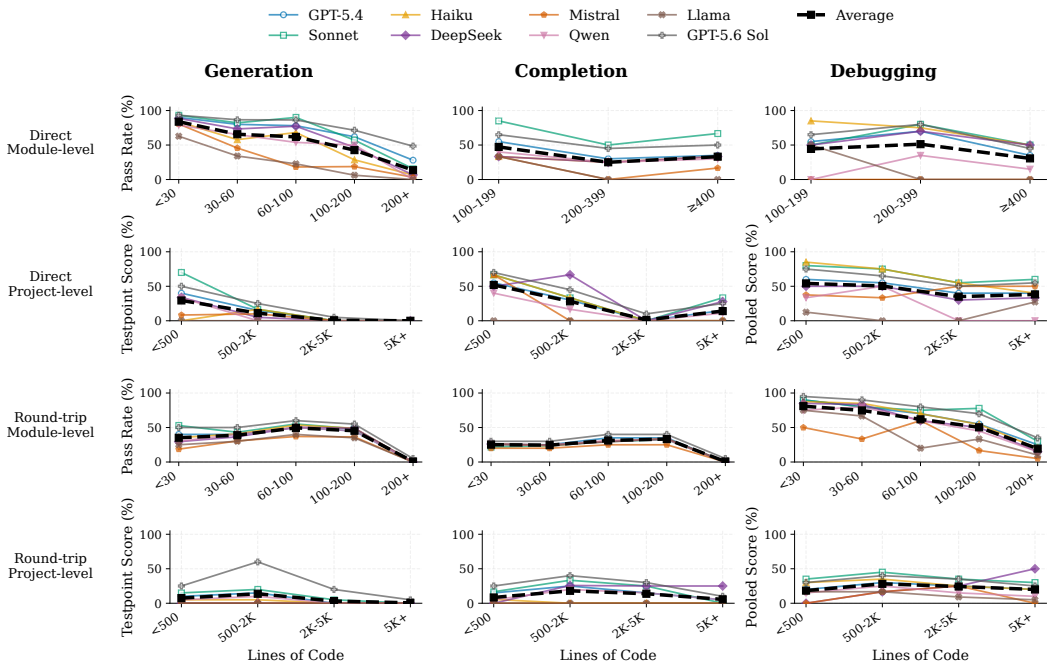


Figure 4: Performance across design size on different tasks. Curves show model results by LoC bin.

6.2 Failure Analysis

In this section, we further analyze failure reasons across tasks. Figure 5 shows the proportion of failure reasons in each task. In the figure, a *full pass* means a full pass on the task. There are some *common failure reasons* shared across all tasks. Specifically, *invalid/incomplete output* means the required RTL submission is incomplete or malformed (project-level tasks usually require the agent to submit in a required format). *Compile/elaboration failed* means that the RTL code has syntax errors and is rejected by tools. *Interface/integration error* refers to the port or module header definition mismatching the requirement. *Verification failure* means the RTL code passes syntax check but fails functional verification. There are also some *task-specific* failures. For example, a *partial pass* is specifically used in project-level tasks, where the generated RTL projects partially pass some testpoints. *Abstraction rejected* only exists in a round-trip task, where the intermediate representation violates code-leakage or length rules. *Localization failed* only exists in debugging tasks, where the agent wrongly identifies the fault location. The results suggest that verification failures are common in module-level completion and debugging, and compilation failures are prominent in direct project-level generation. *Abstraction rejection* is substantial in *Round-trip Project-level* tasks.

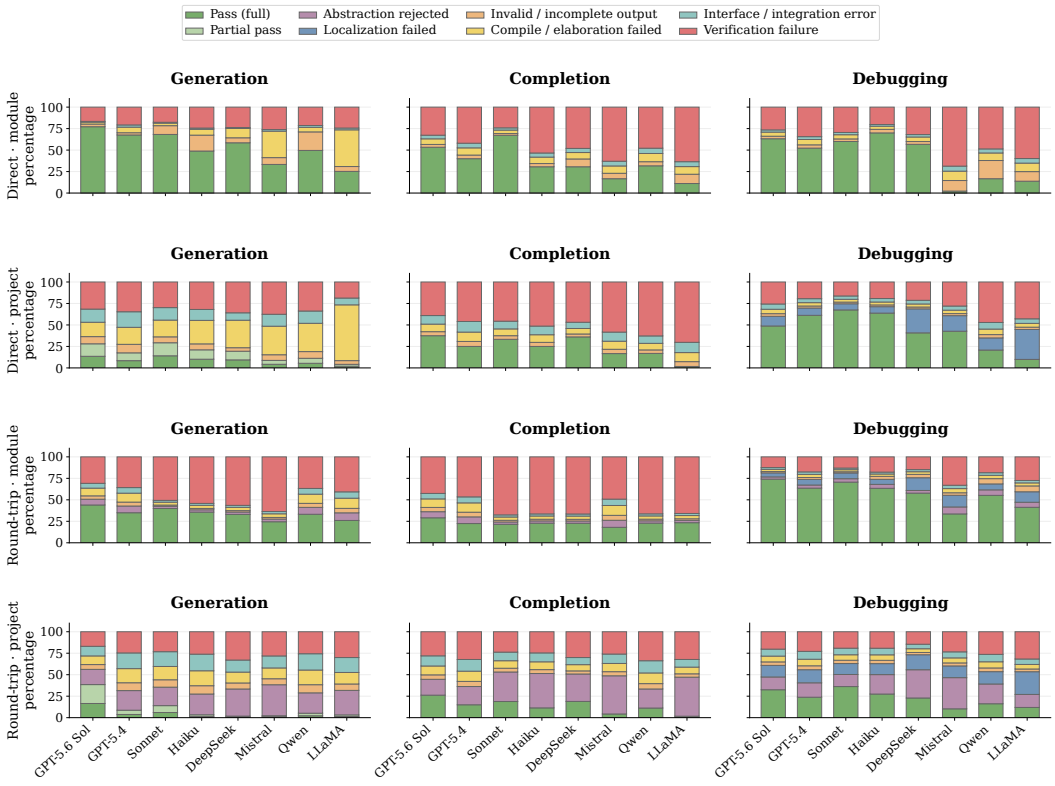


Figure 5: Error distributions on different tasks.

7 Conclusion

We presented RTL-BenchLS, a large-scale RTL benchmark with 17K+ cases and 12 tasks, spanning generation, completion, and debugging task types at the Module and project levels under Direct and Round-trip task protocols. Experiments show that our module-level cases are sufficiently challenging, and project-level tasks expose obvious limitations of current agentic frameworks. The *round-trip* tasks reveal a significant gap between models' ability to represent hardware behavior and their ability to generate RTL. This new benchmark, at unprecedented scale, will help evaluate academic work and industrial solutions and improve LLM/agent solutions for hardware RTL design.

Impact Statement

This benchmark paper collects all RTL designs and evaluation assets from open-source platforms, including GitHub repositories and OpenCores. The intended release contains public sources and evaluation assets. The authors will verify provenance and license terms before publication

AI Use Disclosure

We use Generative AI tools, including ChatGPT and Codex, to help with experiment design, code implementation, literature discovery, figure plotting, and manuscript editing. The human authors remain responsible for reviewing and validating the final references, figures, code, and benchmark artifacts. We have manually drafted the final manuscript.

References

Ahmed Allam et al. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. In *Workshop on LLM-Aided Design (LAD)*, 2024.

Ahmed Allam et al. Rtlpp: A parallel processing rtl code generation framework using llms. In *Workshop on LLM-Aided Design (LAD)*, 2025.

Fu-Chieh Chang, Yu-Hsin Yang, Hung-Ming Huang, Yun-Chia Hsu, Yin-Yu Lin, Ming-Fang Tsai, Chun-Chih Yang, and Pei-Yuan Wu. SpecLoop: An agentic rtl-to-specification framework with formal verification feedback loop. *arXiv preprint arXiv:2603.02895*, 2026.

Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*, 2023.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. *arXiv preprint arXiv:2407.16237*, 2024.

Fan Cui, Hongyuan Hou, Zizhang Luo, Chenyun Yin, and Yun Liang. HWE-Bench: Benchmarking LLM agents on real-world hardware bug repair tasks, 2026. arXiv:2604.14709.

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 *International Conference on Computer Design (ICCD)*, 2024.

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. *arXiv preprint arXiv:2408.08927*, 2024.

Hung-Ming Huang, Yu-Hsin Yang, Fu-Chieh Chang, Yun-Chia Hsu, Yin-Yu Lin, Ming-Fang Tsai, Chun-Chih Yang, and Pei-Yuan Wu. Assessing large language models in generating RTL design specifications. *arXiv preprint arXiv:2512.00045*, 2025.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, Zidong Du, Qi Guo, and Xing Hu. RealBench: Benchmarking Verilog Generation Models with Real-World IP Designs. *arXiv preprint arXiv:2507.16200*, 2025.

Minwoo Kang, Mingjie Liu, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. Fveval: Understanding language model capabilities in formal verification of digital hardware. *arXiv preprint arXiv:2410.23299*, 2024.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. OpenLLM-RTL: Open dataset and benchmark for LLM-aided design RTL generation, 2025a. arXiv:2503.15112.

Yuxiao Liu, Chenxing Xu, Yanrui Zhou, Zhen Li, and Qiang Xu. DeepRTL: Bridging verilog understanding and generation with a unified representation model. *arXiv preprint arXiv:2502.15832*, 2025b.

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)*, pp. 722–727. IEEE, 2024.

Hammond Pearce, Benjamin Tan, and Ramesh Karri. Dave: Deriving automatically verilog from english. In *Workshop on Machine Learning for CAD (MLCAD)*, 2020.

Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance. *arXiv preprint arXiv:2402.03375*, 2024.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks. *arXiv preprint arXiv:2408.11053*, 2024.

Nathaniel Pinckney, Christopher Deng, Chia-Tung Ho, Yi-Dian Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification. *arXiv preprint arXiv:2506.14074*, 2025.

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.

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.

Jing Wang, Shang Liu, Hangan Zhou, and Zhiyao Xie. Rtl-benchmt: Dynamic maintenance of rtl generation benchmark through agent-assisted analysis and revision. *arXiv preprint arXiv:2605.15537*, 2026.

Clifford Wolf and Johann Glaser. Yosys: A free Verilog synthesis suite. In *Proceedings of Austrochip 2013*, pp. 47–52, 2013.

Bingkun Yao, Ning Wang, Jie Zhou, Xi Wang, Hong Gao, Zhe Jiang, and Nan Guan. Location is key: Leveraging large language model for functional bug localization in verilog. *arXiv preprint arXiv:2409.15186*, 2024.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems. *arXiv preprint arXiv:2506.13905*, 2025.

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. Mage: A multi-agent engine for automated rtl code generation. *arXiv preprint arXiv:2412.07822*, 2024.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, et al. QiMeng-CodeV-R1: Reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.24183*, 2025.

A Benchmark Construction Details

A.1 Verification Methods

Testbench-based evaluation. We reuse upstream testbenches from the selected GitHub and OpenCores repositories, with support for Synopsys VCS and open-source Icarus Verilog. This flow mainly evaluates Project-level generation and also supports completion and debugging. We validate each testbench on the reference RTL. In project-level tasks, each testbench will be utilized as one testpoint.

Formal verification. Module-level tasks and formally checked Project-level completion and debugging tasks frequently use formal verification tools to check the equivalence of submitted RTL against the clean or developer-fixed reference. We support Cadence Conformal and Synopsys Formality for logical equivalence checking (LEC), Cadence JasperGold for sequential equivalence checking (SEC), and open-source Yosys (Wolf & Glaser, 2013).

A.2 Source Composition

Figure 6 provides a brief status on the design types across our benchmark. At *module level*, we have five sources, denoted from S1 to S5. Specifically, S1 contains 900+ designs from 25 GitHub repositories and spans processors, accelerators, networking blocks, and reusable hardware libraries. S2–S5 contribute 9k+ designs from OriGen, AutoVCoder, RTLPP, and MG-Verilog. Table 4 shows the data composition on the module-level tasks. From the 10k+ cases, we further derive 700+ high-quality *direct module-level generation* task cases.

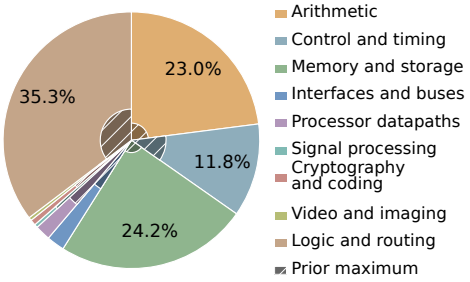


Figure 6: Design type statistics.

Table 4: Data distribution on round-trip module-level tasks.

Group	Source	Count	Avg. LoC	Max. LoC	Description
S1	nvdla/hw	404	41	479	NVIDIA DLA
	basic	230	19	198	General Verilog library
	oh	85	36	312	Open hardware library
	hdl	44	28	165	ADI HDL reference
	openc910	43	86	421	XuanTie C910 CPU
	zet	27	52	247	x86 processor
	v-ethernet	17	68	285	Ethernet MAC/PHY
	e200	16	44	189	HummingBird MCU
	Other repositories (17)	105	38	395	CPUs, FPU, FIFO, etc.
	S1 subtotal	**971**	**38**	**479**	
S2	OriGen	2,708	72	189	Augmented RTL corpus
S3	AutoVCoder	2,468	78	331	Auto Verilog generation
S4	RTLPP	2,164	82	263	Parallel-processing RTL
S5	MG-Verilog	1,717	29	156	Multi-grained dataset
	Total	**10,028**	**63**	**479**	

At *project level*, we collect 70+ cases from 50+ starred GitHub repositories and OpenCore projects. Table 5 lists the collected projects.

Table 5: List of collected projects for project-level generation.

Case ID	Source project
t1.4.upstream172.tiny_gpu	github/adam-maj/tiny-gpu
github/alexforencich/verilog-axi/axi_dma	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axi_fifo	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axi_register	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axil_adapter	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axil_cdc	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axil_reg_if	github/alexforencich/verilog-axi
github/alexforencich/verilog-axi/axil_register	github/alexforencich/verilog-axi
github/alexforencich/verilog-ethernet/arp	github/alexforencich/verilog-ethernet
github/alexforencich/verilog-i2c/i2c_master_axil	github/alexforencich/verilog-i2c
github/alexforencich/verilog-i2c/i2c_master_wbs_16	github/alexforencich/verilog-i2c
github/alexforencich/verilog-i2c/i2c_master_wbs_8	github/alexforencich/verilog-i2c
github/alexforencich/verilog-i2c/i2c_slave_axil_master	github/alexforencich/verilog-i2c
github/alexforencich/verilog-i2c/i2c_slave_wbm	github/alexforencich/verilog-i2c
github/alexforencich/verilog-pcie/dma_if_axi	github/alexforencich/verilog-pcie
t2.3.github_alexforencich_verilog-uart	github/alexforencich/verilog-uart
github/analogdevicesinc/hdl/ad_mux	github/analogdevicesinc/hdl
github/analogdevicesinc/hdl/axi_dmac_transfer	github/analogdevicesinc/hdl
github/chipsalliance/cores-veer-eh1	github/chipsalliance/cores-veer-eh1
t1.4.upstream172.veer_el2	github/chipsalliance/cores-veer-el2
github/corundum/corundum/mqnic_core_axi	github/corundum/corundum
t2.3.github_dpretet_async_fifo	github/dpretet/async_fifo
t1.4.upstream172.axi_crossbar	github/dpretet/axi-crossbar
github/evensgn/risc-v-cpu	github/evensgn/risc-v-cpu
github/lawrie/fpga_pio	github/lawrie/fpga_pio
t1.4.upstream172.step_mips_lab10	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab4	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab5	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab6	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab7	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab8	github/lvyufeng/step_into_mips
t1.4.upstream172.step_mips_lab9	github/lvyufeng/step_into_mips
t1.4.upstream172.schoolmips	github/mipsfpga/schoolmips
github/nonarkitten/amiga_replacement_project/cia8520	github/nonarkitten/amiga_replacement_project
github/pconst/basic_verilog/cookbook_des_crypt	github/pconst/basic_verilog
t2.3.github_secworks_aes	github/secworks/aes
t2.3.github_secworks_sha256	github/secworks/sha256
github/t-k-233/risc-v-single-cycle-cpu/riscv_core	github/t-k-233/risc-v-single-cycle-cpu

Continued on next page

Table 5 continued

Case ID	Source project
github/ultraembedded/riscv/riscv_tcm_top	github/ultraembedded/riscv
github/ultraembedded/riscv/riscv_top	github/ultraembedded/riscv
github/wangxuan95/fpga-ftdi245fifo/ftdi245fifo	github/wangxuan95/fpga-ftdi245fifo
t1.4.upstream172.uart2axi4	github/wangxuan95/fpga-uart
github/zipcpu/wbuart32/wbuart	github/zipcpu/wbuart32
opencores/aes_core/aes_cipher_top	opencores/aes_core
opencores/aes_core/aes_inv_cipher_top	opencores/aes_core
opencores/arithmetic_core_configurable_cordic_core_in_verilog	opencores/arithmetic_core_configurable_cordic_core_in_verilog
t1.4.upstream172.crcahb	opencores/arithmetic_core_crcahb
opencores/arithmetic_core_double_fpu_verilog	opencores/arithmetic_core_double_fpu_verilog
opencores/arithmetic_core_fpu/unique_tuples_16512_v1	opencores/arithmetic_core_fpu
t2.3.opencores_trig64_v1	opencores/arithmetic_core_trigonometric_functions_degrees_in_double_fpu
t2.3.opencores_gpio31_v1	opencores/communication_controller_general-purpose_i-o_gpio_core
t2.3.opencores_i2c_master_v1	opencores/communication_controller_i2c_controller_core
opencores/communication_controller_i2c_to_wb	opencores/communication_controller_i2c_to_wb
t2.3.opencores_simple_spi_v1	opencores/communication_controller_spi_core
t2.3.opencores_uart_to_bus_v1	opencores/communication_controller_uart_to_bus
t2.3.opencores_uartspi_8n1_v1	opencores/communication_controller_uart_to_spi
t2.3.opencores_lpc_host_v1	opencores/communication_controller_wishbone_lpc_host_and_peripheral_bridge
t2.3.opencores_aes128_pipeline_v1	opencores/crypto_core_aes-128_encryption
t1.4.upstream172.fast_aes_10cycle	opencores/crypto_core_fast_aes-128_encryption_only_cores
t1.4.upstream172.fast_aes_1cycle	opencores/crypto_core_fast_aes-128_encryption_only_cores
t1.4.upstream172.fast_aes_5cycle	opencores/crypto_core_fast_aes-128_encryption_only_cores
opencores/crypto_core_high_throughput_and_low_area_aes_core	opencores/crypto_core_high_throughput_and_low_area_aes_core
opencores/crypto_core_montgomery_modular_multiplier_and_exponentiator/modular_multiplier	opencores/crypto_core_montgomery_modular_multiplier_and_exponentiator
t2.3.opencores_present80_v1	opencores/crypto_core_present_cipher_encryption_core
t2.3.opencores_keccak512_high_v1	opencores/crypto_core_sha3_keccak
t2.3.opencores_sha256_v1	opencores/crypto_core_sha_cores
t2.3.opencores_pid32_v1	opencores/dsp_core_pid_controller
t2.3.opencores_sdram32_v1	opencores/memory_core_8-16-32_bit_sdram_controller
opencores/other_programmable_interval_timer	opencores/other_programmable_interval_timer
opencores/processor_turbo_8051/reference_8051	opencores/processor_turbo_8051

Continued on next page

Table 5 continued

Case ID	Source project
`t2.3.opencores_memory_acs32_v1`	`opencores/system_controller_memory_controller_ip_core`
`t2.3.opencores_vga_rgb_v1`	`opencores/video_controller_vga-lcd_controller`

A.3 Module-level Design Filtering

We extract self-contained modules and their dependencies, normalize source packaging, and screen syntax, source size, and module hierarchy. We remove duplicate RTL, and near-duplicate screening uses cosine similarity between TF–IDF vectors of character 4-grams. Figure 7 summarizes the pairwise similarity distribution.

We review RTL behavior and specification consistency, including interfaces, widths, reset behavior, and corner cases. Flawed specifications are revised and independently reviewed. Unresolved specifications and designs are removed. Formal qualification checks whether the designs are compilable and functionally verifiable. Specifically, we formally verify the designs against functionally different mutants to validate that our formal verification pipeline is functioning correctly.

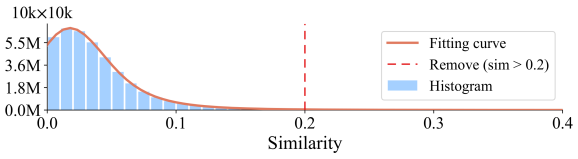


Figure 7: Pairwise similarity of collected Module-level RTL designs.

B Complete Task Definitions

Table 6 summarizes 12 task with their indexing. We clearly define the task formulation here, including the input, output, and verification methods. Direct tasks use supplied specifications or issue reports; Round-trip tasks first derive an intermediate specification from clean RTL and then use it in a separate recovery stage. Reference RTL and verification assets remain hidden from the evaluated generation, completion, or repair stage.

Notation and scoring. Let f_θ denote the model or agent, D a clean module, P a clean project, S a supplied specification, and A a derived specification. Write $X_{\setminus R}$ for RTL with region R removed and $X \oplus Y$ for inserting a completion or applying a patch. The formal score $F(X, R)$ is one when X is proven equivalent to reference R, and zero otherwise. For project generation and completion, $V_P(X)$ uses the case's declared verification method: the fraction of frozen testbenches passed, or the binary formal score. Each complete testbench is one testpoint. Invalid RTL receives zero.

Round-trip constraints. Specification extraction and recovery use isolated contexts; only the validated intermediate specification crosses between stages. Each case declares its permitted representation and length limit, and rejects copied RTL. A failed intermediate check gives an end-to-end score of zero without recovery. The formulas below describe recovery after this check passes.

B.1 Type 1: Generation

Task 1.1: Direct Module-level Generation. The model generates a complete module from a validated specification. Formal verification compares the generated RTL with the hidden clean module.

$$D' = f_\theta(S), \qquad s_{1.1} = F(D', D). \tag{1}$$

Task 1.2: Direct Project-level Generation. The agent receives a project specification, declared interfaces, and permitted public assets, then implements the required project RTL. Hidden testbenches

Table 6: Task contracts in RTL-BenchLS. Arrows indicate input–output mappings; parentheses specify verification methods.

Task	Task name	Input → output (verification)
1.1	Direct Module-level Generation	Specification → module (formal verification).
1.2	Direct Project-level Generation	Specification → project RTL (testbench or formal verification).
1.3	Round-trip Module-level Generation	Module → specification → module (formal verification).
1.4	Round-trip Project-level Generation	Project RTL → specification → project RTL (testbench or formal verification).
2.1	Direct Module-level Completion	Specification + masked module → completion (formal verification).
2.2	Direct Project-level Completion	Specification + retained project → missing module (testbench or formal verification).
2.3	Round-trip Module-level Completion	Module → specification; specification + masked module → completion (formal verification).
2.4	Round-trip Project-level Completion	Project RTL → specification; specification + retained project → completion (testbench or formal verification).
3.1	Direct Module-level Debugging	Specification + module mutant → repair (formal verification).
3.2a	Direct Project-level Debugging (nonfunctional issue)	Nonfunctional issue + project RTL → patch (commit match).
3.2b	Direct Project-level Debugging (functional issue)	Functional issue + faulty project RTL → patch (formal verification).
3.2c	Direct Project-level Debugging (mutant)	Specification + project mutant → patch (formal verification).
3.3	Round-trip Module-level Debugging	Module → specification; specification + mutant → repair (formal verification).
3.4	Round-trip Project-level Debugging	Project RTL → specification; specification + mutant → patch (formal verification).

or formal verification evaluate the implementation.

$$P' = f_\theta(S_P), \qquad s_{1.2} = V_P(P'). \tag{2}$$

Task 1.3: Round-trip Module-level Generation. The first stage describes a clean module; a fresh call reconstructs it from that description alone (Figure 8). Intermediate representations include natural language, diagram-oriented descriptions, executable Python or C, and free-form abstractions, subject to the declared format and length limits.

$$A = f_\theta(D), \qquad D' = f_\theta(A), \qquad s_{1.3} = F(D', D). \tag{3}$$

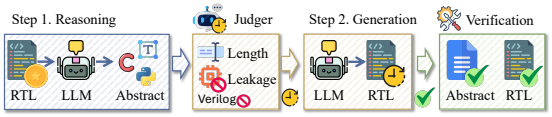


Figure 8: Task 1.3 reconstructs a module from an intermediate specification without access to the original RTL.

Task 1.4: Round-trip Project-level Generation. The first stage derives a specification from the clean project. A fresh agent receives this specification and the public interface contract, then reconstructs the required project without the reference RTL.

$$A_P = f_\theta(P), \qquad P' = f_\theta(A_P), \qquad s_{1.4} = V_P(P'). \tag{4}$$

B.2 Type 2: Completion

Task 2.1: Direct Module-level Completion. The model receives a supplied specification and a module with a missing RTL region B. It reconstructs that region, and formal verification checks the completed module.

$$B' = f_\theta(S, D_{\setminus B}), \qquad s_{2.1} = F(D_{\setminus B} \oplus B', D). \tag{5}$$

Task 2.2: Direct Project-level Completion. The agent implements a missing module M from its validated specification and the retained project context. The recovered module is reinserted into the project for hidden evaluation.

$$M' = f_\theta(S_M, P_{\setminus M}), \qquad s_{2.2} = V_P(P_{\setminus M} \oplus M'). \tag{6}$$

Task 2.3: Round-trip Module-level Completion. The first stage derives module intent from clean RTL. A fresh call receives this intent and the masked module, then reconstructs the missing block (Figure 9). Masks cover complete syntactic regions, such as `always`, `assign`, or `case` blocks.

$$A = f_\theta(D), \qquad B' = f_\theta(A, D_{\setminus B}), \qquad s_{2.3} = F(D_{\setminus B} \oplus B', D). \tag{7}$$

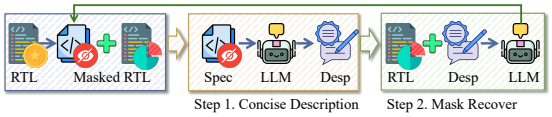


Figure 9: Task 2.3 reconstructs masked RTL using derived intent and the retained module context.

Task 2.4: Round-trip Project-level Completion. The first stage describes the missing region R in its clean project context. A fresh agent uses this description and the retained project to reconstruct R, which may be a logic region or a module. Evaluation checks the reintegrated project.

$$A_R = f_\theta(R, P), \qquad R' = f_\theta(A_R, P_{\setminus R}), \qquad s_{2.4} = V_P(P_{\setminus R} \oplus R'). \tag{8}$$

B.3 Type 3: Debugging

Direct Project-level debugging is Task 3.2, with three scored subtypes: nonfunctional issue fixing (3.2a), functional issue fixing (3.2b), and mutant repair (3.2c).

Let $\mu(X)$ denote a compile-valid mutant whose behavior differs from the clean reference, I an issue report, and Δ a predicted patch. Easy mode supplies the reviewed fault location; hard mode requires correct identification before repair. Below, $g = 1$ in easy mode and is the identification-pass indicator in hard mode; mode-specific location inputs are omitted for brevity. Failed identification stops repair and yields zero.

Task 3.1: Direct Module-level Debugging. The model repairs a module mutant using the original validated specification. Formal verification compares the repaired module with its clean parent.

$$\Delta = f_\theta(S, \mu(D)), \qquad s_{3.1} = g\, F(\mu(D) \oplus \Delta, D). \tag{9}$$

Task 3.2a: Direct Project-level Debugging (nonfunctional issue). The agent addresses a nonfunctional issue in a buildable project while preserving its behavior. The predicted patch is evaluated by commit match against the withheld developer change $\Delta^\star$.

$$\Delta = f_\theta(I_{\text{nonfunc}}, P), \qquad s_{3.2a} = g\ \text{CommitMatch}(\Delta, \Delta^\star). \tag{10}$$

Task 3.2b: Direct Project-level Debugging (functional issue). The agent receives a functional issue and the buildable buggy revision P_{bug}. Formal verification compares the repaired RTL with the developer-fixed reference P_{fix} (Figure 10).

$$\Delta = f_\theta(I_{\text{func}}, P_{\text{bug}}), \qquad s_{3.2b} = g\, F(P_{\text{bug}} \oplus \Delta, P_{\text{fix}}). \tag{11}$$

Task 3.2c: Direct Project-level Debugging (mutant). The agent repairs a project mutant using the unchanged original project specification. Formal verification checks the repaired design against the clean parent; mutation metadata and the golden repair remain hidden.

$$\Delta = f_\theta(S_P, \mu(P)), \qquad s_{3.2c} = g\, F(\mu(P) \oplus \Delta, P). \tag{12}$$

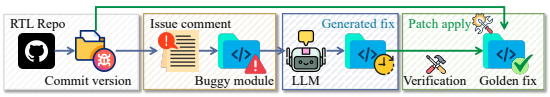


Figure 10: Task 3.2b repairs a repository issue and verifies the result against the withheld developer-fixed behavior.

Task 3.3: Round-trip Module-level Debugging. The first stage derives intent from the clean module. A fresh call repairs its mutant using only that intent and the faulty RTL. Formal verification checks equivalence to the clean module.

$$A = f_\theta(D), \qquad \Delta = f_\theta(A, \mu(D)), \qquad s_{3.3} = g\,F(\mu(D) \oplus \Delta, D). \tag{13}$$

Task 3.4: Round-trip Project-level Debugging. The first stage derives a specification from the clean project. A fresh agent uses it to repair the mutant project, with formal verification against the hidden clean reference.

$$A_P = f_\theta(P), \qquad \Delta = f_\theta(A_P, \mu(P)), \qquad s_{3.4} = g\,F(\mu(P) \oplus \Delta, P). \tag{14}$$

C Additional Benchmarking Results

C.1 Token Consumption and Effective Project Generation

Figure 11 collects token consumption for functional outcomes for Direct Project-level Design Generation (Task 1.2). For DeepSeek, the averaged token consumption rises from 684k tokens on Simple projects to 965k on Hard projects, while the mean testpoint pass rate falls from 33.8% to 0%. On Medium projects, approximately 92% of its recorded tokens are spent on attempts that pass no testpoints; on Hard projects, all tokens failed. Llama consumes fewer tokens (95k–229k per project) but passes no testpoints in any group, so lower consumption alone does not indicate effective generation. These results show that *current models are limited on project-level RTL tasks, with substantial token consumption that yields much less functional pass.*

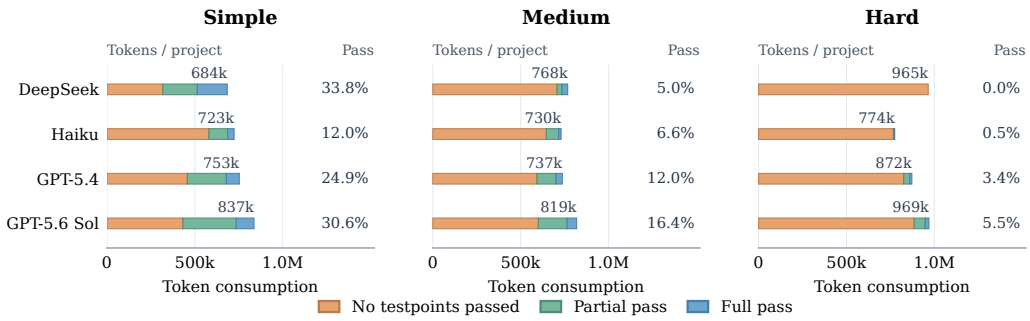


Figure 11: Token consumption and functional outcomes for Task 1.2. Bars show input plus output tokens per project, partitioned by final outcome and averaged over all projects in each group; percentages are mean per-project testpoint pass rates.

C.2 MODULE-LEVEL ROUND-TRIP GENERATION WITH DIFFERENT FORMATS

Table 7: Task 1.3 functional pass rates (%).

Representation	Setting	GPT-5.4	Sonnet	Haiku	DeepSeek	Mistral	Qwen	LLaMA
$\mathcal{A}_{\text{NL}}$	$r = 0.5$	35.3	17.1	11.7	10.3	6.8	4.6	7.9
$\mathcal{A}_{\text{NL-Diag}}$	$r \approx 0.9$	46.1	39.5	30.3	16.4	24.6	3.7	3.3
$\mathcal{A}_{\text{Code}}$	Python ($r \approx 0.9$)	49.7	52.1	31.4	39.5	31.4	0.7	18.8
$\mathcal{A}_{\text{Code}}$	C ($r \approx 1.1$)	43.2	39.5	26.4	33.8	28.3	28.3	19.8
$\mathcal{A}_{\text{Free}}$	$r = 0.5$	38.0	20.6	16.0	14.1	19.2	15.0	14.8
$\mathcal{A}_{\text{Free}}$	$r = 1.0$	48.8	34.0	27.4	22.7	28.9	26.9	18.4

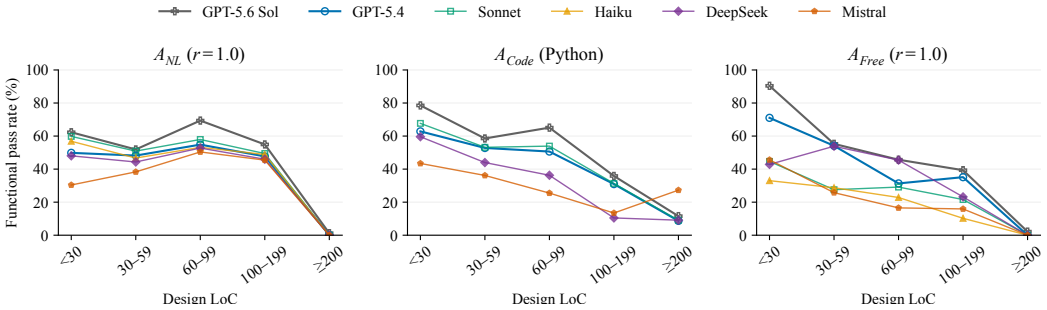


Figure 12: Results on Round-trip module-level generation tasks.

For round-trip tasks, we can explicitly require the format for intermediate representations. For example, in addition to the natural language ($\mathcal{A}_{\text{NL}}$), we can also require the agent to generate Diagram-oriented ($\mathcal{A}_{\text{NL-Diag}}$) and executable code ($\mathcal{A}_{\text{NL-code}}$).

Executable code as intermediate representation. For *module-level round-trip RTL generation* tasks, an interesting exploration is to utilize software code as a representation. We design these experiments to examine whether the pre-trained knowledge on software code can help with hardware design. We pick Python and C as two representative code-based intermediate representations. Python has usually helped achieve higher final RTL pass rates for Sonnet, Haiku, DeepSeek, and Mistral. On the contrary, Qwen3.5 and Llama perform better with C. We observe that Python code as an intermediate representation performs clearly better than other intermediate representations.

Executable intermediate versus final RTL. Because executable code can also generate simulation traces. We compare the software code's simulation results against the reference RTL. Figure 13 demonstrates the results. A considerable proportion of generated software code simulates equivalently to the reference RTL. The final RTL can outperform its intermediate on the simulation-based check. This phenomenon implies that an imperfect intermediate may still preserve enough behavioral information for reconstruction.

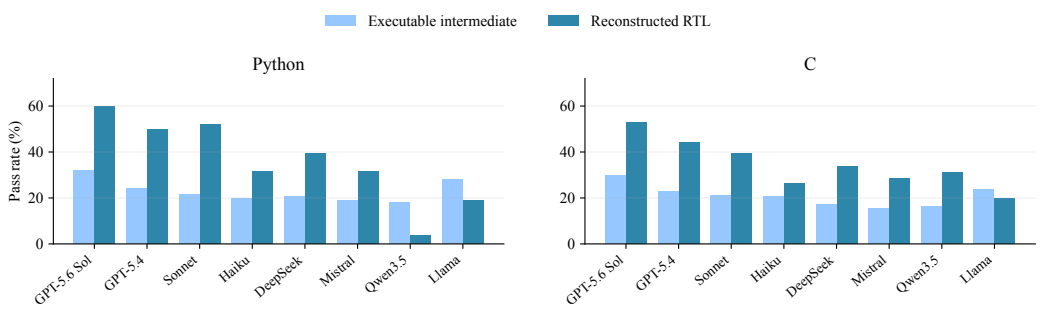


Figure 13: Task 1.3 executable-intermediate and reconstructed-RTL pass rates for Python and C.

C.3 Token Consumption in Task 3.2 Issue Fixing

Direct Project-level debugging subtype scores. Table 8 reports the detailed debugging results on three subtypes. Figure 14 separates token consumption by final outcome for the issue-fixing subtypes of Task 3.2: nonfunctional issues (3.2a) and functional issues (3.2b). Easy mode supplies the issue location and distinguishes failed from successful repairs. Hard mode additionally distinguishes unsuccessful identification from successful identification followed by a failed repair. From the token consumption analysis, we observe that issue fixing is harder than issue identification. Many tokens succeed on the issue identification step while failing at the fixing step.

Table 8: Direct Project-level debugging subtype functional scores (%).

Subtype	GPT 5.6 Sol	GPT 5.4	Sonnet 4.5	Haiku 4.5	DeepSeek V3.2	Mistral Large	Qwen 3.5	Llama 3.3
Nonfunctional issue fixing	44.6	34.7	39.6	30.0	31.7	19.8	24.8	14.9
Functional issue fixing	10.0	7.0	5.6	3.7	12.0	6.5	4.6	5.3
Mutant debugging	48.8	61.2	67.5	63.8	40.8	42.7	20.8	9.9

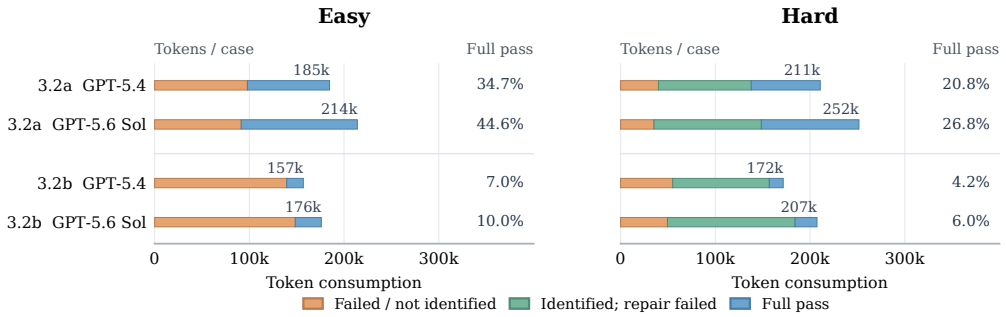


Figure 14: Issue-fixing token consumption and pass rates for Task 3.2 subtypes 3.2a and 3.2b.

D Slice Balance and Additional-Slice Evaluation

In this section, we report the audit of the slices 01, 05, 10, 15, and 20 for round-trip module-level tasks to demonstrate that our slicing machenism is balanced. This ensures that a single slice's result can represent the whole round-trip dataset on module-level tasks.

Figure 15 shows source composition and design size across all 20 original released slices. Mean design LoC ranges from 70.1 to 74.1 across slices, and median LoC ranges from 56 to 60. Source composition also remains broadly aligned across slices. This demonstrates that the slicing is balanced. Figure 16 compares slice-level performance; GPT-5.4 and DeepSeek remain within a stable performance range across slices. Figure 17 shows GPT-5.4 and DeepSeek performance as the design size increases. The results demonstrate that different models demonstrate similar performance trends on each slice.

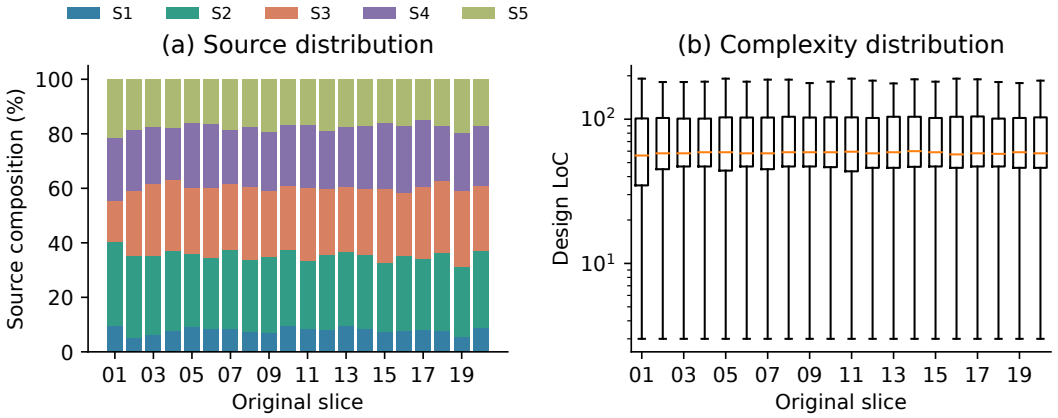


Figure 15: Balance of the 20 original released slices. Left: S1–S5 source composition. Right: design LoC distributions on a logarithmic axis; boxes show the interquartile range and median, with outliers omitted from display.

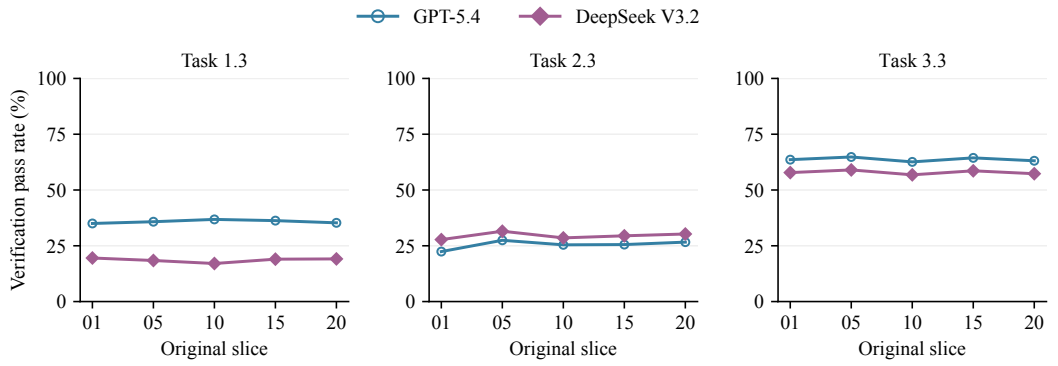


Figure 16: Five-slice verification pass rates for round-trip module-level generation (Task 1.3), completion (Task 2.3), and debugging (Task 3.3).

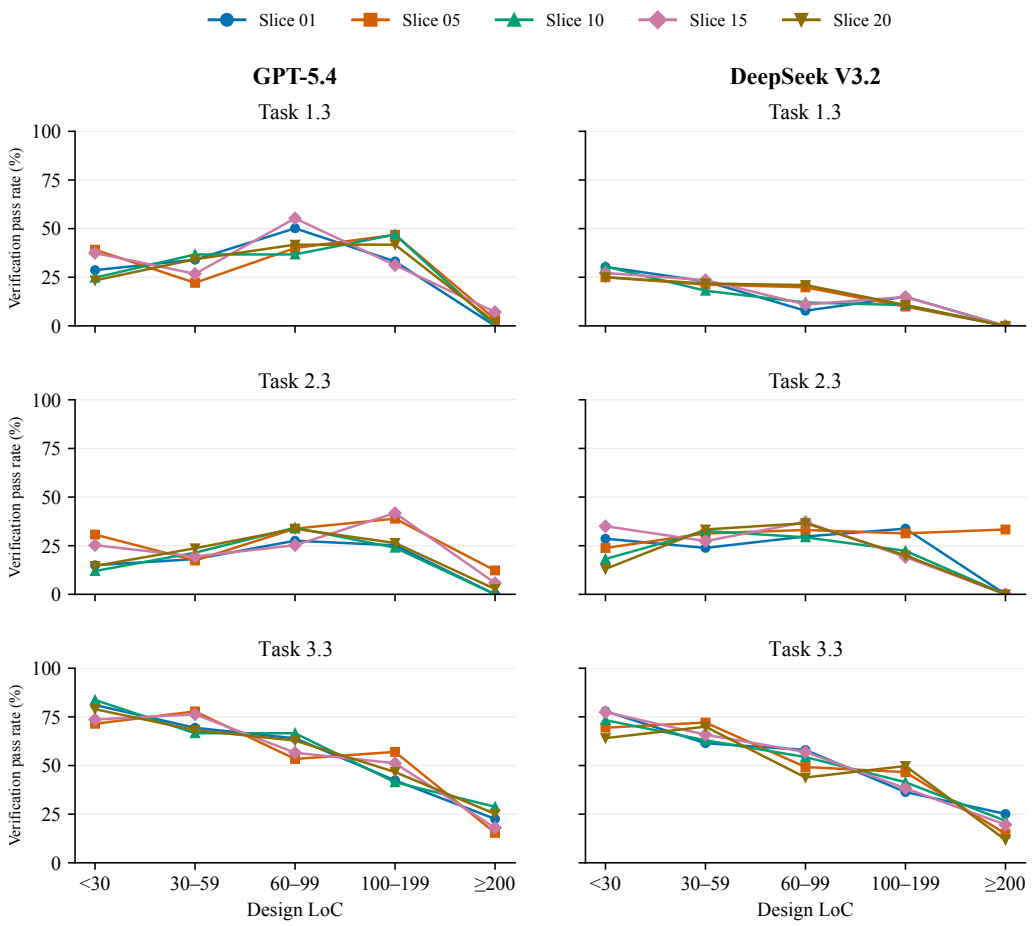


Figure 17: LoC-conditioned verification pass rates by model and slice. Columns: GPT-5.4 (left), DeepSeek V3.2 (right). Rows: Tasks 1.3, 2.3, and 3.3. Each panel shows slices 01, 05, 10, 15, and 20.

E Detailed Analysis and Case Studies

E.1 Representative Cases for Module-level Round-Trip Tasks

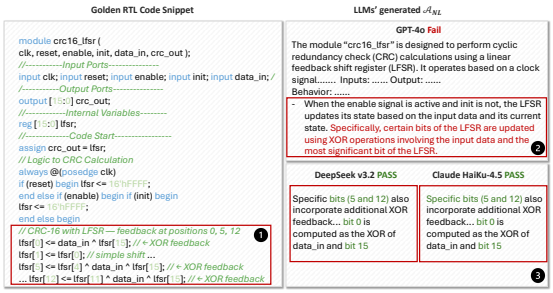


Figure 18: Task 1.3 case: the abstract must preserve the exact LFSR feedback positions to reconstruct the CRC behavior.

Case 1: round-trip design generation.

Figure 18 shows a 49-LoC, 16-bit CRC design. The reference uses LFSR feedback XORs at bit positions 0, 5, and 12. GPT-4o produces a vague intermediate representation stating only that "certain bits" use XOR, so the reconstruction loses the exact taps and fails equivalence. DeepSeek-v3.2 and Haiku-4.5 explicitly preserve all three indices and reconstruct equivalent designs. The case illustrates how a superficially plausible specification can omit one low-level detail that determines the complete behavior.

Case 2: round-trip completion.

Figure 19 masks the overflow logic of an 8-bit add/subtract design. The golden block sets `overflow` to the XOR of `intermediate_carry[6]` and `carry`, covering both paths. DeepSeek-v3.2 and Mistral Large describe overflow in terms of carry XOR and reconstruct the correct formula. GPT-4o and Sonnet-4.5 interpret it as generic signed overflow and emit a familiar sign-bit formula that fails on subtraction. The surrounding design is not sufficient unless the model infers the local convention precisely.

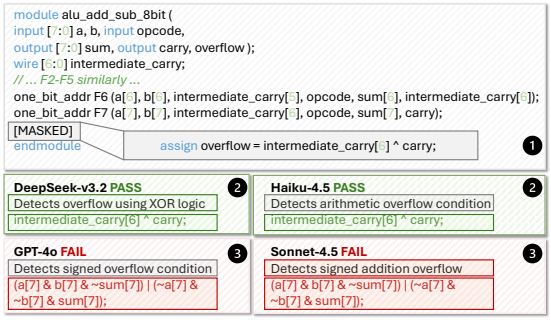


Figure 19: Task 2.3 case: models must infer the design-specific overflow convention rather than substitute a generic formula.

E.2 Examples for Issue-Fixing

Case 3: functional issue-fixing. Figure 20 uses lowRISC/ibex issue #1178. The golden fix changes one slice bound in the PMP NAPOT address mask from `PMPGranularity+2` to `PMPGranularity+1`, while leaving nearby +2 occurrences unchanged. Sonnet-4.5 and DeepSeek-v3.2 make the exact edit; LLaMA-3.3 produces a functionally equivalent rewrite. GPT-4o globally changes every +2, corrupting unrelated widths. Haiku-4.5 localizes the line but adds an unnecessary branch that fails formal equivalence. Passing repairs change at most one line, whereas these failed repairs change three or more.

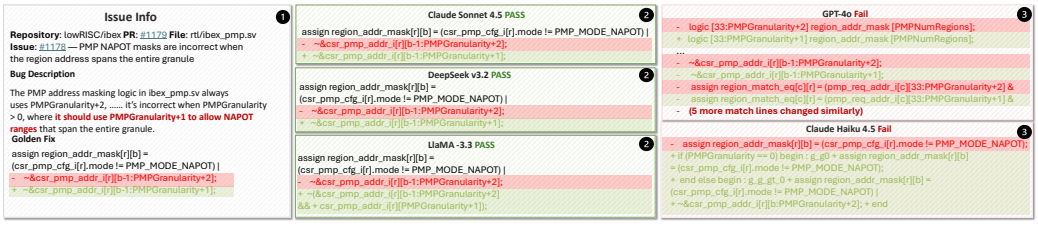


Figure 20: Task 3.2b case: the issue requires one precise bit-slice correction rather than a global textual replacement.

E.3 Examples on a project-level case: VGA/LCD Controller

Figure 21 shows the functional hierarchy and the five top-level testpoints for the case Example `opencores_vga_rgb_v1`. Each testpoint is a complete simulation check. We provide the evaluation results on GPT-5.6 Codex CLI and DeepSeek OpenHands.

Case 4: GPT-5.6 Sol with Codex CLI. The agent read the public specification, module interfaces, and output contract, then wrote all 15 required RTL files. It ran the public checker, corrected an address-decoding warning, and exercised local pixel and frame-buffer simulations. The final submission passed the output-contract and project-compilation gates. In the frozen evaluation, pixels, buffer switching, and underrun passed; registers and timing failed, with testpoint pass rates of 3/5.

Case 5: DeepSeek V3.2 with OpenHands. The agent read the same inputs and wrote all 15 required RTL files. Its public checker reported an invalid register part-select and an invalid top-level output-port connection. The agent fixed the part-select, then reached its 32-iteration limit before repairing the port connection. The final output-contract gate passed, but elaboration failed because the generated `vga_pgen.clut_adr` output was connected to `{cbsw, 8'd0}`, which contains a constant and cannot be driven. The generated RTL fails compilation, so all testpoints fail with compilation errors before functional simulations. This indicates the difficulty of project-level tasks for cutting-edge models.

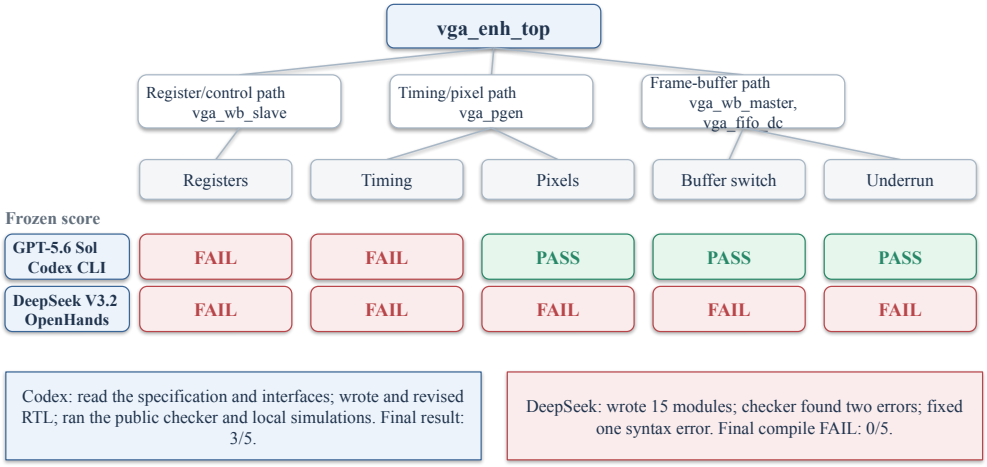


Figure 21: Two agent runs on the same VGA/LCD controller project.