Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation

NATHANIEL PINCKNEY, NVIDIA Corp, Austin, United States
CHRISTOPHER BATTEN, Cornell University, Ithaca, United States
MINGJIE LIU, NVIDIA Corp, Austin, United States
HAOXING REN, NVIDIA Corp, Austin, United States
BRUCEK KHAILANY, NVIDIA Corp, Austin, United States

The application of large language models (LLMs) to digital hardware code generation is an emerging field, with most LLMs primarily trained on natural language and software code. Hardware code like Verilog constitutes a small portion of training data, and few hardware benchmarks exist. The open-source VerilogEval benchmark, released in November 2023, provided a consistent evaluation framework for LLMs on code completion tasks. Since then, both commercial and open models have seen significant development.

In this work, we evaluate new commercial and open models since VerilogEval's original release—including GPT-4o, GPT-4 Turbo, Llama3.1 (8B/70B/405B), Llama3 70B, Mistral Large, DeepSeek Coder (33B and 6.7B), CodeGemma 7B, and RTL-Coder—against an improved VerilogEval benchmark suite. We find measurable improvements in state-of-the-art models: GPT-4o achieves a 63% pass rate on specification-to-RTL tasks. The recently released and open Llama3.1 405B achieves a 58% pass rate, almost matching GPT-4o, while the smaller domain-specific RTL-Coder 6.7B models achieve an impressive 34% pass rate.

Additionally, we enhance VerilogEval's infrastructure by automatically classifying failures, introducing in-context learning support, and extending the tasks to specification-to-RTL translation. We find that prompt engineering remains crucial for achieving good pass rates and varies widely with model and task. A benchmark infrastructure that allows for prompt engineering and failure analysis is essential for continued model development and deployment.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; • **Computing methodologies → Machine learning;**

Additional Key Words and Phrases: Large language models, RTL code generation, benchmarks

ACM Reference Format:
Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Trans. Des. Autom. Electron. Syst.* 30, 6, Article 91 (October 2025), 20 pages. https://doi.org/10.1145/3718088

This article would not have been possible without the generous help of NVIDIA Applied Deep Learning Research (ADLR), especially Teodor-Dumitru Ene, and the NVIDIA Inference Microservices (NIM) teams.
Authors' Contact Information: Nathaniel Pinckney, NVIDIA Corp, Austin, Texas, United States; e-mail: npinckney@nvidia.com; Christopher Batten, Cornell University, Ithaca, New York, United States; e-mail: cbatten@cornell.edu; Mingjie Liu, NVIDIA Corp, Austin, California, United States; e-mail: mingjiel@nvidia.com; Haoxing Ren, NVIDIA Corp, Austin, California, United States; e-mail: haoxingr@nvidia.com; Brucek Khailany, NVIDIA Corp, Austin, California, United States; e-mail: bkhailany@nvidia.com.

ACM 1084-4309/2025/10-ART91
https://doi.org/10.1145/3718088

1 Introduction

Applications of **large language models (LLMs)** to software coding have reached wide deployment, with examples such as GitHub Copilot [10]. Yet, applications of LLMs to hardware design are still in their infancy [5, 7]. Hardware code generation benchmarks have only been available since 2023, including RTLLM [21], VerilogEval [19], VeriGen [29, 30], and most recently RTL-Repo [3]. Despite this, LLM model releases have been extremely rapid. In this work, we survey the progress LLMs have made in the past year by evaluating newer LLMs than those tested in the original VerilogEval paper (published November 2023), including GPT-4o [27] and GPT-4 Turbo [26], open-source models like Llama3.1 [23], and domain-specific models such as RTL-Coder [20]. In short, we assess the latest state-of-the-art language models to determine the current frontier of LLM-based Verilog code generation while also evaluating the impact of prompt tuning. We find that recent open models are competitive with closed models and that prompt tuning varies considerably across models.

We also take the opportunity to release an improved version of VerilogEval to better align with instruction-tuned models and to encourage further prompt tuning research. While RTLLM benchmarked conversational specification-to-RTL generation performance, VerilogEval, VeriGen, and RTL-Repo are code completion benchmarks. Additionally, none of the benchmarks explore a model's generation performance using **in-context learning (ICL)** [6] examples, nor do they provide a detailed way to inspect the reasons for a model's failure.

This work aims to address these limitations by extending VerilogEval [19] (henceforth known as "VerilogEval v1") to support specification-to-RTL tasks in addition to the original code completion task. We also incorporate a variable number of ICL prompts and provide a robust failure classification mechanism to provide a more comprehensive evaluation framework for Verilog code generation tasks. The significance of these improvements is their potential to push LLM development forward for hardware design, through offering insights into model performance and the efficacy of prompt tuning, and to point out differences in generation quality across tasks. Even with similar problem statements and ICL examples, we find divergent responses by LLMs. This variability highlights the importance of understanding how different models respond to various prompts and contexts through the use of the benchmarks, providing granular failure feedback.

The following new features are part of the improved "VerilogEval v2" benchmark infrastructure:

(1) *Specification-to-RTL task support*: VerilogEval v1 only supported code completion tasks, such as used in Copilot [10], while many models are tuned and deployed as instruction-tuned models [35], with question-and-answer prompting.
(2) *In-context learning examples*: No ICL [6] examples were supported as part of the prompt in VerilogEval v1. Prompt tuning techniques, such as ICL, can improve LLM responses.
(3) *Failure classification*: VerilogEval v1 only reported pass/fail results of a benchmark problem and did not give fine-grained feedback on failures.
(4) *Makefile-based evaluation environment*: The original VerilogEval benchmark [19] used a monolithic dataset, whereas the proposed infrastructure uses a Makefile-based approach. This allows for easier scaling while sweeping evaluation settings across more models than the original benchmark, and easier human inspection of the dataset.

The improved VerilogEval benchmark is available publicly at https://github.com/NVlabs/verilog-eval.

2 VerilogEval v1 Revisited

The original VerilogEval [19] contains 156 problems adopted from questions and solutions on the HDLBits Verilog instructional website, picked based on clarity and diversity. The VerilogEval

dataset contains both VerilogEval-machine and VerilogEval-human problem descriptions. The former is based on GPT-3.5-generated descriptions of the solution RTL, while the latter are human-created problem descriptions from HDLBits. While both problem description sets are useful when evaluating an LLM's code generation capability, VerilogEval-human is most aligned with common code generation deployments, such as Copilot [10].

VerilogEval evaluated an LLM's performance by reporting the pass rate, specifically a *pass@k* metric meaning at least one sample passes among k samples. Because LLM responses will be non-deterministic at non-zero temperature, the LLM is sampled multiple times, and if at least one sample passes, the problem passes. If none of the k samples passes, then the problem fails. Formally *pass@k* is defined as follows:

$$pass@k := \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{1}$$

where $n \geq k$ represents the total number of trials for each problem, and c represents the number of trials that pass the functional check.

The original VerilogEval paper reported results with $k = 1, 5, 10$. This *pass@k* metric is useful to evaluate if given knowledge is available within an LLM, and thus beneficial for model development, but a typical LLM deployment for code generation (say, in an interactive copilot application [10]) will only sample for a response once. Therefore, for the evaluation of LLMs in this work, only *pass@1* is reported to mimic single-turn (single-query and -response) scenarios. However, we report two sets of model parameters: high temperature (T = 0.8, top_p = 0.95) and low temperature (T = 0.0, top_p = 0.01) sets. The high-temperature model parameters are identical to those used in [19]. For high temperature, we report *pass@1* across 20 samples (n = 20). In other words, we report how many of the 20 sampled responses pass the benchmark for each problem within the dataset. For low temperature (nearly equivalent to greedy sampling), where responses are generally deterministic, we report a single sample (n = 1).

For this study, we only evaluated models against VerilogEval-human to highlight the most useful LLM evaluation results. VerilogEval-machine, in comparison, can be overly descriptive compared to real-world code generation problems. The infrastructure was revised to more easily re-run a subset of the dataset (discussed in the next session), to better post-process the LLM response, and 14 problems had their descriptions or test benches revised to fix consistency or clarity issues. Beyond these changes, and minor white space differences, the dataset and prompts were kept the same as the study in [19]. In [19] the highest-achieving model was GPT-4 [25] with a *pass@1* pass rate of 43.5% and 60.0% for VerilogEval-human and VerilogEval-machine, respectively. GPT-3.5 exhibited lower pass rates of 26.7% and 46.7%, respectively.

We evaluate 14 publicly available LLMs against these VerilogEval-human code completion prompts:

— OpenAI GPT-4o [27]
— OpenAI GPT-4 Turbo (gpt-4-1106-preview) [26]
— OpenAI GPT-4 (gpt-4-0613) [25]
— Mistral AI Mistral Large [1]
— Meta Llama3.1 405B, 70B, and 8B [23]
— Meta Llama3 70B [23]
— Meta Llama2 70B [23]
— Meta CodeLlama 70B [22]
— Google CodeGemma 7B [13]

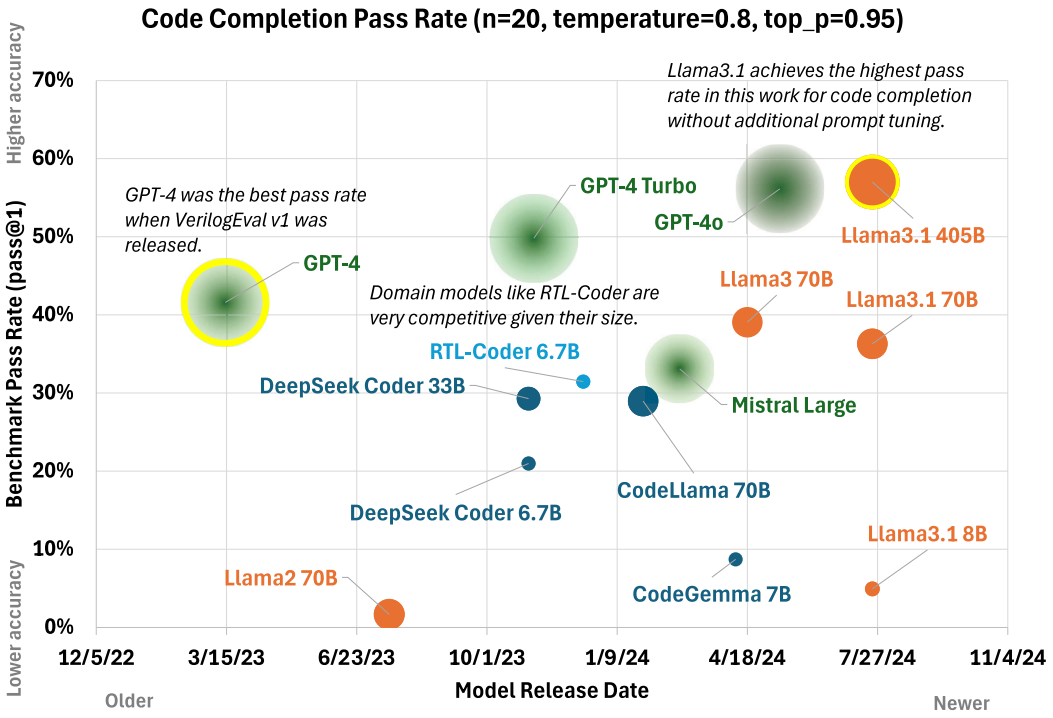


Fig. 1. Pass rate across recent large language models similar to VerilogEval v1 for *pass*@1. Green models are closed general-purpose models, orange are open general-purpose models, dark blue are coding-specific models, and light blue is an RTL-specific model.

— DeepSeek Coder 33B and 6.7B [14]
— RTL-Coder DeepSeek v1.1 6.7B [20]

The models are composed of a range of closed to open source, parameter sizes, and general purpose to specialized. Figure 1 shows the equivalent of VerilogEval-human *pass*@1 results for recent models, with pass rate on the y-axis and model release date on the x-axis. The data point size represents the approximate model size. Note that GPT-4, GPT-4 Turbo, GPT-4o, and Mistral Large have undisclosed sizes. The data points are green in color for undisclosed size, orange for general-purpose open models, dark blue for coding-specific models, and light blue for domain-specific (RTL code generation) models.

GPT-4 with our new infrastructure and adjusted prompts is slightly lower than previously measured in [19], at 41.6% instead of 43.5%. This can be attributed mostly to slight changes in prompt, such as whitespace (line breaks) and punctuation, and is in the measurement noise. Of the large models, GPT-4o (56.1%) and GPT-4 Turbo (49.8%) exceed GPT-4, while Llama3.1 405B goes further still with the best pass rate (57.0%) despite being an open model. Coding-specific models such as DeepSeek Coder 33B (29.3%) and CodeLlama 70B (29.0%) did well for their size at their respective times of release. Significant improvement is seen from Llama2 70B (1.7%) to Llama3 70B (39.1%), while Llama3.1 70B (35.3%) is slightly worse than its predecessor. However, as we shall see in the next section, prompt tuning can change the pass rate, significantly in some cases. Mistral Large (33.1%) and CodeGemma 7B (8.7%) appear to lag Verilog code generation tasks compared to their peers. Notably, RTL-Coder 6.7B (31.5%) performs almost as well as Llama3.1 70B (36.3%) while being an order of magnitude smaller, demonstrating the efficiency of smaller

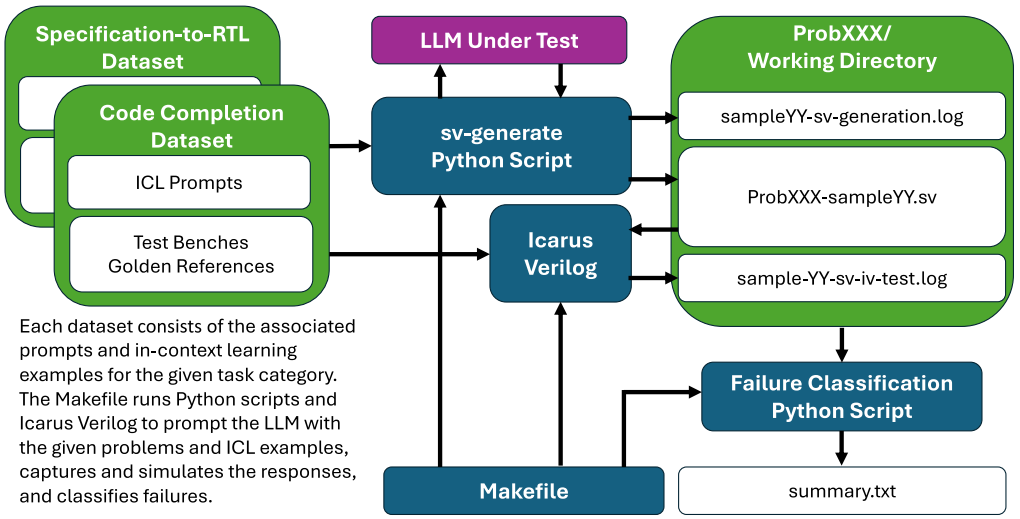


Fig. 2. Overview of VerilogEval v2 flow.

and cheaper domain-specific models. Overall, LLMs have demonstrated tremendous progress across model releases, especially open and domain-specific models.

An LLM's quality of result is not solely dependent on the model; prompt tuning can have a large impact on generated code quality. In the next section, we extend our new VerilogEval v2 benchmark to support two prompt tuning techniques: (1) in-context learning [6] and (2) changing the task type from code completion to specification-to-RTL. In-context learning has been demonstrated to have similar impact to training or tuning [2, 8, 15, 34], while many models are instruction-tuned, such that specification-to-RTL may be better aligned with how models are created.

3 VerilogEval v2 Improvements

The improved VerilogEval v2 flow is shown in Figure 2, which is substantially different than the flow from [19]. The original VerilogEval infrastructure features a monolithic `JSONL` file with the dataset and a Python script to evaluate, while the proposed v2 flow uses a Makefile-based approach. A Makefile parameter specifies which of multiple datasets is used for the evaluation, and each dataset contains the problem prompts, reference solutions, and in-context learning examples. Two datasets are included in the VerilogEval v2 repository, one for code completion and one for specification-to-RTL tasks. An evaluation Python script, similar to the one in [19], queries the LLM under test with prompts and contains problem descriptions along with in-context learning examples, if applicable. LLM responses are saved in a working directory specific to each problem. Icarus Verilog is then used to evaluate the generated SystemVerilog against the reference solution. Lastly, a failure classification script detects keywords in the Icarus Verilog output (both compile time and runtime) to classify failures. A summary of failures across problems is saved into a text file for human analysis.

3.1 Specification-to-RTL Task Support

The enhanced VerilogEval v2 benchmark supports both code completion and specification-to-RTL tasks to better match the instruction tuning [35] of recent models. The full 156-problem dataset from VerilogEval is converted into specification-to-RTL prompting in this work. Code completion

has the problem description in Verilog-compatible comments and always appends the module interface declaration to the end of the prompt, similar to how Copilot [10] is typically implemented in an **integrated development environment (IDE)**.

On the other hand, specification-to-RTL's prompt style is like a chatbot, with well-defined "Question" and "Answer" sections. The specification-to-RTL prompting is implemented in a manner similar to the **Mostly Basic Python Problems (MBPP)** benchmark [4] with `[BEGIN]` and `[DONE]` tags surrounding code blocks. Examples of these two styles can be found in Listings 1 and 2, with only the yellow highlighted code indicating the prompt styles.

3.2 Support for In-context Learning Examples

ICL was proposed by [6] to add examples of task questions and desired responses into the prompt context so that an LLM can better respond to a given task. ICL is implemented through simple Verilog code examples, tailored for both code completion (Listing 1) and specification-to-RTL tasks (Listing 2). The listings contain the one-shot examples used for both tasks, except line width and whitespace were adjusted for printing. The examples were selected to be short and simple, while including a full module (from declaration to `endmodule`).

Two additional examples for each task are also added to the infrastructure, as shown in Listings 3 and 4: a sequential incrementer (Listing 3) similar to the first one-shot example and a basic finite-state machine (Listing 4). The number of shots is parameterized and can easily be swept to determine the sensitivity of a model's pass rate as ICL examples are added to the prompt. One-shot includes only the combinational incrementer, two-shot adds the sequential incrementer, and three-shot includes all three examples in the context prompt.

Listing 1. The one-shot in-context learning example for code completion tasks. The highlighted code is the prompt style.

```
// Implement the Verilog module based on the
// following description. Assume that sigals
// are positive clock/clk triggered unless
// otherwise stated.
//
// The module should implement an incrementer
// which increments the input by one and
// writes the result to the output. Assume
// all values are encoded as two's complement
// binary numbers.
module TopModule
(
  input  logic [7:0] in_,
  output logic [7:0] out
);
  // Combinational logic
  assign out = in_ + 1;
endmodule
```

Listing 2. The one-shot ICL example for specification-to-RTL tasks. The highlighted code is the prompt style.

```
Question:
Implement a hardware module named TopModule
with the following interface. All input and
output ports are one bit unless otherwise
specified.

 - input  in_ (8 bits)
```

```
 - output out (8 bits)

The module should implement an incrementer
which increments the input by one and writes
the result to the output. Assume all values
are encoded as two's complement binary
numbers.

Enclose your code with [BEGIN] and [DONE].
Only output the code snippet and do NOT output
anything else.

Answer:
[BEGIN]
module TopModule
(
  input  logic [7:0] in_,
  output logic [7:0] out
);
  // Combinational logic
  assign out = in_ + 1;
endmodule
[DONE]
```

Listing 3. The two-shot in-context learning example for code completion tasks.

```
// Implement the Verilog module based on
// the following description. Assume that sigals
// are positive clock/clk triggered unless
// otherwise stated.
//
// The module should implement an 8-bit registered incrementer with an
// active-high synchronous reset. The 8-bit input is first registered and
// then incremented by one on the next cycle. The internal state should be
// reset to zero when the reset input is one. Assume all values are encoded
// as two's complement binary numbers. Assume all sequential logic is
// triggered on the positive edge of the clock.

module TopModule
(
  input  logic       clk,
  input  logic       reset,
  input  logic [7:0] in_,
  output logic [7:0] out
);
  // Sequential logic
  logic [7:0] reg_out;
  always @( posedge clk ) begin
    if ( reset )
      reg_out <= 0;
    else
      reg_out <= in_;
  end
  // Combinational logic
  logic [7:0] temp_wire;
  always @(*) begin
    temp_wire = reg_out + 1;
  end
```

```
  // Structural connections
  assign out = temp_wire;
endmodule
```

Listing 4. The three-shot in-context learning example for code completion tasks.

```
// Implement the Verilog module based on
// the following description. Assume that sigals
// are positive clock/clk triggered unless
// otherwise stated.
//
// Build a finite-state machine that takes as input a
// serial bit stream and outputs a one whenever the bit stream contains two
// consecutive one's. The output is one on the cycle _after_ there are two
// consecutive one's.
//
// The reset input is active high synchronous and should reset the
// finite-state machine to an appropriate initial state.
module TopModule
(
  input  logic clk,
  input  logic reset,
  input  logic in_,
  output logic out
);
  // State enum
  localparam STATE_A = 2'b00;
  localparam STATE_B = 2'b01;
  localparam STATE_C = 2'b10;
  // State register
  logic [1:0] state;
  logic [1:0] state_next;
  always @(posedge clk) begin
    if ( reset )
      state <= STATE_A;
    else
      state <= state_next;
  end
  // Next state combinational logic
  always @(*) begin
    state_next = state;
    case ( state )
      STATE_A: state_next = ( in_ ) ? STATE_B : STATE_A;
      STATE_B: state_next = ( in_ ) ? STATE_C : STATE_A;
      STATE_C: state_next = ( in_ ) ? STATE_C : STATE_A;
    endcase
  end
  // Output combinational logic
  always @(*) begin
    out = 1'b0;
    case ( state )
      STATE_A: out = 1'b0;
      STATE_B: out = 1'b0;
      STATE_C: out = 1'b1;
    endcase
  end
endmodule
```

Table 1. Types of Failures Supported by the Automatic Failure Classification

Failure Type	Example
Compile-time Failures	
Unable to Bind Wire/Reg "clk"	Clk is missing in interface ports list, such as if a code completion task does not specify a clock to be used yet the LLM used it in the generated code.
Unable to Bind Wire/Reg	Other port-related bind problems.
Explicit Cast Required	A datatype problem occurred, often with use of enums.
Module Missing	Typically indicates the modular declaration is missing from the generated code.
Sensitivity Problem	Sensitivity lists for always blocks are not defined properly.
Reg Declared as Wire	A wire is assigned to as a reg.
Syntax Error	General syntax errors in generated code.
General Compiler Error	Other compiler errors without specific classification.
Runtime Failures	
Reset Issue	Reset should be synchronous but is asynchronous.
Timeout	The simulation did not complete in reasonable time, indicating a sequential block does not have a correct implementation.
General Runtime Error	Other runtime errors that are not classified, including mismatched outputs.

3.3 Support for Failure Classification

Failures of LLM-generated responses are automatically classified by broad reasons for failure, including both Verilog compile-time errors and simulation runtime errors, such as incorrectly using a wire as a register, incorrect bit widths, and missing module interface definitions. This classification feature provides insight into the most common reasons for failures and how to mitigate poor code generation through prompt tuning. The classification is dependent on specific warnings and errors given by Icarus Verilog or the test harness. The failures are classified in Table 1.

Classifications were developed by human inspection of common failure modes across the code completion benchmark. For example, LLMs were observed frequently mixing up the use of registers and wires. Solutions in prompt tuning could vary: from adding prompt rules to only use wires on ports; to suggesting the use of SystemVerilog `logic` port types, obviating the immediate type confusion; to allowing the LLM to generate the interface entirely on its own (as in the case of specification-to-RTL, rather than code completion). By classifying failures, the impact of prompt changes on code generation performance can be directly observed and guided.

3.4 Other Infrastructural Improvements

The original VerilogEval benchmark contained all problems in a monolithic `JSONL` format. This is efficient to run but inefficient to inspect manually using a text editor. In the improved benchmark, each problem was split into a set of files, including problem prompts, module interfaces, and test benches. Autoconf [12] and GNU Make [11] were employed to target a model evaluation build directory to a specific evaluation target, including the LLM itself to run, number of shots, number of samples, task to complete, and other parameters. For each problem, a resulting problem evaluation directory is created containing a log of the LLM prompt/responses, generated Verilog file, and Icarus Verilog output log. This infrastructure allows for scalable sweeps through the use of Make's parallel run feature, helps to resume an evaluation run if it is interrupted, and allows for easy human inspection of the resulting collateral. A backwards-compatible mode with `JSONL` support is planned for VerilogEval v2.

4 VerilogEval v2 Evaluation

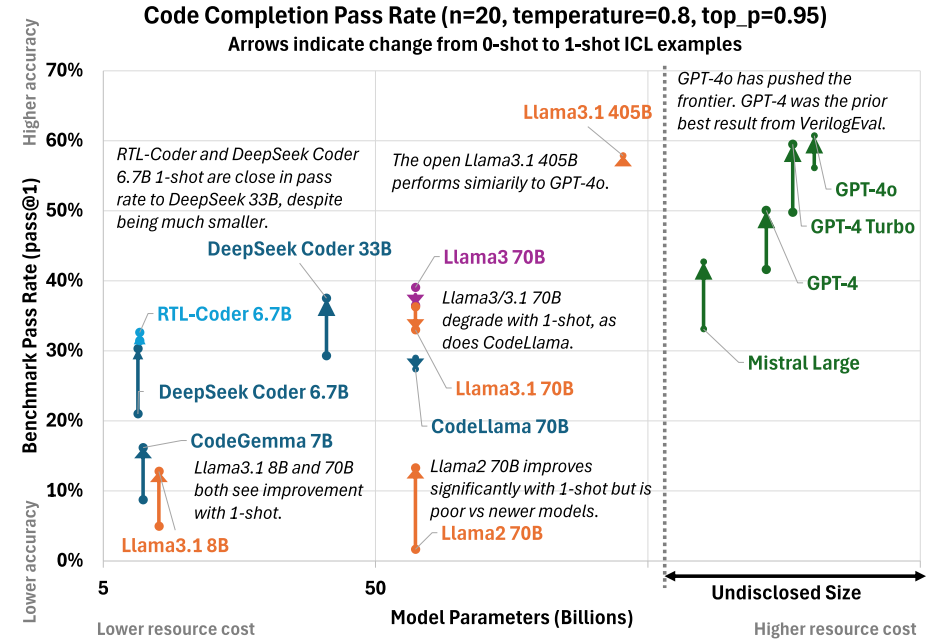
The graph in Figure 3 illustrates the performance of the recent LLMs on code completion and specification-to-RTL translation tasks, as measured by the benchmark pass rate (*pass*@1 in [19]). As in the previous section, model results were captured as both a 20-sample (n = 20) high-temperature (T = 0.8, top_p = 0.95) set and 1-sample (n = 1) low-temperature (T = 0.0, top_p = 0.01) set. Models are arranged along the x-axis by model size, with undisclosed model sizes on the right. The evaluation compares models with and without one-shot ICL examples, represented by arrows indicating the change in performance as one-shot examples are added. For code completion tasks, Llama3.1 405B initially achieves the highest pass rate in zero-shot, as previously shown in Figure 1. However, when one-shot is added, GPT-4o achieves the highest pass rate at approximately 61% from 56% in zero-shot, establishing the new state-of-the-art frontier. As both prompt configurations show, GPT-4o has robust improvement over GPT-4 for RTL generation tasks.

Llama3.1 405b established the new Pareto frontier at zero-shot (57.0%), demonstrating that open models have matched closed commercial models, but showed very little improvement in one-shot (57.9%). The older Llama3 70B (purple line) was included to demonstrate a clear counter-example to ICL's improving pass rate. While Llama3.1 generally improves with ICL examples, Llama3 70B declines in pass rate when the one-shot ICL example is added to the prompt, which will be discussed in detail in the next section. Among the smaller specialized models, RTL-Coder 6.7B showed an impressive pass rate of around 33%, while being much smaller than general-purpose models. RTL-Coder when originally sampled did not properly insert whitespace after `endmodule` statements and would often repeat code blocks. We modified our post-process script that extracts the Verilog code from the response to match the post-processing in RTL-Coder's evaluation scripts [16], and Figure 3's RTL-Coder results are shown using this modified response extraction. This post-processing is also used across the other models as well. DeepSeek Coder 6.7B (30.3%) nearly reaches the pass rate of RTL-Coder (32.6%) in code completion when an ICL example is added.

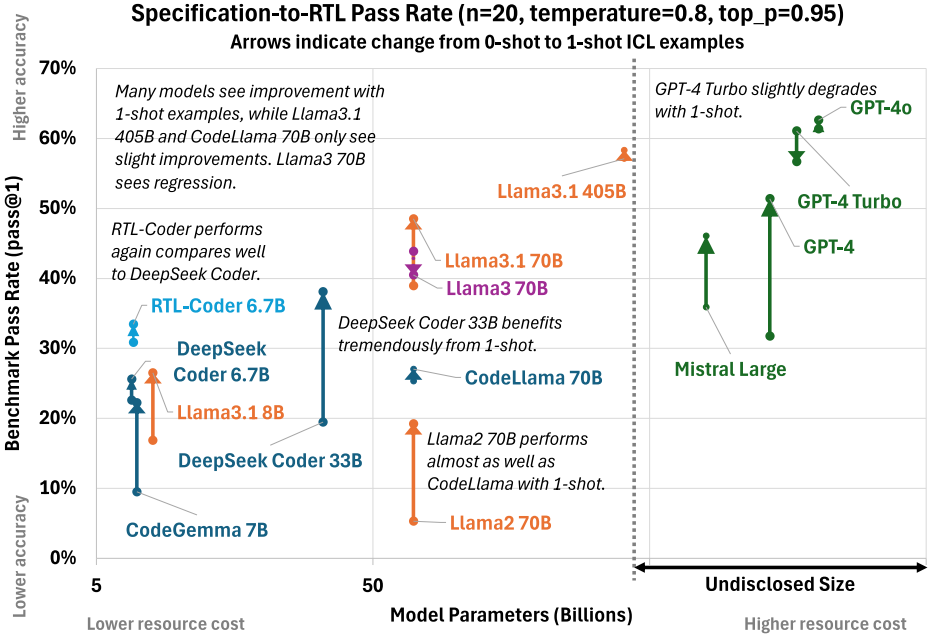
Specification-to-RTL task results showed generally similar pass rates compared to code completion, with some exceptions. GPT-4 Turbo showed noticeable pass rate improvement in code completion tasks but some degradation in spec-to-RTL. Mistral Large showed improvements in both tasks. Llama3.1 70B and CodeGemma 7B saw much larger improvements in specification-to-RTL when adding one-shot ICL. In particular, at this prompt tuning configuration, Llama3.1 70B exceeds the highest pass rate achieved by Llama3 70B across all configurations presented, despite Llama3.1 starting at a lower pass rate than Llama3 across both tasks at zero-shot. In Llama3.1 405B across both tasks, adding an ICL example made little difference in pass rate. Llama3 70B saw a decline with one-shot in spec-to-RTL, more so than in code completion. As with code completion, RTL-Coder 6.7B (33.5%) maintained a lead over DeepSeek Coder 6.7B in spec-to-RTL (25.6%).

The full results are shown in Table 2 and include both n = 20 (20 samples, temperature = 0.8, top_p = 0.95) from Figure 3 along with deterministic n = 1 (1 sample, temperature = 0.0, top_p = 0.01). Some models performed notably better in spec-to-RTL than code completion, such as Llama3.1 70B one-shot with 48.5% in spec-to-RTL versus 39.0% in code completion. This variability underscores the importance of tailored prompt tuning and the potential of ICL to enhance code generation performance in certain models.

Overall, larger models generally achieve higher pass rates, though resource costs and model-specific responses to ICL examples vary significantly. Within the context of VerilogEval, GPT-4o and Llama3.1 405B have become clear leaders for the highest-achieved pass rates, demonstrating that open models (Llama3.1 405B) have reached parity with closed models. Additionally, smaller (70B) open models have become competitive with last year's larger closed models. Domain-specific models (RTL-Coder) are also competitive in some scenarios at a much smaller size.



(a) Code Completion Task



(b) Specification-to-RTL Task

Fig. 3. Pass rate across recent large language models. Green models are closed general-purpose models, orange are open general-purpose models, dark blue are coding-specific models, and light blue is an RTL-specific model. Purple is the older Llama3 70B to demonstrate a large degradation due to ICLs.

Table 2. VerilogEval Pass Rates of Recent Large Language Models

Model Name	Model Size	License	Type	Task: Code Completion				Task: Specification-to-RTL			
In-context Learning Examples:				Zero-shot		One-shot		Zero-shot		One-shot	
Temperature:				T = 0	T = 0.8	T = 0	T = 0.8	T = 0	T = 0.8	T = 0	T = 0.8
GPT-4o [27]	Undisclosed	Closed	General	59.0%	56.1%	62.8%	60.7%	62.5%	61.4%	65.1%	62.6%
GPT-4 Turbo [26]	Undisclosed	Closed	General	53.9%	49.8%	59.6%	59.5%	59.6%	61.1%	56.4%	56.7%
GPT-4 [25]	Undisclosed	Closed	General	42.3%	41.6%	51.3%	50.1%	32.0%	31.7%	48.7%	51.4%
Mistral Large [1]	Undisclosed	Closed	General	34.0%	33.1%	44.2%	42.7%	37.5%	35.9%	48.7%	46.0%
Llama3.1 [23]	405B	Open	General	56.4%	57.0%	59.6%	57.9%	57.2%	57.1%	57.9%	58.3%
Llama3.1 [23]	70B	Open	General	35.3%	36.3%	34.0%	33.0%	42.8%	39.0%	48.0%	48.5%
Llama3 [23]	70B	Open	General	37.8%	39.1%	36.5%	36.5%	40.8%	43.9%	39.5%	40.5%
Llama2 [23]	70B	Open	General	1.3%	1.7%	15.4%	13.3%	4.6%	5.3%	17.8%	19.2%
CodeLlama [22]	70B	Open	Coding	37.2%	29.0%	41.7%	27.4%	34.9%	25.3%	41.5%	27.0%
DeepSeek Coder [14]	33B	Open	Coding	25.0%	29.3%	42.3%	37.5%	21.7%	19.5%	40.1%	38.1%
Llama3.1 [23]	8B	Open	General	2.6%	4.9%	10.9%	12.8%	19.1%	16.8%	27.6%	26.5%
CodeGemma [13]	7B	Open	Coding	8.3%	8.7%	19.9%	16.2%	6.6%	9.5%	24.3%	22.2%
DeepSeek Coder [14]	6.7B	Open	Coding	24.4%	21.0%	33.3%	30.3%	29.6%	22.6%	27.6%	25.6%
RTL-Coder [20]	6.7B	Open	Verilog RTL	35.9%	31.5%	37.2%	32.6%	36.8%	30.9%	34.9%	33.5%

Number of samples $n = 1$ when $T = 0$ and $n = 20$ when $T = 0.8$.

5 Impact of ICL on Pass Rates and Failures

As demonstrated in the previous section, ICL examples improve model generation accuracy in some conditions but degrade accuracy in others. ICL impact bears further investigation to better understand strategies to apply prompt tuning.

5.1 Increased In-context Learning Examples

Higher-shot ICL runs were conducted for four models across parameter size classes: GPT-4o, Llama3.1 70B, Llama3 70B, and RTL-Coder 6.7B. Pass rates of these four models for the two tasks across zero-shot to three-shot are shown in Figure 4. The figure highlights the varying impact of ICL examples on different models and tasks, emphasizing the potential benefits of task-specific tuning and the necessity of providing contextual examples to enhance model outputs. Notably, GPT-4o exhibits stable and high performance across all ICL example counts of at least one-shot, maintaining a pass rate of 55% to 63%. In contrast, Llama3 70B demonstrates divergent trends: its spec-to-RTL performance improves from 40% to nearly 50% with more ICL examples, whereas its code completion performance declines from 40% to just above 30%. Llama3.1 70B achieves even better pass rates with ICL examples as compared to Llama3 for spec-to-RTL and is fairly stable in code completion. RTL-Coder shows stability from zero-shot to three-shot in both code completion and spec-to-RTL, with only very little improvements in the latter case. To understand how specific ICL examples can influence response pass and failure, we will look at specific case examples in the next section across Llama models.

5.2 Case Study: Problem 9 and Problem 34

ICL may improve model pass rate through better aligning the model response with the task at hand if the ICL example is well chosen. To better understand the impact of in-context learning on Verilog code generation pass rate, we consider two problems from the dataset: Problem 9 and Problem 34. Problem 9 is a simple combinational design question that asks: "`A ''population count'' circuit counts the number of '1's in an input vector. Build a population count circuit for a 3-bit input vector.`" In contrast, Problem 34 is a simple sequential design question: "`Create 8 D flip-flops. All DFFs should be triggered by the positive edge of clk.`"

In zero-shot (no ICL examples are given) for Problem 9, none of the 20 high-temperature samples pass on Llama3 70B. The failure case is an "`Unable to Bind Wire/Reg 'clk'`" error given by

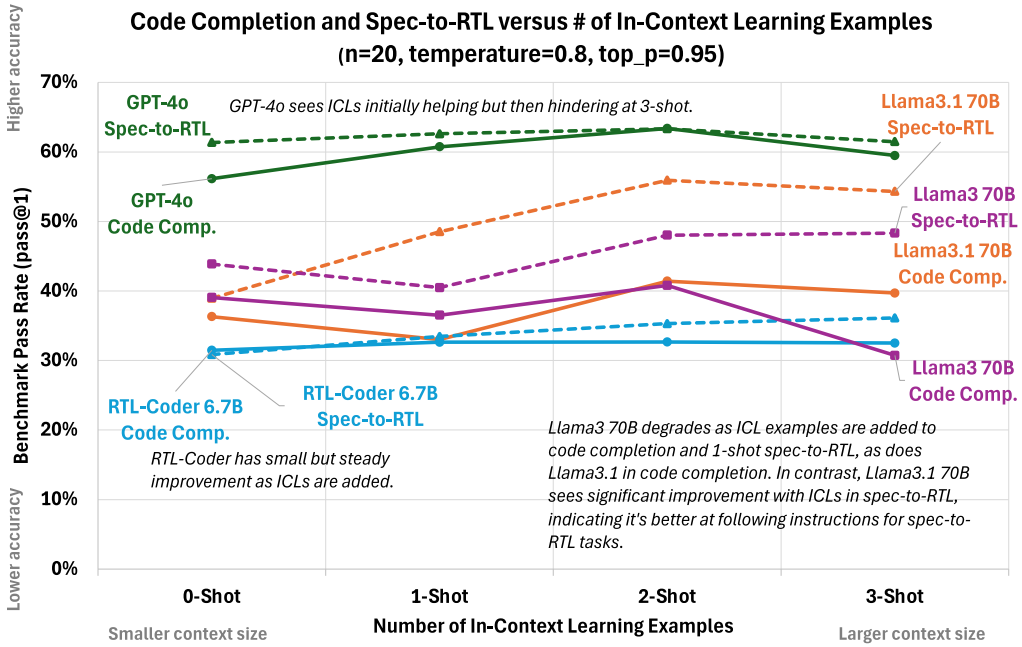


Fig. 4. Pass rate of three models for code completion and specification-to-RTL tasks, with zero-shot to three-shot in-context learning examples. Solid lines are code completion and dashed lines are spec-to-RTL.

Icarus Verilog, in which the LLM tries to use a non-existent clock signal. In effect, the LLM is confusing a combinational design problem for a sequential design problem. An example failing response to Problem 9 is given in the listing below:

```
module TopModule (
  input [2:0] in,
  output [1:0] out
);
  reg [1:0] out_reg;
  always @(posedge clk) begin
      out_reg = (in[2] == 1'b1) + (in[1] == 1'b1) + (in[0] == 1'b1);
  end
  assign out = out_reg;
endmodule
```

In this example, the clock is not present, which causes the error. In code completion tasks, the LLM does not have the flexibility to define the port types (`wire` or `reg`) but attempts to use the port as an incorrect type. When the one-shot example shown in Listing 1, which demonstrates a simple combination incrementer, is added to the prompt as an ICL example, the pass rate of Problem 9 increases substantially to 65% (13 of the 20 samples pass). A typical Llama3 70B passing response to this problem is:

```
module TopModule (
  input [2:0] in,
  output [1:0] out
);
  // Combinational logic
```

```
  assign out = (in[2] + (in[1] + in[0]));
endmodule
```

In effect, the LLM response has been better aligned to handle combinational design problems, at least for a trivial example like Problem 9. Of the remaining seven failures in one-shot, three are missing an `endmodule` keyword that terminates the module definition, and four try to use the output port as a register instead of a wire (`Reg Declared as Wire`). This suggests that even with the ICL example, the LLM does not understand where it should terminate its response and the distinction between wires and registers.

The added one-shot example focuses on combinational design, yet Problem 34 is a sequential design. Initially, the 20-sample pass rate of Problem 34 is 100% in zero-shot but drops precipitously to 5% in one-shot. Thus, a question arises of whether sequential design problems are improved in one-shot at the expense of combinational design problems. However, on inspection, all failing responses are due to a missing `endmodule` and the above-mentioned confusion on sequential versus combinational design. An example failing response for Problem 34 is, where an `endmodule` keyword is missing:

```
module TopModule (
  input clk,
  input [7:0] d,
  output reg [7:0] q
);

always @(posedge clk)
  q <= d;
```

Adding another ICL example in two-shot (Listing 3, a combinational design example) increases the Problem 9 pass rate to 100% in two-shot and three-shot, eliminating both of the above-mentioned missing `endmodule` and reg versus wire confusion demonstrated for Problem 9. However, only 50% of the 20 sample responses pass for Problem 34 in two-shot, with all failing cases missing an `endmodule`. One culprit may be the confusion with `begin` and `end` block start and end keywords. All failing cases use `begin` and `end` for the sequential `always` block, while in the passing cases `begin` and `end` are omitted.

Adding one more ICL example in three-shot (Listing 4, an FSM design example), Problem 34 gets even worse: 0% of the cases pass. All cases use `begin` and `end` while omitting `endmodule`. Ostensibly, the multiple always of Listing 4 caused the LLM to over-emphasize the `always` blocks with `begin` and `end` but did not learn how to correctly use `endmodule`. However, Problem 9 maintains a 100% pass rate in three-shot.

While the code completion task by nature constrains the port types (`reg` or `wire`), specification-to-RTL allows the full interface specification to be given in the response. This leads to a higher zero-shot pass rate on Llama3, with a 75% pass rate on Problem 9 (compared to 0% in code completion) and a 100% pass rate on Problem 34 (compared to 65%). However, in one-shot the spec-to-RTL pass rate drops to 0% on Problem 9 because it picks up a slightly different input port name (`in_`) used in the ICL example that is not present in the prompt. Problem 34 is unaffected and is successful across all samples. Increasing ICL examples to two-shot fully recovers and surpasses zero-shot to 100% pass rates across both problems. Similarly, three-shot has full 100% pass rates on both. Overall, specification-to-RTL tasks perform better than code completion for these problems on Llama3 70B.

Broadening the results across older and newer generations of the Llama 70B models on code completion, Llama2 70B only has a 0% and 15% pass rate for Problems 9 and 34, respectively, with an assortment of failures. In one-shot, Problem 9 shows no improvement, while Problem 34 increases

to 30%. Code sections are often repeated in the responses, causing failures and demonstrating poor instruction following. Adding additional ICL examples continues to show no improvement, with Problem 9 never passing in two- or three-shot and Problem 34 regressing to 15% in two-shot and 5% in three-shot.

The newer Llama3.1 70B in zero-shot fails all Problem 9 samples, from a mix of failures including using a non-existent clock and redefining the output port as a register, but passes all Problem 34 samples. Adding one-shot, Problem 9 improves to a 75% pass rate and Problem 34 maintains a 100% pass rate across all 20 samples. Interestingly, all five failures in Problem 9 are due to bad combinational logic for the counting task, causing mismatch against the golden reference in simulation, and not compile-time errors. In particular, the Boolean expressions derived for population counting are incorrect.

Adding the Listing 3 ICL example to the prompt in two-shot to Llama3.1 70B code completion reduces Problem 9 pass rates to 60% and causes all sample responses on Problem 34 to fail. In this case, the failure is consistent inclusion of an English natural language explanation of the model being created, causing obvious syntax issues. None of the Problem 9 failures feature this failure mode. Increasing ICL examples to three-shot completely clears up the Problem 34 failures and returns Problem 9 to a 75% pass rate with only Boolean logic failures. As we can see for Llama3.1 70B, adding ICL examples can generally improve results, but sometimes wrong behavior will be learned, as in the case of two-shot.

The much larger Llama3.1 405B exhibited 100% 20-sample pass rates for Problem 9 and Problem 34 in all ICL shot scenarios, understanding these basic problems well. Thus, the exact impact of prompt tuning on a model can be very model dependent in terms of model parameter size, lineage, and generation.

5.3 Aggregate Failure Analysis

Figure 5 employs the new failure classification feature of the improved benchmark infrastructure to illustrate the number and types of failures encountered by different models across various numbers of ICL examples. The y-axis represents the number of failures, with lower values indicating better pass rates. Each bar is segmented to show different categories of errors, with orange shades representing compiler errors and blue shades representing runtime errors. The figure is divided into three sections for three generations of Llama 70B models, highlighting the numbers and types of failures across zero-shot to three-shot ICL examples. As compiler errors will be flagged and mask runtime errors (since code that does not compile never runs), the bars on the graph are best read from bottom to top. A reduction in runtime errors for the same total bar height indicates that compiler errors have displaced runtime errors. This layering effect should be considered when interpreting the improvements or degradations in model performance as additional ICL examples are introduced.

Llama2 70B initially improves in code completion from zero-shot to one-shot but then degrades as more examples are added. The "`Reg Declared as Wire`" error, which confuses wire output ports with registered output ports, as discussed in the last section, is especially pronounced at three-shot. In comparison, specification-to-RTL greatly improves compile-time errors, including wire and registered output confusion, even at one-shot. This suggests that specification-to-RTL has a naturally higher pass rate because of the flexibility of the LLM to define the interface types. However, runtime errors remain as the LLM is unable to solve the dataset problems with correct logic.

Llama3 and Llama3.1 70B both exhibit a similar pattern of code completion versus specification-to-RTL, with "`Reg Declared as Wire`" errors mostly eliminated in high ICL shot cases. However, only Llama3.1 70B improves substantially and consistently as ICL examples are added. It's also

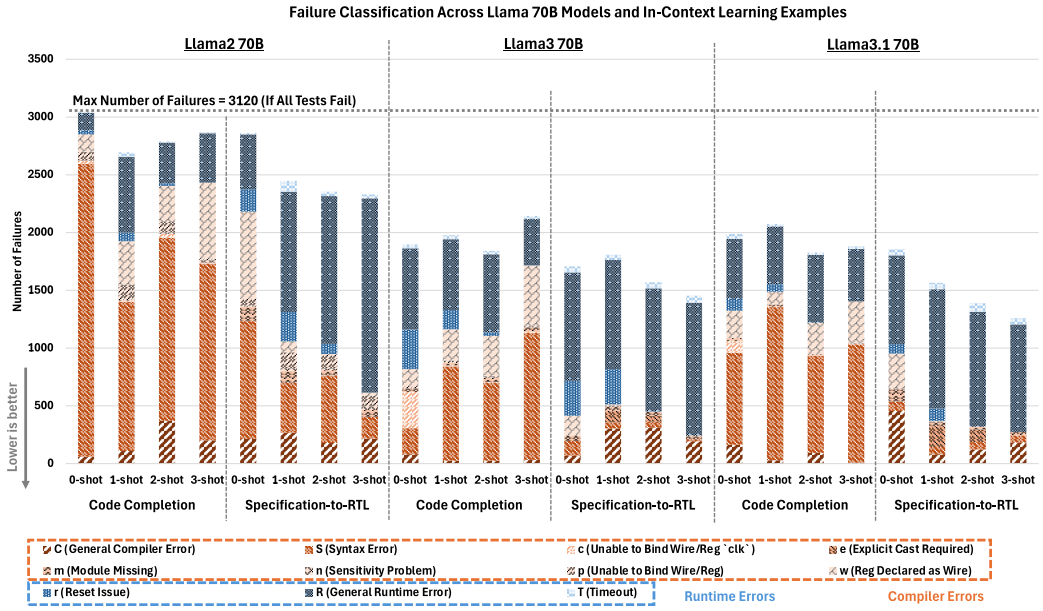


Fig. 5. Failure classification for Llama2 70B, Llama3 70B, and Llama3.1 70B models with zero-shot to three-shot ICL examples across the two tasks. Orange coloring indicates compiler errors, while blue indicates runtime issues.

notable that general syntax errors are much reduced in specification-to-RTL as compared to code completion. This is true even in zero-shot cases, although the overall failure rate is similar. In the context of code completion in Llama3 70B, the inclusion of ICL examples has a tendency to increase the number of compile-time failures, whereas the failure rate remains relatively stable with ICL examples in Llama3.1 70B.

The results emphasize the need for careful tuning of ICL examples to optimize results. While ICL can help correct certain types of mistakes, it can also introduce new issues, leading to similar or even worse performance. In addition to the failure classification feature capturing high-level counts of types of failures across different models and prompting settings, it also allows for detailed inspection on a problem-by-problem basis within a run. This granular analysis helps identify whether specific problems or categories of problems have systematic types of failures. Such insights can guide more careful tuning of prompts across the benchmark, leading to more effective and targeted improvements in model performance. A careful analysis of the problem categories within VerilogEval and comparative failure counts could help find the best ICL examples to use for a given model.

6 Future Work

Since its release last year, VerilogEval has been commonly used in state-of-the-art LLM Verilog code generation research, with over 100 citations.[1] However, application of VerilogEval is quickly becoming limited due to the low complexity of the dataset problems, especially when LLM deployments move beyond single-turn prompts and responses and into multi-turn flows with feedback, as we shall discuss below. Additionally, hardware design extends far beyond specification-to-RTL

[1]As reported by Google Scholar.

code generation tasks, and benchmarks need to cover many other tasks for LLM-based hardware design automation to continue.

6.1 Agent-based Code Generation

Moving beyond single-turn prompt response to agent-based approaches is key to solving more complex tasks. LLM-based *agentic* approaches are typically composed of a high-level planning agent (a dedicated prompt and model pair that is optimized for high-level planning) to enumerate tasks to the prompted problem and dedicated expert LLMs to implement said tasks. Additionally, there may be LLMs to integrate and combine disparate responses, rank the best responses, and detect if a given problem is solved or if instead the solution must be refined further. These dedicated LLMs may use the same LLM model but with different prompting or context, or models may be heterogeneous to efficiently solve targeted tasks.

Recent agent-based Verilog code generation approaches include RTLFixer [32], VeriAssist [18], AIvril [33], MAGE [37], PromptV [24], and VerilogCoder [17]. RTLFixer employs RAG and a thought–action–observation loop to resolve syntactical issues found during code compilation, improving VerilogEval-human *pass*@1 by 10%. VeriAssist applies multi-turn, chain-of-thought reasoning to fix both syntactical and functional errors in generated code and improves VerilogEval-human *pass*@1 by about 7%. AIvril includes a coding agent to generate code and a review agent to analyze compilation and simulation errors. The review agent provides feedback to the coding agent to revise the code. Their approach improves *pass*@1 by 14% to a total 65% on VerilogEval when using gpt-4o. PromptV employs code and testbench generation agents, code and testbench learner agents, and a single teacher agent to suggest errors and fixes to the learners based on simulation results, achieving 80% *pass*@1.

VerilogCoder [17] and MAGE [37] achieve some of the highest pass rates of agent-based approaches. VerilogCoder includes a planning agent, a plan verification assistant, a Verilog engineer agent, and a Verilog verification assistant in its multi-agent frameworks. The task planner breaks down the natural language prompt into various subtasks to implement, and the plan verification assistant reviews whether the tasks were implemented properly. This approach achieves a 94% *pass*@1 rate on VerilogEval, far exceeding the roughly 63% pass rate shown from state-of-the-art models in this work. However, VerilogCoder achieves this high pass rate by also having access to testbenches, simulation, and waveforms, so it can automatically debug a failing case until it passes. This is far more data and computing resources than what is given to models in single-turn, and individual models still have much room for improvement. Even more recently, MAGE leverages multiple agents, including a testbench agent, RTL agent, judge agent, and debug agent. Multiple RTL candidates are generated, ranked, and refined iteratively. MAGE's approach achieves a 95% pass rate on VerilogEval.

Agentic approaches mimic the implement-debug cycle that human designers employ. Additionally, real-world design specifications are far from perfect and often have bugs and ambiguities themselves. LLM-based agent flows will be essential in solving future real-world hardware design problems that move beyond trivial toy examples. Benchmarks must rise to meet these complexity needs and continue to aid in pushing the frontier by presenting realistic design problems that are challenging for AI agents to solve. VerilogEval cannot meet this challenge, as the dataset is already solved for VerilogCoder and MAGE.

The construction of future benchmarks to address the needs demanded by agentic flows will not be trivial: realistic and complex design problems are needed, and they must not be overly descriptive but instead allow for reasoning and flexibility that a human designer would be afforded. This may include an assumed set of base knowledge on traditional digital design structures and best practices but afford microarchitectural decision-making. An overly descriptive prompt that

defines every wire or flip-flop will, at best, be a poor natural language proxy for RTL. Instead, benchmarks should include accurate behavioral specifications and design goals and allow an LLM agent to approach generating a solution within reasonable decision space bounds.

6.2 Related Hardware Design Tasks

Hardware design is not limited to only RTL code generation, and many areas of hardware design would benefit from LLM enhancement within a tool flow. Even for RTL code generation, agentic approaches will demand specialized models or prompts for targeted tasks. This includes, but is not limited to, testbench creation [24, 36], assertion generation [28], documentation generation, debugging [31, 32], code review, analyzing consistency of specifications, correspondence of the testbench or RTL to test plan or specification, microarchitectural optimization, Q&A, and many more [9]. However, overall, benchmark development for frontend hardware tasks beyond code generation is still early, and future benchmarks should strive to thoroughly and systematically evaluate many tasks.

There are fundamental differences between RTL and testbench code generation that must be addressed by benchmarks, models, and agents. For instance, Verilog testbench code is not constrained by the synthesizable subset of Verilog required for RTL. Moreover, verification code often adheres to coding conventions distinct from those used for RTL within an organization. Evaluation metrics must extend beyond mere syntactical and functional correctness. For verification code, key metrics include coverage of the RTL under test, whereas for RTL, the primary focus is on quality metrics such as power, performance, and area.

In essence, assessing LLMs and agents solely on their language capabilities is insufficient for hardware design. Tackling complex, high-level hardware design tasks necessitates breaking them down into smaller, manageable subtasks. This decomposition is particularly suited to agent-based approaches that leverage specialized LLMs for specific subtasks. Identifying and addressing weaknesses in models for these subtasks requires the development of additional benchmarks tailored to these categories.

7 Conclusions

Much improvement has been observed in LLMs for hardware code generation since VerilogEval v1 [19] was released. Furthermore, the enhanced VerilogEval v2 benchmark proposed in this work provides a more robust framework for evaluating the performance of LLMs on digital hardware code generation tasks. Llama-3.1 405B and GPT-4o have both pushed the state of the art as open and commercial models, while domain-specific models such as RTL-Coder 6.7B and DeepSeek Coder 6.7B have offered impressive pass rates for their parameter size.

When evaluated on zero-shot ICL for code completion, Llama3.1 405B outperforms GPT-4o in the equivalent of *pass*@1 on the VerilogEval-human benchmark. However, as in-context examples are added to the prompt, GPT-4o achieves parity with Llama3.1 405B, though Llama3.1 benefits from being an open model. Most models showed performance improvements with the addition of ICL examples. However, exceptions were observed, such as the degradation of Llama3 70B's performance from zero-shot to one-shot. Transitioning the benchmark task from code completion to specification-to-RTL generally yielded better results, as this approach grants models greater flexibility in defining interfaces. These findings highlight the critical role of task-specific tuning in enhancing code generation accuracy.

The improved benchmark infrastructure, including the new failure classification feature, provides more in-depth insights into the types of errors encountered by different models. For example, Llama3 70B frequently encounters `endmodule` missing errors during code completion, which careful prompt tuning or model alignment may be able to fix. The ability to classify and inspect failures

on a problem-by-problem basis is critical for understanding and mitigating poor code generation, leading to more effective and targeted improvements in LLM performance for digital hardware code generation.

In the future, the research community would benefit from digital hardware benchmarks further expanded to include more tasks beyond RTL code generation representative of the digital hardware design flow. The enhanced VerilogEval v2 benchmark in this work is meant to be a step toward facilitating additional task support on top of a common set of design problems that allows for a more comprehensive assessment of model performance for hardware design. While the application of LLMs to hardware design is still in its infancy, models and generative AI techniques are quickly becoming capable of overcoming the code generation problems in the VerilogEval benchmark suite. It is imperative that as models and agents become more powerful, benchmark complexity increases to continue pushing technological development and sophistication in generative AI for hardware design.

References

[1] Mistral AI. 2024. Au Large. https://mistral.ai/news/mistral-large/ Section: news.

[2] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2023. What Learning Algorithm Is In-context Learning? Investigations with Linear Models. arXiv:2211.15661 [cs.LG]. https://arxiv.org/abs/2211.15661

[3] Ahmed Allam and Mohamed Shalan. 2024. RTL-Repo: A Benchmark for Evaluating LLMs on Large-scale RTL Design Projects. arXiv:2405.17378 [cs.LG]

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[5] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD '23)*. IEEE. https://doi.org/10.1109/mlcad58807.2023.10299874

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models Are Few-shot Learners. arXiv:2005.14165 [cs.CL]

[7] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *Computing Research Repository (CoRR)* arXiv:2305.14019 (May 2023).

[8] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. 2023. Why Can GPT Learn In-context? Language Models Implicitly Perform Gradient Descent as Meta-optimizers. arXiv:2212.10559 [cs.CL]. https://arxiv.org/abs/2212.10559

[9] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. GPT4AIChip: Towards next-generation AI accelerator design automation via large language models. In *International Conference on Computer-aided Design (ICCAD '23)*.

[10] GitHub. 2021. GitHub Copilot · Your AI Pair Programmer. https://github.com/features/copilot/

[11] GNU. 1988. Make—GNU Project—Free Software Foundation. https://www.gnu.org/software/make/

[12] GNU. 1991. Autoconf—GNU Project—Free Software Foundation. https://www.gnu.org/software/autoconf/

[13] Google. [n. d.]. google/codegemma-7b · Hugging Face. https://huggingface.co/google/codegemma-7b

[14] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE]

[15] Roee Hendel, Mor Geva, and Amir Globerson. 2023. In-context Learning Creates Task Vectors. arXiv:2310.15916 [cs.CL]. https://arxiv.org/abs/2310.15916

[16] hkust zhiyao. [n. d.]. hkust-zhiyao/RTL-Coder. https://github.com/hkust-zhiyao/RTL-Coder. original-date: 2023-11-20T13:01:13Z.

[17] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. VerilogCoder: Autonomous Verilog coding agents with graph-based planning and Abstract Syntax Tree (AST)-based waveform tracing tool. arXiv preprint. arXiv:2408.08927 (2024).

[18] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. 2024. Towards LLM-powered Verilog RTL Assistant: Self-verification and Self-correction. arXiv:2406.00115 [cs.PL]. https://arxiv.org/abs/2406.00115

[19] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv:2309.07544 [cs.LG]
[20] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-source Dataset and Lightweight Solution. arXiv:2312.08617 [cs.PL]
[21] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An Open-source Benchmark for Design RTL Generation with Large Language Model. arXiv:2308.05345 [cs.LG]
[22] Meta. [n. d.]. meta-llama/CodeLlama-70b-Instruct-hf · Hugging Face. https://huggingface.co/meta-llama/CodeLlama-70b-Instruct-hf
[23] Meta. 2024. meta-llama/llama-models. https://github.com/meta-llama/llama-models. Original-date: 2024-06-27T22:14:09Z.
[24] Zhendong Mi, Renming Zheng, Haowen Zhong, Yue Sun, and Shaoyi Huang. 2024. PromptV: Leveraging LLM-powered Multi-agent Prompting for High-quality Verilog Generation. arXiv:2412.11014 [cs.LG]. https://arxiv.org/abs/2412.11014
[25] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[26] OpenAI. 2023. *New Models and Developer Products Announced at DevDay.* https://openai.com/index/new-models-and-developer-products-announced-at-devday/
[27] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/
[28] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. 2021. AutoSVA: Democratizing formal verification of RTL module interactions. In *Design Automation Conference (DAC '21).*
[29] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated Verilog RTL code generation. In *Design, Automation, and Test in Europe (DATE '23).*
[30] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 29, 3 (Apr 2024), 1–31.
[31] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. AutoChip: Automating HDL generation using LLM feedback. *Computing Research Repository (CoRR).* arXiv:2311.04887 (Nov 2023).
[32] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2023. RTLFixer: Automatically fixing RTL syntax errors with large language models. *Computing Research Repository (CoRR).* arXivv:2311.16543 (Nov 2023).
[33] Mubashir ul Islam, Humza Sami, Pierre-Emmanuel Gaillardon, and Valerio Tenace. 2024. AIvril: AI-driven RTL Generation with Verification In-the-loop. arXiv:2409.11411 [cs.AI]. https://arxiv.org/abs/2409.11411
[34] Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. 2023. Transformers Learn In-context by Gradient Descent. arXiv:2212.07677 [cs.LG]. https://arxiv.org/abs/2212.07677
[35] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating Instruction-tuned Large Language Models on Code Comprehension and Generation. arXiv:2308.01240 [cs.CL]
[36] Zixi Zhang, Greg Chadwick, Hugo McNally, Yiren Zhao, and Robert Mullins. 2023. LLM4DV: Using large language models for hardware test stimuli generation. *Computing Research Repository (CoRR).* arXiv:2310.04535 (Oct 2023).
[37] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2024. MAGE: A Multi-agent Engine for Automated RTL Code Generation. arXiv:2412.07822 [cs.AR]. https://arxiv.org/abs/2412.07822

Received 30 September 2024; revised 27 December 2024; accepted 3 February 2025