

Politecnico di Torino

Politecnico di Torino

Corso di Laurea

A.a. 2025/2026

Graduation Session October 2026

Agentic Workflows for RTL code Generation

Supervisors:

Alessio Burrello
Daniele Jahier Pagliari
Carlo Marra
Giovanni Pollo
Luca Benfenati
Aofeng Shen

Candidate:

Lisa Giacobazzi

Table of Contents

List of Figures IV

1 Introduction 1

2 Background 3
2.1 Hardware Design and Verification Fundamentals . . . 3
2.1.1 RTL Design . . . 4
2.1.2 Testbenches and Simulation . . . 4
2.1.3 Limitations . . . 6
2.2 Large Language Models . . . 6
2.2.1 Training Process . . . 6
2.2.2 Inference . . . 8
2.2.3 Transformer Architecture . . . 9
2.2.4 Applications to Code Generation . . . 9
2.3 Agentic Systems . . . 11
2.3.1 Common Architecture . . . 11
2.3.2 OpenCode . . . 12
2.3.3 Claude Code . . . 16
2.3.4 Claw Code . . . 17
2.3.5 Chipmind Agents . . . 19
2.4 Motivation . . . 19

3 Related Works 22
3.1 Hardware Design Benchmarks . . . 22
3.1.1 RTLLLM . . . 23
3.1.2 VerilogEval . . . 26
3.1.3 VerilogEvalV2 . . . 28
3.1.4 CVDP . . . 32
3.2 LLMs for RTL Code Generation . . . 37
3.2.1 Domain-specific LLMs . . . 37
3.3 Agentic Approaches . . . 39

3.3.1 ReAct . . . 39
3.3.2 OpenAGI . . . 41
3.3.3 Veri Sure . . . 42
3.3.4 EvolVE . . . 45
3.3.5 ACE . . . 48
3.3.6 ACE-RTL . . . 49

4 Methodology 55
4.1 Benchmark Customization . . . 55
4.1.1 Dataset Selection . . . 56
4.1.2 LLM Integration . . . 56
4.1.3 Agent Integration . . . 57
4.1.4 Execution Environment . . . 59
4.2 Experimental Protocol . . . 60
4.2.1 Evaluation Configurations . . . 60
4.2.2 Evaluation . . . 61
4.3 Evaluation Metrics . . . 61
4.3.1 Pass@k . . . 62
4.3.2 Agentic Pass Rate . . . 62
4.4 Failure Analysis . . . 63

5 Results 66
5.1 Experimental Setup . . . 66
5.2 Agentic Systems Comparison . . . 67
5.2.1 Task Difficulty and Categories . . . 67
5.2.2 Commercial and Non-Commercial Dataset Differences . . . 69
5.2.3 Testbench Awareness . . . 70
5.2.4 Impact of Model Scaling . . . 73
5.2.5 Time and Cost . . . 74
5.3 Failure Analysis . . . 75
5.3.1 Overall Failure Patterns . . . 75
5.3.2 Failure Mechanisms . . . 76
5.3.3 Benchmark Issues . . . 77
5.4 Discussion and Practical Implications . . . 78

6 Conclusions 81

A Functional description of a CVDP datapoint 83

Bibliography 91

List of Figures

2.1 ASIC design flow from [1] 5
2.2 Pipeline of the training process 7
2.3 pipeline of an LLM iteration 10
2.4 Generic pipeline of an agent iteration 13
2.5 Figure from [28] describing the high level structure of Claude Code 18
2.6 Figure from [31] describing the agent multi-turn interaction framework 20

3.1 RTLLM workflow for completely automated design RTL generation and evaluation. [13] 25
3.2 VerilogEval sandbox environment, Fig. 1 in [12] 27
3.3 pipeline of a task execution in VerilogEvalV2 30
3.4 Types of failures supported by automatic failure classification. [36] 31
3.5 CVDP evaluation flow from [14] 35
3.6 Picture from [42] paper: agents representation 43
3.7 Picture from Evolve [43]: workflow representation 46
3.8 ACE-RTL iteration representation from [45] 51
3.9 Parallel scaling representation from [45] 53

Acronyms

ACE Agentic Context Evolution.

AI Artificial Intelligence.

API Application Programming Interface.

APR Agentic Pass Rate.

ASIC Application-Specific Integrated Circuit.

AST Abstract Syntax Trees.

CVDP Comprehensive Verilog Design Problems.

DUT Design Under Test.

EDA Electronic Design Automation.

HDL Hardware Description Languages.

ICL In-Context Learning.

IGR Idea-Guided-Refinement.

LLM Large Language Model.

MCTS Monte-Carlo Tree Search.

NLP Natural Language Processing.

PPA Power, Performance, and Area.

RNN Recurrent Neural Network.

RTL Register Transfer Level.

STG Structured Testbench Generation.

TA Testbench Aware.

TU Testbench Unaware.

Chapter 1

Introduction

The increasing complexity of modern digital circuits has made their design and testing one of the most difficult and costly phases of the entire development process, which brought forth a need for automation [1]. Electronic Design Automation (EDA) tools allowed for the successful automatization of numerous development phases for digital design, such as the logical synthesis phase and the timing analysis. Contrarily, the Register Transfer Level (RTL) design and functional verification still involve substantial human effort. Engineers traditionally translate system specifications into RTL descriptions, develop verification environments, and identify and correct functional errors. Despite recent advances, complete automation remains challenging [2].

In the last few years, there has also been a revolution in the Artificial Intelligence (AI) industry, driven by the Large Language Models (LLM) advancement in natural language comprehension and code generation [3, 4]. Their success in software development brought the scientific community to wonder if it could be extended to the hardware setting.

Various works [5, 6, 7] have shown how LLMs can be used to generate RTL code starting from textual specifications, modify it, and debug. While it is proven that LLMs are able to work in the hardware setting, it appears to still be harder then the software one for them. This difference in perceived difficulty is caused by the lack of training data for hardware specialization [2] and by the fact that while software tends to be sequential in execution, hardware is by default parallel [8].

The most recent evolution brought to the birth of *AI Coding Agents* [9, 10]. Differently from traditional code generation systems, these agents are not limited to a single answer, but can operate autonomously inside of the development environment, in fact they can analyze existing repositories, modify multiple files and execute simulations and compilations and interpret the outputs. This paradigm is closer to the real pipeline executed by hardware engineers during a RTL development project.

Even though great progress has been made, as of now the agents capability are still limited when confronted with real-word scenarios [11]. A lot of existing benchmarks only evaluate single-module RTL generation [12, 13]. This ignores fundamental parts of industrial hardware developments, such as module reuse, functional debugging, EDA tools interaction. To close this testing gap, new benchmarks have been recently proposed. The Comprehensive Verilog Design Problems (CVDP) [14] benchmark was specifically created to evaluate both models and agents on an extensive set of different activities, typical of hardware design and testing.

This thesis' work goal is to evaluate the performances of modern LLM and coding agents on RTL design. In particular, this work is focused on integrating and evaluating state-of-the art agents and LLMs on the CVDP benchmark 'code generation' category; and on researching and understand the motivation of AI agents failures. The final goal is to contribute to the understanding of agents potentialities and limitations in the hardware design context, highlighting the main criticalities and possible future avenues. This thesis is organized as follows:

1. Section 2 exposes the fundamentals concepts of digital hardware design and test, LLMs and agents base concepts, and a simple explanation of the agents integrated in this thesis.

2. Section 3 explains the state-of-art of benchmarks, LLMs and agents for hardware design.

3. Section 4 illustrates the benchmark customization and how the failure analysis was conducted.

4. In Section 5, this thesis results are presented and explained.

5. Section 6 has this works conclusions and proposes possible future developments.

Chapter 2

Background

The application of Artificial Intelligence to hardware design lies at the intersection of two rapidly evolving fields: digital circuit design and Large Language Models; understanding the opportunities and challenges introduced by this combination requires concepts from both the domains. The design and verification of modern digital systems are complex processes that require the translation of high-level functional requirements into hardware implementations whose behavior must be carefully validated. At the core of this process, Register-Transfer Level (RTL) descriptions provide a representation of the structure and behavior of digital circuits, while verification methodologies are used to assess their functional correctness. The increasing complexity of hardware systems has motivated the exploration of automated approaches to support these activities. In particular, recent advances in Large Language Models (LLMs) have enabled their application to code generation and, subsequently, to Hardware Description Languages (HDL). Beyond single-inference code generation, agentic systems provide a different paradigm in which LLMs can iteratively interact with a working environment, inspect existing code, modify files, execute external tools, and use their outputs as feedback for subsequent decisions. These capabilities are particularly relevant to hardware design and verification, where development naturally involves repeated cycles of implementation, simulation, error analysis, and refinement.

2.1 Hardware Design and Verification Fundamentals

The design of modern electronic systems can reference both analog and digital circuits. The main difference between those two circuits is the nature of their signal. The analog circuits elaborate continuous signals, which are the voltage and current values, while the digital circuit has binary signals, which allow the implementation

of logical operations, memory units, and complex elaboration systems [15]. This work is set into the Digital Circuits Design field.

Hardware design follows a hierarchical flow, reported in Figure 2.1, that transforms a functional specification into physical circuit [1]. This flow includes various phases (not all explicit in Figure 2.1), but in this work we will focus on the phases of Functional Specification, RTL Design, Functional Verification and Logical Synthesis.

The design flow starts with a "Functional Description" (example in the Appendix A), which is a natural language description of the hardware goal, signals, ports, module interface and submodules. The RTL project is developed starting from this description and represents the first formal implementation of the circuit. Afterwards, there is functional verification of the project, to verify that the behavior defined reflects the initial specifications and requirements. Only after a successful verification the following phases can begin, starting with the logic synthesis, which transforms the RTL into a network of implementable logical ports [16].

2.1.1 RTL Design

The Register Transfer Level (RTL) represents one of the main level of abstraction used in digital circuits, where the system's behavior is described using register, combinatorial logic and memory transfers, synchronized to the clock signal. Engineers use the RTL descriptions to translate functional requirements into hardware description that can later be verified and synthesized [8].

The RTL descriptions are usually developed using Hardware Description Languages (HDL), mainly Verilog and SystemVerilog. The HDL's programming language describes the hardware structures and concurrent behavior, rather then the sequential software execution typical of traditional programming languages. The concurrent semantic of these languages allows to represent the parallel behavior of hardware, modeling registers and signals [17]. Once the RTL description is complete, the project can be simulated and verified.

2.1.2 Testbenches and Simulation

To verify that an RTL project reflects the behavior specified by the functional description there is the phase of Functional Verification, which is arguably one of the most important phases of the design process. The verification is usually done through a simulation. During the simulation the hardware module (which is usually referred to as Design Under Test (DUT)) is executed inside of a testbench.

The testbench generates the input stimuli to be applied to the DUT, observes the output produced and compares the behavior expect to the one observed, thus

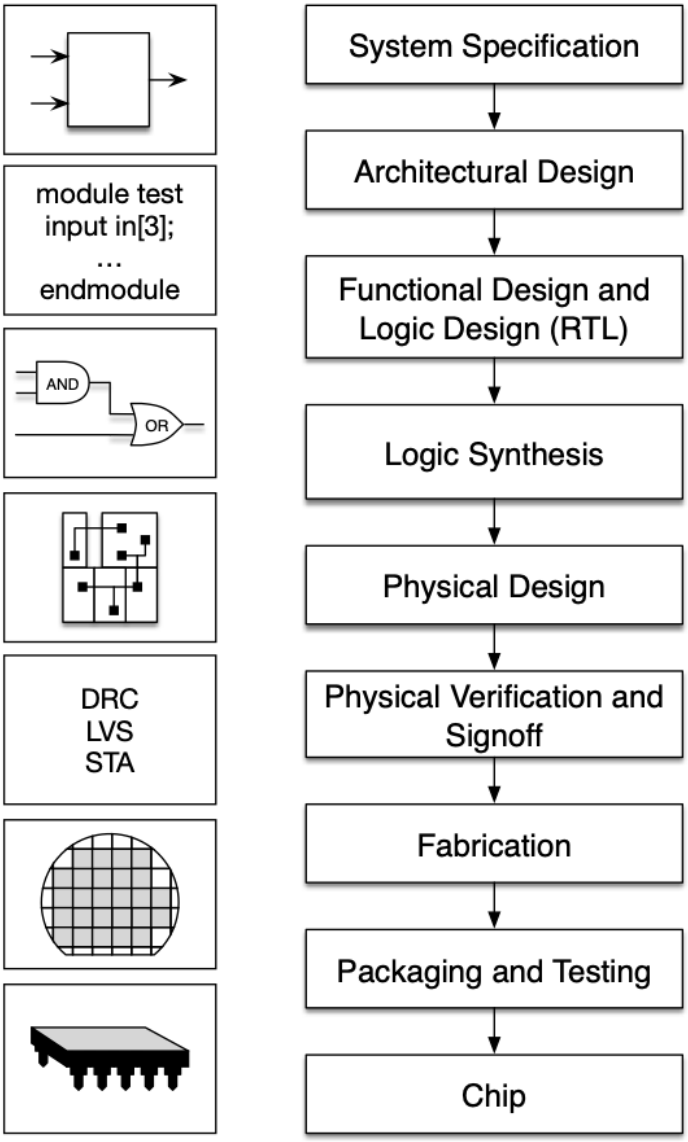


Figure 2.1: ASIC design flow from [1]

highlighting the functional errors. This allows to spot design errors before the synthesis and the fabrication, thus lowering the cost and time of production.

2.1.3 Limitations

While several later stages of the hardware design flow are extensively supported by EDA tools, the phases of RTL design and Functional Verification still require an high amount of human intervention: RTL coding and testbench writing remain labor-intensive prevalently manual activities [18]. The dependency to human intervention makes this activities expensive both in time and costs. In the last years this motivated a raising interest into the use of LLM and agentic workflows, to automate these activities [2].

2.2 Large Language Models

Large Language Models (LLMs) are a revolutionary AI technology capable of comprehending and generating natural language. LLMs were originally developed for Natural Language Processing (NLP) applications, but soon it was observed that they also had remarkable capabilities in programming-related tasks [3]. LLMs are trained on huge amounts of data and are currently mainly based on the Transformer architecture [19].

2.2.1 Training Process

The capabilities of modern LLMs are not acquired through a single training phase, but through a sequence of training and alignment stages, each serving a different purpose. The initial stages are responsible for learning linguistic knowledge and reasoning capabilities from large-scale text corpora, while the following stages progressively adapt the model to follow user instructions, align its behavior with human preferences, and, when required, specialize it for specific application domains [20]. Figure 2.2 summarizes the typical training pipeline adopted by modern language models.

Pretraining

LLMs get their capabilities through an initial step called pretraining, where they are exposed to huge amounts of text, from books, scientific articles, web pages, and also code repositories; this data are commonly referred to as the corpus. Pretraining is inherently general-purpose: the learning is not specific to a single domain. During this phase, autoregressive models learn to predict the next token and thus builds complex statistical representations, which will provide the model with factual and semantic knowledge [3].

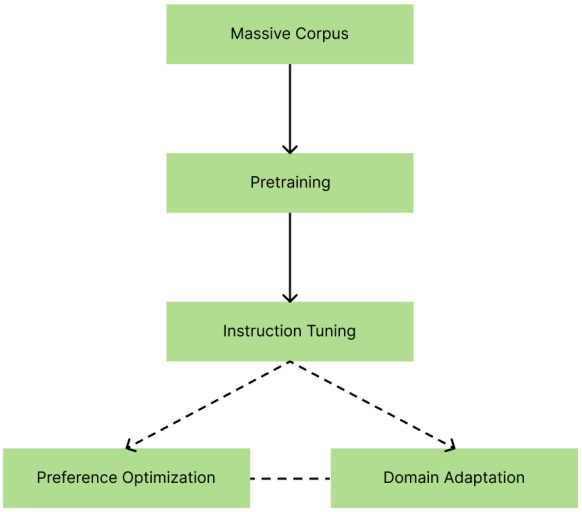


Figure 2.2: Pipeline of the training process

Instruction Tuning

After the pretraining phase, the model has a deep knowledge on the natural language, on code and generally on the information contained in the corpus. What it does not have is the know-how to interact with the user and to follow instructions. For this reason there usually is a second training phase: the instruction tuning. Instruction tuning does not necessarily introduce new information or knowledge: it changes the model behavior, improving its ability to answer questions, generate code, and in general to complete the tasks given by the user [21].

In specialized applications this phase can be adapted using instructions based on a specific domain. In this way the model will be familiar with the requests that it will encounter during real-use.

Domain Adaptation

Although pretraining and instruction tuning enable LLMs to perform a wide variety of tasks, their performance may still be limited when applied to highly specialized domains. To improve their effectiveness, models can undergo **domain adaptation**, whose goal is to better align their knowledge and behavior with the target application.

Domain adaptation can be achieved through different strategies. One approach

consists of continuing the pretraining phase using large collections of domain-specific data, allowing the model to acquire the terminology, conventions, and patterns characteristic of the target domain [22]. Another approach consists of performing additional instruction tuning using domain-specific instruction and response pairs, enabling the model to better understand and follow the types of requests it will encounter during deployment.

In the context of hardware design, several works have explored these techniques to adapt LLMs for RTL code generation. For example, VerilogEval [12] employs a Self-Instruct pipeline to automatically generate Verilog (instruction, response) pairs, which are subsequently used for domain-specific instruction tuning.

2.2.2 Inference

Before being able to analyze a text, an LLM needs to convert it into a numerical representation. This process is known as **tokenization** and consists of dividing the text in units called **tokens**, which will be the real unit the model will process. A token is not necessarily a word, it can also represent a punctuation symbol, a word fragment, or a special character. The use of tokens allows the model to manage uniformly natural languages, code and symbols that actually belong to very different domains. Once an input is tokenized, the number of tokens that can be processed at the same time is limited to the **context window**. The context window is defined as the maximum amount of information that the model can manage during a single inference. Inside the context window there are all information that is available to the model, including system instructions, user prompts, the conversation history, etc. Consequently, the dimension of the context window has an impact on the ability of the model to tackle difficult problems, since a larger context window would allow the model to use more information, while a model with a smaller context window will at some point need to delete some of the information available [20].

Initially, LLMs would transform a token sequence in another, only using the information contained in their context window and in their pretraining. While this paradigm was effective for a wide range of tasks, it limited the model's ability to interact with external environments and access information or capabilities beyond those encoded in its parameters. To overcome this limitation, tool-use mechanisms were introduced to allow models to perform actions such as invoking a function or a tool, get the result and use it in their reasoning [23]. The available tools can provide a wide range of capabilities, including web access, mathematical computation, and interaction with file systems.

2.2.3 Transformer Architecture

The transformer architecture represents LLMs key breakthrough [19]. Before the introduction of the Transformer, neural language modeling commonly relied on recurrent architectures, including Recurrent Neural Networks (RNN) [24] and their evolutions like Long Short-Term Memory (LSTM) [25]. Recurrent neural networks elaborate text in a strictly sequential manner, handling one token at a time and keeping an **hidden state** which is updated each time a token is processed. This mechanism is known as **recurrency**, and it enables the model to retain information from previously processed tokens when interpreting the current input. While this approach enables to model sequential data (which text is), it also makes it difficult to capture the relationship between words that are far apart in the sentence.

Unlike RNNs, the transformer architecture doesn't apply recurrency, but it introduces self-attention [19]. With self-attention the model does not need to process the tokens one after the other, it receives the whole sequence of tokens as input and determines, for each one, which other tokens are most relevant to understand the meaning. In this way the LLM is able to learn during the training which relations are important to understand the meaning of a sentence.

With self-attention the LLM is now able to capture long-range dependencies without relying on RNN's sequential recurrence. Since the representation of each token does not depend on the completion of the previous recurrent step, the tokens of a sequence can be processed in parallel during training, significantly improving computational efficiency. Transformers are now the underlying architecture for nearly all LLMs, including GPT, Claude and Gemini.

2.2.4 Applications to Code Generation

LLMs were originally applied to obtain automatic translations, question answering, text summarization and other NLP tasks, but it was soon observed that they could also be used for programming. This is because during pretraining, modern LLMs are exposed not only to natural language but also to large quantities of publicly available source code. As a result, they learn the syntax, programming patterns, software libraries, and coding conventions of multiple programming languages, enabling them to perform a wide range of software engineering tasks.

Early applications mainly focused on code completion and function generation. More recently, advances in reasoning capabilities and tool use have enabled LLMs to support increasingly complex software engineering activities. Models like GPT, Claude, Gemini, etc, are able to analyze repositories, write code, debug and in general assist developers during the software development cycle.

Given the structural similarities between software source code and HDL such as Verilog and SystemVerilog, recent research has investigated whether the same

models can assist digital hardware design. However, RTL generation introduces additional challenges, including concurrency, clock-driven semantics, timing constraints, and functional correctness requirements that are not commonly encountered in traditional software development.

While tool-use extends the capabilities of an LLM beyond text generation, each iteration (Figure 2.3) still consists of a single inference. More complex tasks, such as software or hardware development, often require multiple reasoning steps, repeated interactions with external tools, and iterative refinement. These requirements have led to the emergence of agentic systems, discussed in the following section.

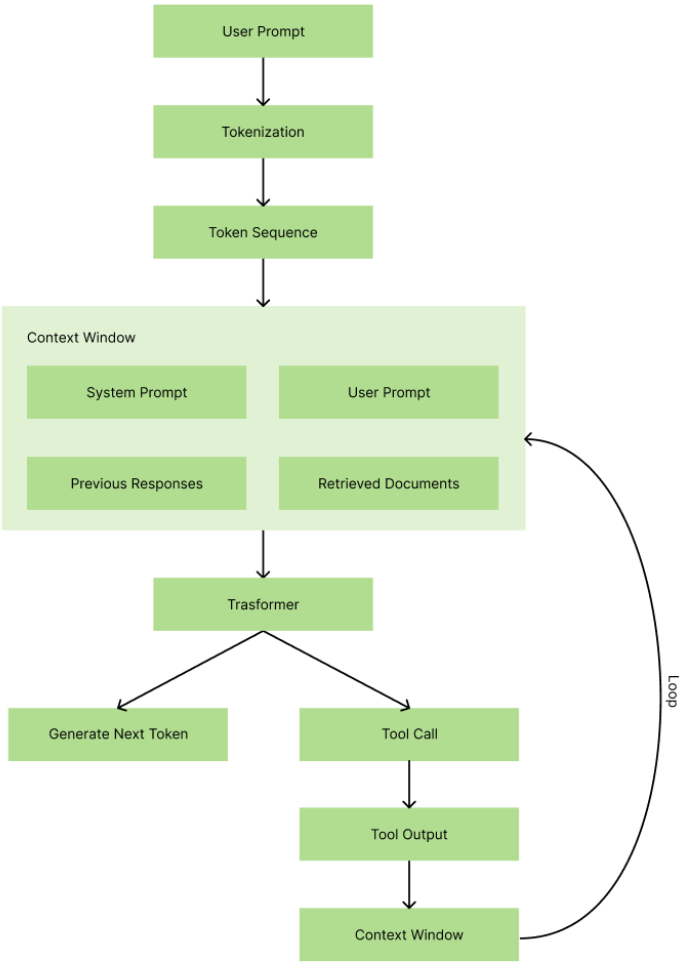


Figure 2.3: pipeline of an LLM iteration

2.3 Agentic Systems

LLMs are capable of generating high-quality natural language and source code. However, a standard LLM interaction consists of a single inference step: the model receives an input prompt, generates an output, and the execution terminates. Many real-world tasks, however, cannot be solved reliably in a single inference. Software and hardware development, for example, require iterative reasoning involving planning, code generation, compilation, testing, debugging, tool invocation, and repeated refinement.

To address these limitations, recent research has shifted towards agentic systems, where an LLM is embedded inside an execution framework capable of iterative reasoning and interaction with external tools [9].

Rather than considering the LLM as the complete application, agentic systems separate reasoning from execution. The language model acts as the reasoning engine, the decisional component, while an external runtime manages context construction, tool execution, memory, permissions, and interaction with the environment [10].

A common paradigm for agentic frameworks is an Observation-Reasoning-Action loop, inspired by approaches such as ReAct [9], where the model alternates between reasoning steps and interactions with external tools. At each iteration the model analyzes the context, decides if a tool is needed, observes the result and updates its reasoning, until the final iteration, as can be seen at figure 2.4

The innovation of modern agents is the implementation of a variant of the **agentic loop**: an iterative cycle in which the runtime and the language model cooperate throughout multiple iterations until the task is completed. The execution typically begins with the construction of the context window. Besides the user's request, the runtime includes the conversation history, available tools, system instructions, and any additional contextual information required for the current step. The runtime then invokes the LLM, which can either generate a final textual response or request the execution of one or more tools. When a tool invocation is requested, the runtime executes the corresponding operation, collects its output, updates the context window, and performs another inference. This process repeats until the model decides that the task has been completed or a predefined stopping condition is reached [10, 9]. This paradigm is particularly useful for software and hardware development, where the design process is naturally iterative and comprehends multiple phases of code modification, with compilation, test execution and error correction.

2.3.1 Common Architecture

Although modern agentic systems differ in their implementation and capabilities, they generally share a common architectural structure (Figure 2.4). Rather than

relying solely on the language model, these systems combine the LLM with an external runtime responsible for orchestrating the execution process.

For each iteration, the runtime constructs the context window by combining the user's request, the conversation history, system instructions, available tools, and any additional information required for the current task. The resulting prompt is then submitted to the language model, which decides either to generate a textual response or to invoke one or more external tools.

The tool system extends the capabilities of the language model by allowing interactions with the external environment [23]. Depending on the application domain, the available tools may include file system operations, shell command execution, web search, repository navigation, code editing, compilation, simulation, or access to external services through APIs. Instead of directly performing these operations, the LLM generates structured tool requests that are interpreted and executed by the runtime.

Agentic runtimes may also implement permission mechanisms to control potentially unsafe or environment-modifying tool calls: before executing a requested action, the runtime verifies whether the operation is authorized according to predefined policies or explicit user approval. This mechanism provides an additional safety layer while allowing the agent to perform complex tasks autonomously.

Long-running executions introduce another important challenge: the limited size of the context window: as each interaction, tool output, and intermediate reasoning step consumes additional tokens, the available context can quickly become exhausted. Modern frameworks therefore employ context management strategies such as summarization, pruning, retrieval, or context compaction to preserve the most relevant information while remaining within the model's context limit.

Some recent agentic systems further improve scalability through the use of specialized subagents. Instead of solving every aspect of a task within a single execution loop, the main agent delegates specific subtasks to secondary agents operating in isolated contexts. Once completed, only the final results are returned to the main execution, reducing context growth while enabling parallel execution of independent activities.

2.3.2 OpenCode

OpenCode [26] in an open-source general-purpose framework for LLM-based agentic workflows. The difference between openCode and a traditional chatbot is that openCode does not receive the prompt and directly produce a response, but it integrates the language model inside of a runtime that orchestrates the entire execution. The LLM is still responsible of the reasoning, while the runtime is responsible of all operative aspects, such as the interaction with the file system or the building of the context.

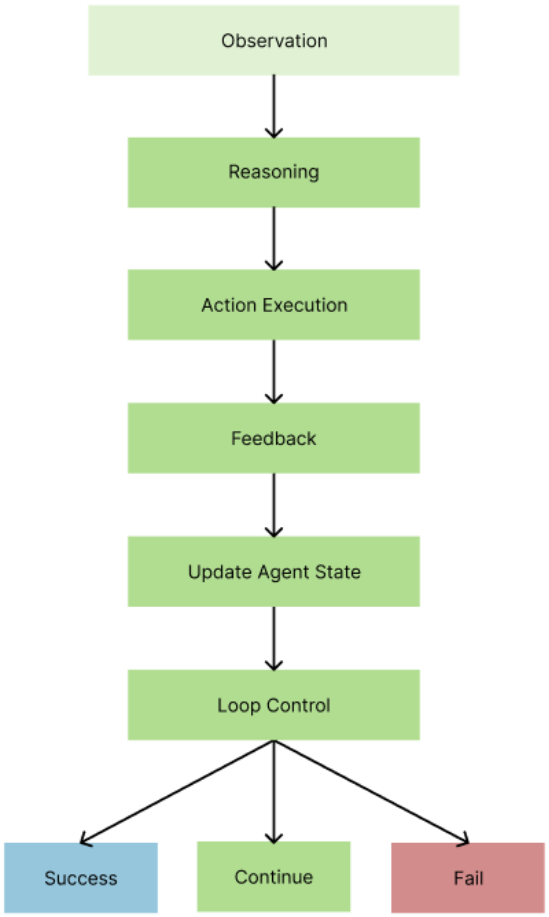


Figure 2.4: Generic pipeline of an agent iteration

The agent architecture is completely domain agnostic: it is not specific to software or hardware development. OpenCode's combines an LLM with an execution runtime capable of interacting with the local environment through a rich set of tools, which makes it particularly suited to development workflows.

The openCode agentic loop is a state machine driven by the LLM's events, with recursive child sessions for the subagents, an automatic management of the context, and a granular permission system for each action. This means that the agent execution is organized around a loop implementing the ReAct paradigm (introduced in Section 2.3). At each iteration the model receives the context, reasons, decides if it wants to call a tool and uses the results as a new context for the next iteration. This goes on up until the the model decides that the task is

completed.

From an implementation point of view, this behavior is realized with a double while cycle. The external loop orchestrates the session: it keeps the state of the session, verifies the termination conditions, manages pending subtasks, and manages the context and its compaction. The internal loop is strictly dedicated to the interaction with the LLM. In the internal loop the context is provided to the LLM which returns an answer in the form of a stream of events, which can represent text, reasoning phases, tool requests, tool results or termination signals. At the end of the elaboration, this internal loop returns to the external one a state (*continue*, *stop*, or *compact*), which determines the external loop next action.

Agents and Tools

OpenCode has different types of agents, each one with different responsibilities and permissions. There is a main agent, which is used for most activities, and then there are other agents, subagents and hidden agents as seen in Table 2.1. Some agents are dedicated to the planning stage, others to the repository exploration, other to summarizing, etc.

A very interesting characteristic of this architecture are the *subagents*. A subagent such as '**general**' is not merely an LLM call, but it constitutes a completely new instance of the agentic system. When a model decides to delegate a subtask, a child session is created and given a context, an agentic loop and set of tools. The subagent will autonomously execute its reasoning cycle and return the final result to the parent session.

If not launched with a specific tool, the subagent execution is sequential. Multiple subagent instances can be launched in parallel, each with a different subtask, using the '**batch**' tool. Agents can also be user-defined.

Name	Mode	Role
build	primary (default)	Default agent: executes tools; planning and coding
plan	primary	Only planning; it cannot modify the files
general	subagent	Multi-step tasks parallelization
explore	subagent	Fast exploration of the codebase
compaction	primary (hidden)	Compresses the conversational history, shortens the context
title	primary (hidden)	Generates titles for the sessions
summary	primary (hidden)	Produces session's summaries

Table 2.1: OpenCode's agents description

The interaction with the environment happens through multiple tools, as reported at Table 2.2, which represent the model interactive interface with the external system. The system also supports the integration of personal tools and services compatible to the Model Context Protocol.

Tool	Description
bash	Executes shell commands
read	Reads files and directories
glob	Pattern matching on files
grep	Researches file content
edit	Modifies existing files (string replacement)
write	Create/overwrites files
task	Launches a subagent in a child session
webfetch	HTTP requests
websearch	Web research
todowrite	Manages a todo list
question	Asks clarifying questions to the user
batch	Parallel tool execution
multiedit	Multiple edits in a single call
lsp	Language Server Protocol integration (experimental)

Table 2.2: OpenCode's tools description

Since complex tasks can require hundreds od consecutive operations, openCode implements various mechanisms to limit the growth of the context window.

The first one is the **automatic compaction**, which is activated when the number of token rises over the threshold. In this case, a dedicated agent will produce a summary of the conversation, substituting a consistent part of the conversation with a synthetic description of the main events.

A second mechanism is the **complete compaction**, which is launched when an overflow of the context window is alerted by the LLM provider. Before completely compacting the context, the system tries to reduce it by deleting particularly big elements, like multimedia files.

At last, a **pruning** technique is also implemented, which replaces the output of old tool calls with placeholders instead of removing them completely, thus allowing the structure of the conversation to be preserved.

2.3.3 Claude Code

Claude Code [27] is an agentic system developed by Anthropic specifically for terminal software development: it is a domain-specific agent. This agentic model (unlike traditional coding assistant) does not just generate code: it interacts with the file system, launches commands, modifies files and delegates to subagents. The step from simple command suggestion to autonomous execution introduces specific architectural problems, mainly related to security, context and session persistence.

The description of Claude Code architecture derives from the analysis of the publicly available TypeScript source code that can be found in [28]. The authors explain that the agentic kernel is simple: a single cycle invokes the model, executes the tools and and repeats the process.

The main part of the implementation is constituted by the operational harness, which is the deterministic infrastructure that encapsulates the model and manages permissions, tools, context etc.

Philosophy

The paper [28] finds in the architecture five main values.

The first value is **human decision authority**, by which the user must maintain complete authority over the system actions and must be able to observe and manage the agent behavior. The second value is the **privacy and security** of the user, even when the user itself is not aware of the danger. The **reliable execution** is the third value, and it comprehends both the correct interpretation of the request and the coherence during long tasks. The fourth value is **capability amplification**, the augmentation of the user abilities through the agent and the fifth value **contextual adaptability** by which the agent adapts itself to the user preferences.

There are thirteen design principles that make sure that this five values are respected by asking the agents a question he must answer.

General Architecture

Claude Code is said to have 7 principal components: user, interface, agent loop, permission system, tool, state and execution environment.

The user sends a task through an interface, the request enters into the agentic cycle, the context is constructed and the model (and eventually the tools) called. The tool requests go through the authorization system and if approved, are inoltrated to instruments that actually execute them. The results are then reintroduced into the cycle and inserted into the chronology and the state (as per paper [28], Figure 2.5).

At the hearth of the system there seems to be a 'queryLoop', which is an asynchronous generator that realizes the agentic cycle. Tasks can be delegated by the model, specifying the type of the subagent and subtask assigned to it. A subagent is not an isolated model call: it executes a new agentic loop with a separated context window and different set of tools. The main session will not receive the entire subagent's chronology, but only the end result and the metadata, while the transcription of the subagent conversation will be saved separately to reduce the consume of the context window.

The model answers can contain text or structured blocks to request tool use. When there are tool requests, the runtime interrogates the authorization system and executes the authorized ones. These results will become part of the next conversation. The tool execution can be partially concurrent: the operations that are considered safe to parallelize (essentially the reads) can be started while the model is still streaming the answer.

Claude Code appears to manage the context window as the primary limiting resource. Before each call the context is constructed from different sources, like rules specific to certain subdirectories, user global memories, project memories, and several others.

The 'CLAUDE.md' file is a transparent set of instruction that the user can modify and that are presented to the model as context, consequentially the model observance to those instruction is not certain. The automatic memory is also based on readable and writable files, since the model prioritizes transparency and user control.

2.3.4 Claw Code

Claw Code [29] is an open-source command-line coding agent, designed to assist developers in software engineering tasks through natural language interaction, and inspired from the publicly available TypeScript source code of Claude Code.

Rather than acting solely as a code completion assistant, Claw Code operates as an autonomous agent capable of exploring a project, modifying source files,

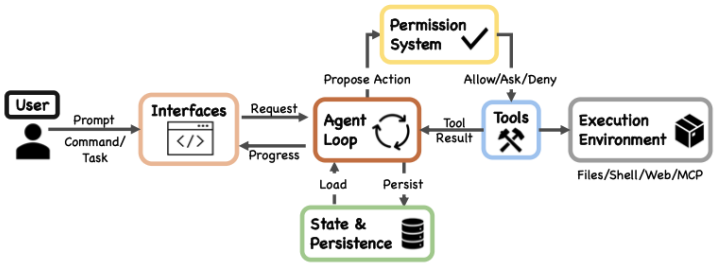


Figure 2.5: Figure from [28] describing the high level structure of Claude Code

executing commands, and iteratively refining its solutions based on execution feedback.

Like in Claude Code, in Claw Code the user specifies a high-level objective in natural language, while the agent autonomously determines the sequence of actions required to accomplish the requested task. Depending on the problem, these actions may include inspecting the repository structure, searching for relevant files, editing source code, executing shell commands, compiling the project, running simulations, or analyzing test results before deciding on subsequent actions.

Unlike traditional code completion systems, Claw Code continuously interacts with its execution environment. Information gathered during execution is incorporated into the conversation context, enabling the language model to update its reasoning and progressively improve the generated solution. This iterative reasoning process allows the agent to handle complex software engineering tasks that require multiple interactions with external tools rather than a single code generation step

A key characteristic of Claw Code is its extensive tool integration. Besides file system operations, the agent can invoke shell commands and interact with external development tools available in the execution environment, including version control systems, compilers, simulators, testing frameworks, and other command-line utilities. This capability enables the construction of complete software engineering workflows in which code generation, execution, verification, and debugging are performed autonomously by the agent.

Claw Code is an actively developed open-source project, with frequent updates and changes to its implementation. Consequently, reproducibility requires fixing a specific software revision during experimentation. The experimental results reported in this thesis correspond exclusively to the version evaluated [30] and should not be considered representative of future releases, whose behavior may differ due to the rapid evolution of the project.

2.3.5 Chipmind Agents

Chipmind Agents is a proprietary coding agent developed specifically for hardware design and verification tasks. While its implementation details are not publicly available, its overall architecture follows the CodeAct [31] paradigm, where the language model iteratively reasons, invokes tools, observes the execution results, and decides the next action.

CodeAct is an agentic framework that unifies reasoning and acting into a single iterative execution loop (Figure 2.6). Instead of separating planning from execution, the language model directly generates executable actions, such as shell commands, code modifications, tool invocations, or file operations, which are immediately executed by the surrounding runtime. The outcome of each action is then observed and appended to the conversation context, allowing the model to iteratively reason over the updated environment before deciding the next action. This observation-reasoning-action cycle continues until the task is completed or a termination condition is reached. Unlike traditional workflows that rely on predefined pipelines, CodeAct treats actions as first-class outputs of the language model, enabling flexible interaction with external tools and the execution environment.

The execution is organized around a single main agentic loop, which orchestrates the interaction between the language model and the execution environment. During each iteration, the agent can invoke a wide range of tools, execute shell commands, inspect project files, and delegate specialized subtasks to dedicated subagents. These subagents are primarily organized in a sequential fashion, each responsible for a specific activity within the overall workflow, while the main agent remains responsible for global coordination and decision making.

To better support RTL development, Chipmind Agents integrates domain-specific capabilities, including specialized RTL-oriented tools and skills, together with hardware-aware guardrails. The execution flow and context management are also regulated, helping the agent remain focused as the conversational context grows during extended interactions. This is particularly relevant for hardware design tasks, where multiple iterations of code generation, debugging, simulation, and refinement can quickly lead to very large execution histories.

The system is primarily designed for engineering tasks on large codebases, where maintaining coherent execution flow and managing extensive context is essential for effective operation.

2.4 Motivation

Digital hardware design is a complex and resource-intensive process that requires several iterative phases, including specification analysis, RTL implementation, functional verification, debugging, and validation. Although EDA tools have

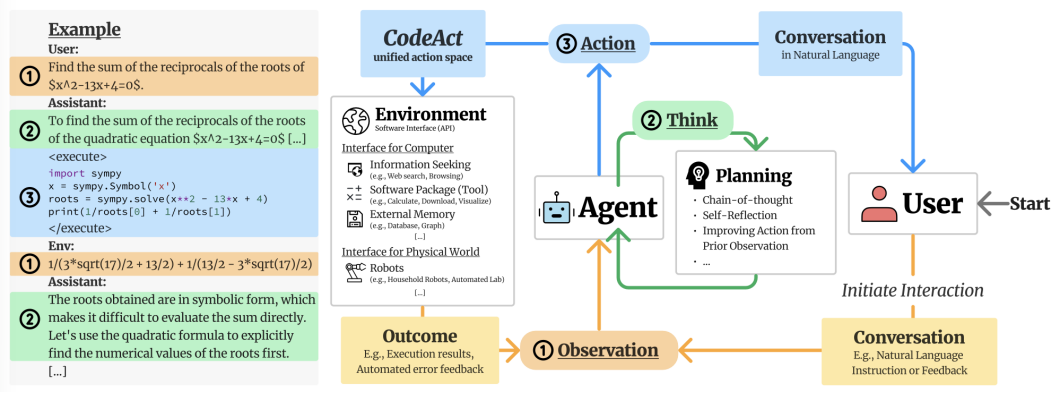


Figure 2.6: Figure from [31] describing the agent multi-turn interaction framework

significantly improved the designer's productivity over the last decades, many stages of the design flow still rely heavily on manual engineering effort. In particular, debugging functional failures and implementing RTL modifications often require repeated interactions between code development and simulation, making hardware development both time-consuming and expensive.

Recent advances in LLMs have demonstrated remarkable capabilities in software engineering tasks. Modern coding assistants are now capable of generating source code, explaining existing implementations, fixing bugs, and interacting with development environments. Their success has naturally motivated researchers to investigate whether similar techniques can be applied to digital hardware design.

Several recent works have explored the use of LLMs for Verilog generation, while new hardware-specific benchmarks such as VerilogEval and CVDP have been introduced to systematically evaluate model performance. However, most existing studies primarily focus on **non-agentic** code generation, where a language model receives a prompt and produces a single RTL implementation. While this paradigm provides a useful baseline for measuring synthesis capabilities, it does not accurately reflect how modern coding assistants operate in practice.

Contemporary coding agents can autonomously inspect repositories, execute external tools, analyze simulation outputs, modify source files, and iteratively refine their solutions based on execution feedback. Such capabilities are particularly relevant for hardware development, where functional correctness is typically established through repeated simulation and verification rather than through a single code generation step.

Despite the rapid emergence of agentic coding frameworks, relatively little work has investigated their effectiveness for RTL design and hardware verification tasks. In particular, there is currently limited experimental evidence comparing different agentic coding assistants under a common, **difficult** evaluation framework and

analyzing their strengths, limitations, and failure modes in hardware-oriented workflows.

This thesis addresses this gap by evaluating state-of-the-art coding agents on the Comprehensive Verilog Design Problems (CVDP) benchmark. By comparing multiple agentic frameworks under identical experimental conditions and investigating their failures, this work aims to provide a better understanding of the current capabilities and limitations of LLM-based autonomous agents for digital hardware design.

Chapter 3

Related Works

The rapid progress of LLMs has stimulated a growing body of research investigating their application to digital hardware design. Initially, most efforts focused on evaluating the capability of general-purpose language models to generate synthesizable RTL code from natural language specifications. As the field matured, researchers progressively explored more sophisticated approaches, including domain-specific model adaptation, specialized evaluation benchmarks, and agentic workflows capable of interacting with external tools and iteratively refining their solutions.

3.1 Hardware Design Benchmarks

The rapid adoption of LLMs for RTL code generation has highlighted the need for standardized evaluation methodologies, capable of objectively measuring and comparing different models and approaches. Unlike traditional software generation tasks, hardware design requires not only syntactically correct code, but also functionally correct implementations that satisfy the intended circuit specification. For this reason, evaluating generated RTL designs typically involves simulation, functional verification, and increasingly sophisticated correctness metrics.

To address these challenges, several benchmark suites have been proposed in recent years. While they all aim to assess the ability of LLMs, in the RTL domain a 'good performance' is not a unique concept: code generation, code completion, code comprehension, testing, metrics, etc, can vary between benchmarks. For this reason each benchmark can differ from another in datasets, prompts, test infrastructure and metrics.

3.1.1 RTLLM

As the application of Large Language Models to RTL code generation began attracting increasing research interest, the need for standardized evaluation methodologies became evident. Existing works [32, 33] typically considered a limited number of relatively simple designs, often proposed by the authors themselves, making fair and reproducible comparisons between different approaches difficult. Furthermore, most evaluations focused exclusively on syntax or functional correctness, without assessing the quality of the generated hardware implementations.

To address these limitations, Lu et al. proposed RTLLM [13] an open-source benchmark for the evaluation of RTL code generation from natural language specifications. Compared with previous benchmarks, RTLLM introduces a larger and more diverse collection of hardware designs, covering different functionality types, implementation complexities, and circuit scales. More importantly, it extends the evaluation methodology beyond syntax and functional correctness by introducing a third objective, referred to as the *quality goal*, which assesses the generated designs in terms of power, performance, and area (PPA). This allows RTLLM to provide a more comprehensive assessment of RTL generation systems and better reflects the practical requirements of digital hardware design.

Dataset Construction

RTLLM is composed of 30 hardware design problems covering a broad range of digital design functionalities and implementation complexities. Unlike previous benchmarks, which mainly focused on small educational examples, RTLLM includes designs ranging from simple arithmetic and logic circuits to significantly more complex systems, providing a more representative evaluation of RTL generation capabilities.

For each design, the benchmark provides three complementary artifacts.

- The first is a natural language specification (*design_description.txt*), describing the required functionality of the circuit together with the module name and the complete definition of all input and output signals. By explicitly specifying the module interface, the benchmark enables fully automated compilation and functional verification of the generated RTL.

- The second artifact consists of a Verilog testbench (*testbench.v*), containing multiple test cases used to verify the functional correctness of the generated implementation.

- Finally, the benchmark provides a human-designed reference implementation (*designer_RTL.v*). Besides serving as a validated implementation of the target functionality, this reference design is synthesized to obtain its PPA metrics,

which are then used as the baseline for evaluating the quality of the RTL generated by the language model.

The benchmark includes 11 arithmetic and 19 logic designs, covering common digital circuits such as adders, multipliers, dividers, finite state machines, FIFOs, memories, arithmetic logic units, and even a simplified RISC processor. To capture different levels of implementation difficulty, RTLLM reports both the number of HDL lines and the synthesized circuit size, measured as the number of cells in the post-synthesis netlist. This results in a benchmark with considerably greater diversity in both functionality and circuit scale than previously released RTL generation datasets.

Evaluation Methodology

Unlike previous benchmarks, which primarily focus on the functional correctness of the generated RTL, RTLLM adopts a more comprehensive evaluation methodology (Figure 3.1). The benchmark defines three *progressive goals* that assess increasingly sophisticated aspects of the generated design: the *Syntax Goal*, which verifies that the generated RTL is syntactically correct and can be synthesized; the *Functionality Goal*, which evaluates whether the synthesized design behaves according to the specification; and the *Quality Goal*, which measures the implementation quality of functionally correct designs in terms of power, performance, and area (PPA). By combining these three objectives, RTLLM provides a more complete assessment of RTL generation systems, evaluating not only whether a design is correct, but also whether it represents a competitive hardware implementation.

Syntax Goal This is the first and most fundamental evaluation criterion. Its objective is to verify that the generated RTL is syntactically correct and can be successfully processed by standard EDA tools. Rather than relying solely on parser checks, RTLLM considers a design syntactically valid only if it can be synthesized into a gate-level netlist without errors. This ensures that the generated RTL satisfies the syntax requirements of the target hardware description language and can be further processed by the subsequent stages of the digital design flow.

Functionality Goal Once syntax correctness has been established, the benchmark evaluates the functional correctness of the generated RTL through simulation. Each design is compiled together with the provided testbench, which contains a set of representative test cases covering the expected behavior of the circuit. A design is considered functionally correct only if it successfully passes all test cases. As acknowledged by the authors, the testbenches sample a representative subset of the possible input combinations rather than exhaustively exploring the entire input

space. Consequently, passing all test cases provides strong evidence of functional correctness, although it cannot formally guarantee it.

Quality Goal Unlike previous RTL generation benchmarks, RTLLM also evaluates the implementation quality of functionally correct designs. After a generated RTL successfully satisfies both the Syntax and Functionality Goals, it is synthesized and analyzed to measure its PPA characteristics. These metrics are then compared against those obtained from a human-designed reference implementation provided by the benchmark.

By introducing this third evaluation objective, RTLLM recognizes that multiple RTL implementations may exhibit identical functionality while significantly differing in hardware efficiency. The Quality Goal therefore extends the evaluation beyond functional correctness, providing a more realistic assessment of the practical usefulness of automatically generated RTL designs.

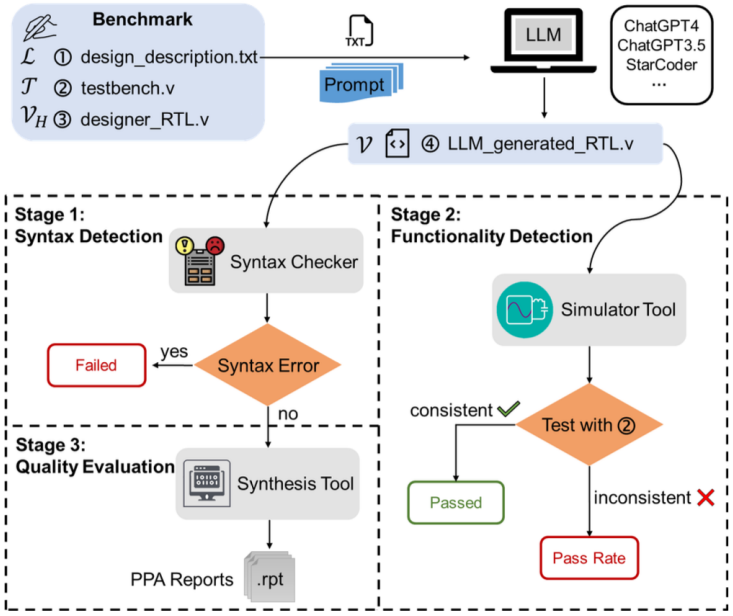


Figure 3.1: RTLLM workflow for completely automated design RTL generation and evaluation. [13]

Strengths and Limitations

RTLLM represents one of the first comprehensive open-source benchmarks specifically designed for RTL generation from natural language specifications. Compared with previous works, which typically evaluated language models on a small number

of relatively simple circuits, RTLLM introduces a considerably broader and more diverse benchmark, covering arithmetic and logic designs with different implementation requirements, circuit scales, and levels of complexity. This makes it possible to evaluate RTL generation systems on a wider range of realistic hardware design tasks.

A major contribution of RTLLM is its evaluation methodology. Rather than considering only whether a generated design is syntactically and functionally correct, the benchmark introduces three progressive evaluation goals, culminating in the assessment of the implementation quality through post-synthesis PPA metrics. This observation is particularly relevant in hardware design, where multiple RTL implementations may exhibit identical functionality while leading to significantly different area, power, and timing characteristics after synthesis. By incorporating this third evaluation objective, RTLLM provides a more realistic assessment of the practical usefulness of automatically generated RTL designs.

Despite these contributions, RTLLM still presents some limitations. The benchmark relies on simulation-based functional verification, meaning that passing all provided test cases does not formally guarantee the correctness of the generated design. Furthermore, although the Quality Goal evaluates the synthesized implementation, it considers only post-synthesis PPA metrics and does not account for later stages of the physical design flow, such as placement and routing.

Another limitation of RTLLM is that the natural language specifications explicitly provide the module interface, including the module name and all input/output signals. While this simplifies automated verification, it also reduces the level of design inference required from the language model, making the task less representative of real-world hardware design scenarios, where interface definition is often part of the design process.

3.1.2 VerilogEval

VerilogEval [12] was one of the first benchmarks specifically designed to evaluate the functional correctness of LLMs in RTL code generation. Inspired by the HumanEval [34] benchmark for Python code generation, it provides a standardized and reproducible framework for assessing the ability of language models to synthesize Verilog modules starting from natural language specifications. Rather than comparing the generated code with a reference implementation through textual similarity, VerilogEval evaluates whether the generated design behaves correctly when simulated. The overall sandbox environment, is shown in Figure 3.2.

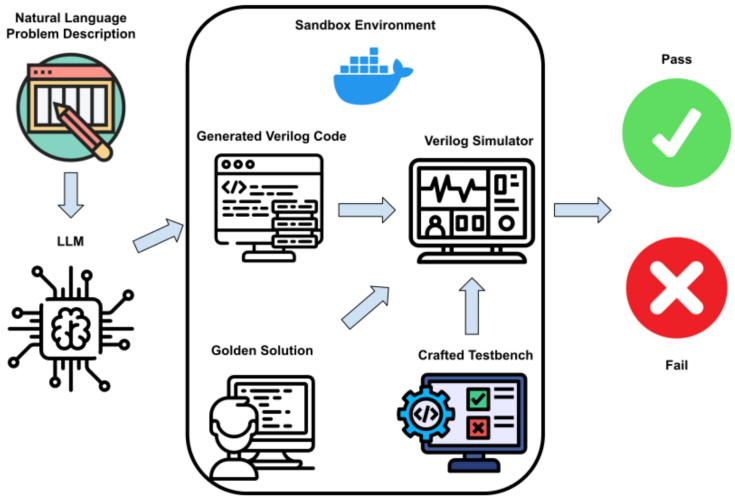


Figure 3.2: VerilogEval sandbox environment, Fig. 1 in [12]

Dataset Construction

The benchmark consists of 156 design problems derived from HDLBits [35], an educational platform widely used for learning digital design through Verilog exercises. Each problem requires the implementation of a self-contained RTL module from a natural language specification.

To make the benchmark suitable for evaluating language models, the original HDLBits exercises were transformed into (prompt, solution) pairs. The benchmark is released in two variants:

- VerilogEval-Human contains the original problem descriptions written by HDLBits authors,

- VerilogEval-Machine replaces the original problem descriptions with automatically generated descriptions produced using GPT-3.5. A generated description was considered valid only if the corresponding golden implementation successfully passed the verification process for at least one of 100 generated code samples, ensuring that the prompt contained sufficient information to solve the task.

Although both variants correspond to the same underlying RTL tasks, the machine-generated prompts are generally more verbose and explicit, making them easier for language models to interpret.

Evaluation Methodology

For each benchmark problem, the language model receives only the natural language specification and is required to generate the corresponding Verilog implementation. The generated module is first compiled together with the provided testbench and is then simulated using the verification environment.

Unlike text-based evaluation metrics, the correctness of a generated design is determined exclusively through functional verification. The outputs produced by the generated module are compared against those of the golden reference implementation, using two categories of input stimuli: manually crafted test vectors, designed to cover important corner cases, and randomly generated inputs, aimed at increasing the coverage of the verification process. A design is considered correct only if it successfully passes all the verification tests.

The primary evaluation metric adopted by VerilogEval is *pass@k*, which measures the probability that at least one of the k generated solutions is functionally correct. Since modern language models are probabilistic, multiple candidate implementations can be sampled for the same specification, making pass@k a more representative measure of practical performance than single-sample accuracy.

The authors also compare pass@k with text-similarity metrics such as the Bilingual Evaluation Understudy (BLEU) metric, which evaluates the similarity between generated and reference code. However, they show that BLEU is poorly correlated with functional correctness: implementations that are syntactically similar to the reference solution may still fail simulation, while functionally correct designs can differ significantly from the golden implementation.

VerilogEval adopts functional verification through simulation as its primary evaluation methodology.

Strenghts and Limitations

Although VerilogEval represented a significant step towards standardized RTL code generation evaluation, it focuses primarily on relatively small educational designs and provides only a binary notion of functional correctness. These limitations motivated the development of more comprehensive benchmarks, with larger datasets and more realistic verification methodologies.

3.1.3 VerilogEvalV2

The success of VerilogEval established a standardized methodology for evaluating RTL code generation using functional simulation. However, as LLM's capabilities rapidly improved, several limitations of the original benchmark became apparent. In particular, its relatively small number of tasks and the limited complexity

of the considered designs, reduced its ability to effectively discriminate between increasingly capable models.

VerilogEvalV2 [36] does not introduce a new benchmark dataset, instead it retains the same 156 HDLBits tasks in their "human" specification format. VerilogEvalV2 novelty consists in introducing several improvements to the evaluation framework, including specification-to-RTL support, in-context learning, automatic failure classification, and a standardized evaluation infrastructure.

Specification-to-RTL

When VerilogEval was first published, most code generation models mainly completed code completion tasks: the model did not use to receive a description of the problem and code the solution; instead it received also a part of the solution source code, and it only had to complete it.

Because of the rapid evolution of LLMs, this paradigm has since become not representative of real-life use. Recent LLM model are instruction-tuned and are normally used conversationally from the user, whom usually requires the implementation of the entire solution code.

To reflect this evolutions, VerilogEvalV2 introduces a second evaluation paradigm: **Specification-to-RTL**, where the model receives exclusively the textual specification of the circuit, and is asked to generate the entire Verilog module, as the benchmark RTLLM also did earlier [13]. This extension does not equate to an extension of the problem dataset: the 156 original problems are still the only problems in the dataset, they are just evaluated both in a code-completion, and specification-to-RTL mode. In this way, the benchmark does not increase the number of circuits considered, but the modes in which this problems are evaluated. This allows the evaluation of LLMs in two different scenarios: one for the original code-completion models, and one for the more recent instruction-tuned models.

In-Context Learning

Since the introduction of instruction-tuned models, the significance of prompt formulation has become significant in the model's performance. One of the most used prompt engineering techniques is the In-Context Learning (ICL) technique, where the model receives, with the problem description, also one or more examples of (request, solution) couples in the prompt. The ICL technique does not modify the model's parameters, the new information are only available via prompting and are used to guide the generation process.

To support this paradigm, VerilogEvalV2 introduced parameterizable ICL examples, that can be automatically added to the prompt before the problem description. This means that the user can choose to add examples to the prompt in a few shot mode, or to keep a zero-shot mode without adding any (Figure 3.3). This allows

for the models to be evaluated both in a zero-shot, and in a few-shot mode. The examples available are chosen to be easy, to evaluate the ICL impact, while also evaluating fairly the functional verification.

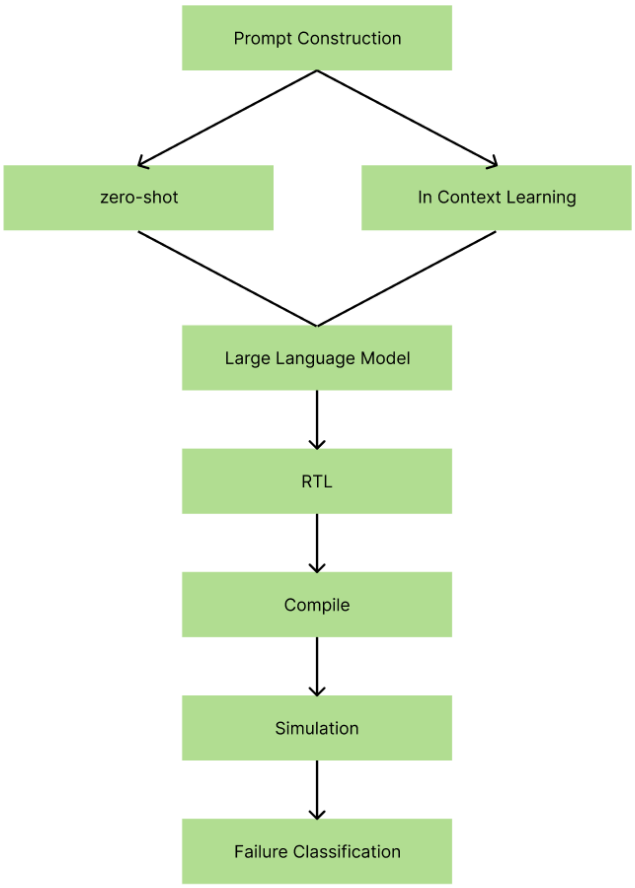


Figure 3.3: pipeline of a task execution in VerilogEvalV2

Failure Classification

The VerilogEval results were binary: a solution was classified as correct only if it passed all tests, otherwise it was wrong. This is also true for VerilogEvalV2, but a Failure Classification System was added to produce information on why the failures causes, to understand what were the models main limitations. The Failure Classification system analyzes automatically the failures and categorizes the errors by broad reasons for failures, an example can be seen in Figure 3.4.

The introduction of Failure Classification upgrades the benchmark from an evaluation tool, to a tool for both evaluation and analysis. The statistics on the error causes allow to better understand the models limitations and the behavior between different models, which in turn allows to understand the areas that need to be focused on in future works for the upgrading of RTL-generation models.

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.
Run-Time 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.

Figure 3.4: Types of failures supported by automatic failure classification. [36]

Evaluation Infrastructure

In addition to extending the benchmark capabilities, VerilogEvalV2 introduces a redesigned evaluation infrastructure, aimed at improving automation, reproducibility, and ease of experimentation. The entire evaluation workflow is managed through a Makefile-based framework. This allows users to configure the benchmark, select the desired evaluation task, specify the prompting strategy, and execute the complete generation and verification pipeline through a unified interface.

For each experiment, the framework automatically constructs the evaluation prompt, invokes the selected language model, collects the generated RTL implementation, and performs compilation and functional simulation using the provided verification testbenches. All generated outputs, prompts, execution logs, and simulation results are systematically recorded, enabling reproducible experiments and facilitating subsequent analysis.

The modular design of the evaluation infrastructure also simplifies the integration of new language models, prompting strategies, and future benchmark extensions. Combined with the newly introduced failure classification system, this infrastructure transforms VerilogEvalV2 from a simple benchmark into a comprehensive

experimental framework for evaluating RTL code generation systems.

3.1.4 CVDP

Despite the significant advances introduced by benchmarks such as VerilogEval and RTLLM, the evaluation of LLMs for hardware design remained largely focused on isolated RTL generation tasks. These benchmarks typically considered self-contained design problems and single-turn interactions, providing limited support for the iterative, multi-file, and tool-intensive workflows that characterize real-world hardware development. Moreover, the rapid progress achieved by state-of-the-art language models progressively reduced the discriminative power of existing benchmarks, particularly for evaluating emerging agentic systems.

The *Comprehensive Verilog Design Problems* (CVDP) benchmark [14] addresses these limitations. Rather than viewing hardware design as a single RTL generation task, CVDP models it as a collection of heterogeneous engineering activities spanning both design and verification. Similarly to previous RTL benchmarks, the correctness of generated designs is assessed through automated simulation-based functional verification. However, CVDP extends this evaluation paradigm to a much wider variety of hardware engineering tasks while explicitly supporting both Non-Agentic and Agentic evaluation settings, enabling the assessment of modern agentic systems alongside conventional language models. Even in the Non-Agentic setting, the tasks were selected in a way that they could be considered Agentic sub-tasks.

The benchmark consists of 783 problems authored from scratch by experienced hardware engineers, manually reviewed by domain experts, and subsequently filtered through an automated quality assurance process. By combining a large collection of human-authored tasks with a standardized evaluation framework, CVDP represents one of the first benchmarks specifically designed to evaluate realistic hardware engineering workflows rather than isolated RTL generation problems.

Dataset Construction

Software benchmarks like SWE-bench [11] build their tasks from existing open-source repository, in the RTL domain this is not easily done: there is a highly limited amount of public repository of good quality. For this reason it is impossible to create a good, representative, hardware benchmark from open-source projects.

For this reason the entire CVDP problem dataset was created from scratch by 35 experienced hardware design. The experts designed problems that they thought representative of their typical activities. The generated problems were then manually revised and automatically filtered. Not only were the golden solutions checked for correctness, but also the prompt was checked by LLMs to spot linguistic

ambiguities. Initially, 1313 were created, but only 783 passed the filtering phase.

Task Categories The benchmark datapoints are organized in 13 categories, designed to cover the most part of an hardware development project activities. CVDP does not limit itself to the evaluation of RTL code-generation, but it includes also code generation, debugging, code modifications and other activities, as seen at Table 3.1. These categories are also further divided into 2 macro groups:

- The Non-Agentic activities are traditional code generation or code comprehension problems, which could also be considered an agentic sub-task. The model gets all necessary information in the prompt and produces a single response.
- The Agentic activities are meant to reproduce more realistic real-life activities. The model needs to interact with a little repository with multiple directories and files, to explore the context and modify files.

This distinction allows the benchmark to evaluate the models performance in both agentic and non-agentic workflows, providing a comprehensive assessment of theirs capabilities in hardware development tasks.

Datapoints Structure Each problem in the CVDP benchmark is represented as a structured datapoint (see example in Appendix A), a little hardware project, containing all the necessary information for the model to understand and solve the task.

The first element is the **prompt**, a natural language description of the task and of the goal. In the Non-Agentic datapoints, the prompt contains all the information needed to solve the task, such as the functional specification, the eventual partial code, and the expected output requirements. In the Agentic datapoints, the prompt is more general and does not contain all the information needed to solve the task. It describes the goal, but most of the needed information are to be extracted from the context files by the Agent. The **context** can include pre-existing RTL configurations, functional specifications, and other relevant files that provide the necessary background for the task.

No matter the task category, each datapoint has a test harness, that can only be used in the automatic evaluation and to which the models do not have access. In this way, the benchmark clearly separates the information available to the model from the infrastructure used to evaluate its output, closely resembling the development workflow adopted in industrial hardware design.

Evaluation Methodology

The evaluation of the CVDP benchmark is independent from the problem category and from the workflow mode. The final evaluation criteria is the functional

Task ID	Description
cid02 RTL Code Completion	Completion of partially provided Verilog modules.
cid03 Natural Language Specification to RTL	Generation of Verilog code starting from a natural language description.
cid04 RTL Code Modification	Modification or extension of existing RTL modules based on new requirements.
cid05 Specification to RTL with Module Reuse	Generation of RTL code leveraging and integrating existing modules.
cid07 RTL Code Improvement (Linting / QoR)	Refinement of RTL code to improve quality (e.g., style, synthesizability, or performance).
cid12 Testbench Stimulus Generation	Creation of input stimuli for validating RTL designs.
cid13 Testbench Checker Generation	Generation of checkers to validate correctness of RTL outputs.
cid14 Assertion Generation	Creation of formal assertions for design verification.
cid16 Debugging / Bug Fixing	Identification and correction of errors in RTL code.

Table 3.1: CVDP task Categories

correctness, but the evaluation infrastructure is significantly more articulated than in previous works.

For each datapoint, the benchmark defines not only the problem, but also the context available to the model, the execution environment and the evaluation criteria. The entire evaluation process guarantees that all model are evaluated in the same experimental conditions.

The evaluation pipeline has three main phases: the task execution inside of a docker container, the functional verification of the generated solution, and the final scoring (see Figure 3.5). This pipeline allows for the evaluation not only of functional correctness, but also of the models ability to interact with a complex environment.

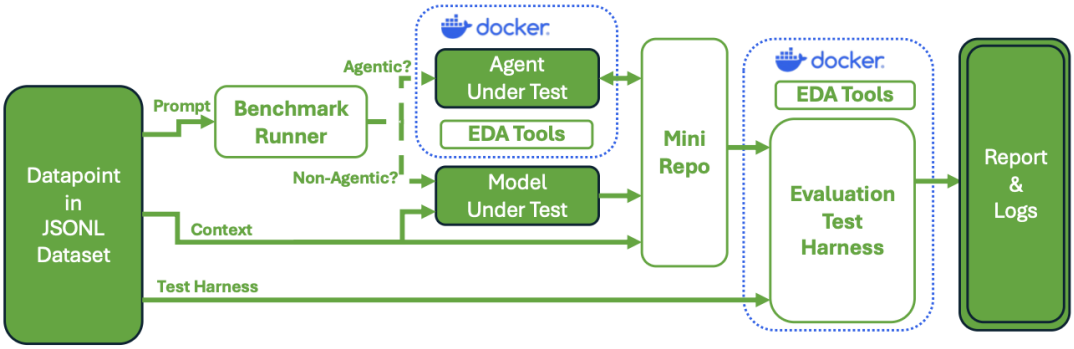


Figure 3.5: CVDP evaluation flow from [14]

Execution Environment A task execution is performed in a task execution environment, which is the work environment available to the model during the task execution, containing all and only the resources needed to solve the task. In the agentic tasks, this environment represents the repository that the agent can explore.

Once the task is completed, the generated solution is evaluated in a separate evaluation environment. In this phase the test harness is executed to verify the functional correctness of the generated solution. The test harness is a set of scripts for compilation, simulation and functional verification of the generated solution, and it is designed to be executed in a fully automated way. The test harness is not available to the model, and it is executed in a separate environment to ensure that the evaluation is independent from the task execution, and that there is no leakage of information and guarantee the fairness of the evaluation.

Evaluation Metrics Since CVDP encompasses thirteen categories covering heterogeneous hardware engineering activities, the benchmark does not rely on a single evaluation metric. Instead, the evaluation criterion is selected according to the objective of each task category, ensuring that the reported performance accurately reflects the capabilities required by the corresponding engineering activity.

In Code Generation tasks, the primary evaluation criterion is **functional correctness**. The generated solution is compiled and executed using the task-specific test harness, and a problem is considered successfully solved only if all verification tests pass. Model performance is therefore primarily reported using the *pass@1* metric, representing the percentage of tasks correctly solved on the first attempt. Differently, in Code Comprehension tasks the correctness of the generated responses is assessed by comparing them against reference solutions using metrics appropriate for the specific problem formulation or with the use of an LLM-judge. Each category employs evaluation criteria that best capture the intended objective of the task,

rather than forcing all problems into a single unified metric.

Finally, benchmark results are reported both as an overall score, as category-wise and difficulty-wise performance. This organization enables a more detailed analysis of model capabilities, highlighting strengths and weaknesses across the different hardware design and verification activities represented in the benchmark.

Strengths and Limitations

Compared with previous RTL benchmarks, CVDP represents a substantial step toward the realistic evaluation of hardware-oriented language models. Its main strength lies in the breadth of the covered tasks: rather than focusing exclusively on RTL generation, the benchmark encompasses the diverse activities routinely performed throughout the hardware design and verification flow, including code generation, debugging, testbench development, assertion generation, code modification, and design comprehension. This diversity enables a more comprehensive assessment of model capabilities and better reflects real engineering workflows.

Another important contribution is the explicit support for both non-agentic and agentic evaluation settings. By providing complete project repositories, supporting documentation, and executable development environments, CVDP enables the evaluation of modern coding agents that interact with external tools and perform iterative reasoning. This makes the benchmark significantly more representative of current industrial design practices than earlier single-prompt benchmarks.

The benchmark also provides a standardized and reproducible evaluation infrastructure based on containerized execution environments and automated functional verification. By clearly separating the task execution environment from the hidden evaluation harness, CVDP ensures fair comparisons across different models while preventing information leakage during task execution.

Despite these advantages, CVDP also presents several limitations. The benchmark primarily evaluates the functional correctness of the final solution and does not explicitly assess other important aspects of hardware design, such as implementation quality, timing, power consumption, area optimization, or code maintainability. Furthermore, although the benchmark supports agentic workflows, the evaluation remains based exclusively on the final outcome, without considering intermediate reasoning steps, tool usage efficiency, or the overall interaction strategy adopted by the agent.

Finally, while CVDP considerably expands the scope of previous RTL benchmarks, it still represents a controlled benchmark environment. Real industrial hardware development often involves significantly larger codebases, longer development cycles, collaborative design processes, aspects that remain outside the current scope of the benchmark.

3.2 LLMs for RTL Code Generation

The application of general-purpose LLMs to RTL code generation presents some difficulties in the fact that an HDL describes not a sequence of operations, but the structure and the concurrent behavior of hardware circuits. This means that correct generation does not only require the knowledge of the language syntax, but also the ability to correctly represent parallelism, sequential and combinatorial logic, temporal dependencies and cycle-accurate behavior. To these difficulties, the limited availability of data related to hardware design is a further problem for the creation of the corpora needed for LLMs pre-training. Compared to amply used software languages such as Python or C++, publicly available RTL code is rare because most industrial-level hardware projects are proprietary.

3.2.1 Domain-specific LLMs

To improve LLMs capacity in RTL code generation, a first strategy is to develop models specific to the hardware domain. Instead of training a model from zero, in these approaches the model starts already pretrained on a corpus of natural language and software code examples, and is then specialized on HDL datasets. In this way the model is able to maintain its original capabilities, while also obtaining a newfound knowledge of the characteristic syntax and patterns of hardware development.

One of the first works in this direction is VeriGen [5]: this work starts from a model specialized in code generation and fine-tunes it on a domain-specific corpus for Verilog coding. The dataset is made of GitHub code repositories and from books on Verilog and hardware development. The results show that domain-specific fine-tuning significantly improves the model's Verilog generation capabilities, achieving a 41% improvement in the generation of syntactically correct code compared to its pre-trained counterpart.

RTLCoder [37] is a later work that focuses not only on model specialization, but also on the construction and refinement of the dataset used for fine-tuning. The authors propose an automated pipeline to generate more than 27,000 instruction-code pairs, in which generated RTL solutions are filtered through syntax checking before being included in the training dataset. In addition, RTLCoder introduces a training scheme based on code-quality feedback, where multiple candidate solutions are evaluated and their scores are used to guide the model towards higher-quality generations.

These works highlight how the quantity and quality of domain-specific training data play a fundamental role in improving the RTL generation capabilities of LLMs.

The importance of the training dataset is further investigated in "Data Is All YouNeed" [38], who address the scarcity of aligned natural-language and Verilog

data through an automated data augmentation framework. Rather than relying only on code-completion examples, the proposed approach parses existing Verilog designs into Abstract Syntax Trees (ASTs), hierarchical representations of the syntactic structure of source code, where nodes correspond to language constructs such as declarations, expressions, and control structures. Predefined rules are then applied to the AST nodes to translate the corresponding Verilog structures into natural-language descriptions, producing aligned instruction-code pairs for fine-tuning. This alignment has a significant impact on functional correctness: on their evaluation setup, fine-tuning Llama 2-13B using only code-completion augmentation achieves a 25.7% pass rate, while the addition of program-analysis-based natural-language alignment increases it to 45.7%. These results further suggest that the effectiveness of domain specialization depends not only on the amount of available RTL data, but also on how closely the training samples represent the target generation task.

Beyond improving the construction of domain-specific datasets, BetterV [39] investigates how RTL generation can be explicitly guided towards desired design properties. The framework first applies domain-specific instruction tuning to adapt a pretrained code model to Verilog generation. It then introduces generative discriminators, trained to distinguish between designs according to specific objectives and used to guide the model during generation. This approach allows the generation process to incorporate hardware-specific objectives beyond syntactic correctness alone, including functional and implementation-related properties. BetterV therefore extends domain specialization from simply learning the distribution of Verilog code towards explicitly controlling the characteristics of the generated RTL.

More recent approaches further integrate functional verification into the model specialization process. QiMeng-SALV [40] observes that evaluating an RTL design only through module-level pass or fail feedback discards useful information, since an incorrect implementation may still correctly reproduce the behavior of a subset of its output signals. To exploit this information, the authors introduce signal-aware verification feedback, where individual output signals are evaluated and associated with the corresponding portions of the RTL implementation. This fine-grained information is then incorporated into preference optimization, allowing the model to learn not only from completely correct designs, but also from partially correct implementations. This represents a further step towards specialization techniques in which functional hardware verification directly contributes to the training signal.

Overall, these works show a progressive evolution of domain-specific RTL generation techniques. Early approaches mainly rely on fine-tuning pretrained code models on Verilog corpora, while later methods increasingly focus on the quality and alignment of training data and on incorporating hardware-specific feedback into the specialization process. In particular, functional verification provides a stronger supervision signal than syntactic correctness alone, allowing the training

process to better reflect the actual correctness requirements of RTL design.

3.3 Agentic Approaches

While domain-specific models aim to improve RTL generation by specializing the capabilities of the underlying LLM, a different line of research focuses on improving the generation process itself. In traditional text-to-RTL approaches, the model receives a hardware specification and directly produces an implementation, with limited opportunities to inspect, verify, and refine the generated design.

Agentic approaches extend this paradigm by embedding the LLM within an iterative workflow in which generation can be interleaved with reasoning, external tool execution, verification, and subsequent refinement. Instead of treating RTL generation as a single inference problem, these systems progressively develop a solution by observing the results of previous actions and using the obtained feedback to determine the next step.

This paradigm is particularly relevant to hardware design, where the correctness of generated RTL can be evaluated through existing EDA tools, including compilers, simulators, synthesis tools, and formal verification engines. Their outputs can therefore provide concrete feedback to the model, allowing errors to be identified and potentially corrected during the generation process rather than relying exclusively on the knowledge encoded in the model parameters.

The following works explore different implementations of this idea, ranging from general reasoning-and-action frameworks to increasingly specialized workflows designed specifically for RTL generation and verification.

3.3.1 ReAct

The rapid progress of LLMs demonstrated that they are capable of solving a wide range of reasoning tasks through natural language generation alone. Many real-world problems require more than a single forward inference: they involve gathering additional information, interacting with external environments, verifying intermediate results, and adapting the reasoning process based on newly acquired observations. Traditional prompting strategies, such as Chain-of-Thought, improve the reasoning capabilities of language models but remain limited to internal reasoning, preventing the model from actively interacting with its environment.

ReAct [9] is a **prompting framework** that tightly interleaves natural language reasoning with external actions. Rather than generating a complete solution in a single response, ReAct enables language models to iteratively reason about the current state of the problem, invoke external tools or information sources, observe their outputs, and incorporate the newly acquired information into subsequent reasoning steps.

Although originally proposed as a general-purpose prompting framework, ReAct has had a profound impact on the development of agentic systems. Its observation-reasoning-action paradigm has become the foundation of many modern AI agents, including coding assistants and hardware design agents, which extend the original framework with richer toolsets, persistent workspaces, and iterative code refinement capabilities.

Reasoning-Action Loop

ReAct was built around the idea of integrating the reasoning process with external actions. The model does not produce a complete solution in a single inference; instead, instead it alternates phases of reasoning and action, gradually building the final solution.

The ReAct framework operates through an iterative cycle in which the language model alternates between *Thought*, *Action*, and *Observation* steps. Unlike traditional Chain-of-Thought prompting, where reasoning is completed before producing the final answer, ReAct mixes reasoning, with interactions with the external environment.

At each iteration, the model first generates a *Thought*, representing its current reasoning process. Rather than aiming directly at the final solution, the Thought is used to analyze the current state of the problem, interpret previously gathered information, and decide the next step. In this way, reasoning becomes an evolving process that is continuously refined throughout execution. Based on this reasoning, the model produces an *Action*, corresponding to an executable command in the environment. The specific actions depend on the application domain. The execution of the action returns an *Observation*, which contains the outcome of the interaction with the environment. This observation is appended to the model's context and becomes the basis for the next reasoning step, allowing the model to revise its hypotheses, recover from previous mistakes, and adapt its strategy as new information becomes available.

The key contribution of ReAct lies in the tight integration between reasoning and acting: reasoning is used to guide the selection of appropriate actions (**reason to act**), while the information gathered through actions is exploited to improve subsequent reasoning (**act to reason**). This bidirectional interaction enables the model to solve tasks requiring both planning and iterative information gathering, overcoming several limitations of purely reasoning-based prompting approaches.

Strengths and Limitations

ReAct introduced a simple yet highly influential paradigm for combining reasoning and interaction within a single prompting framework. Its main contribution is the explicit integration of internal reasoning with external actions, enabling language

models to iteratively acquire new information, revise previous assumptions, and adapt their strategy throughout task execution. This substantially improves the model's ability to solve problems requiring sequential decision making, information retrieval, or interaction with dynamic environments.

The reasoning process ReAct produces though, is entirely prompt-driven, the quality of the generated trajectories strongly depends on the language model's capabilities and on the design of the few-shot demonstrations. The framework provides no explicit planning or memory mechanisms, making it difficult to handle long-horizon tasks that require maintaining complex state information across many interaction steps. ReAct also assumes relatively simple interactions with the environment, where each action immediately produces an observation that is directly incorporated into the next reasoning step. It does not address many practical aspects of modern agentic systems, such as persistent workspaces, structured tool orchestration, parallel tool execution, autonomous code modification, or iterative software development workflows. These capabilities have been introduced by subsequent agentic frameworks inspired by the ReAct paradigm.

3.3.2 OpenAGI

While ReAct introduces the reasoning-action paradigm, the framework ultimately remains a prompting strategy rather than a complete agentic system.

OpenAGI [41] is an open-source framework for developing and evaluating LLM-based agentic systems. The goal is to develop a general-purpose agentic framework where an LLM can plan and execute complex tasks by interacting with a variety of external tools and environments. The fundamental idea is that a single LLM does not need to have all the capabilities required to solve a complex task: in this work the LLM is only the orchestrator, while specialized models execute the operations belonging to their domain.

Architecture

The architecture of OpenAGI is built upon a clear separation between reasoning, planning, and execution. Unlike ReAct, where the language model iteratively alternates reasoning and actions, OpenAGI adopts a more modular design in which the LLM primarily acts as a planner, while the execution of individual operations is delegated to specialized external components referred to as *Domain Expert Models*.

The workflow starts with a user prompt describing the final goal. This prompt is processed by an LLM-based *Planner*, whose responsibility is to analyze the task, and decompose it into an ordered sequence of elementary operations. The planner does not directly solve the problem, but it generates an execution plan specifying which models or tools should be invoked and in which order.

Once the plan has been generated, OpenAGI enters the *Plan Execution* phase. Each step of the plan is executed by invoking the corresponding Domain Expert Model. The framework considers as expert models any specialized component capable of performing a specific task, including machine learning models, software tools, plugins, or external APIs. The output produced by each component is propagated to subsequent execution steps, enabling the construction of complex multi-stage workflows.

Finally, the output produced by the execution pipeline is evaluated against the ground truth using the benchmark-specific metrics. In addition to measuring the performance of the generated plan, this feedback can be exploited during training through the proposed *Reinforcement Learning from Task Feedback* (RLTF) framework, enabling the planner to progressively improve its planning strategy without requiring direct human supervision.

Strengths and Limitations

OpenAGI represents one of the earliest frameworks to formalize the development of general-purpose LLM agents. By separating planning from execution and enabling the orchestration of external expert models and tools, it established several architectural principles that have been adopted by many subsequent agentic systems.

Despite these contributions, OpenAGI also presents several limitations. The framework assumes the availability of suitable Domain Expert Models for each required capability, making its performance dependent on both the planner and the quality of the available expert models. Also, although it introduces tool orchestration, it focuses primarily on selecting and sequencing existing tools rather than on autonomous interaction with software development environments. Capabilities that are now common in modern coding agents are outside the scope of the original framework. While OpenAGI laid the foundations for modern agentic architectures, it does not directly address the software and hardware engineering workflows targeted by more recent domain-specific agents.

3.3.3 Veri Sure

Even considering the performance advances introduced by earlier agentic frameworks, the RTL code generation task is still critical. Most existing frameworks are based on iterative loops of generation and simulation, where the model generates a candidate solution, simulates it, and uses the simulation results to refine its next attempt. This paradigm is very effective, but it is also highly dependent on the availability of high-quality testbenches.

Veri-Sure [42] is a multi-agent framework for RTL code generation that integrates

simulation, waveform analysis and formal verification into a single iterative loop, without code re-generation. The goal is to have a workflow close to real-life traditional EDA workflows.

Workflow

Veri-Sure adopts a multi-agent workflow in which the RTL development process is decomposed into a sequence of specialized stages, each handled by a dedicated agent. Rather than relying on a single language model to repeatedly generate and debug the design, the framework assigns distinct responsibilities to multiple agents (Figure 3.6) that collaboratively perform specification analysis, RTL generation, verification, and debugging.

The workflow is centered around a shared Design Contract, which is a structured representation of the original specification, that serves as the common reference throughout the entire pipeline. This contract is generated before any RTL is produced and is subsequently used by all agents to preserve the original design intent, reducing semantic drift during iterative refinement. From this common specification, the framework generates both the verification collateral and the RTL implementation. Whenever the generated design fails verification, dedicated debugging agents analyze the observed failures and produce localized code patches instead of regenerating the entire module, enabling a more efficient and reliable repair process.

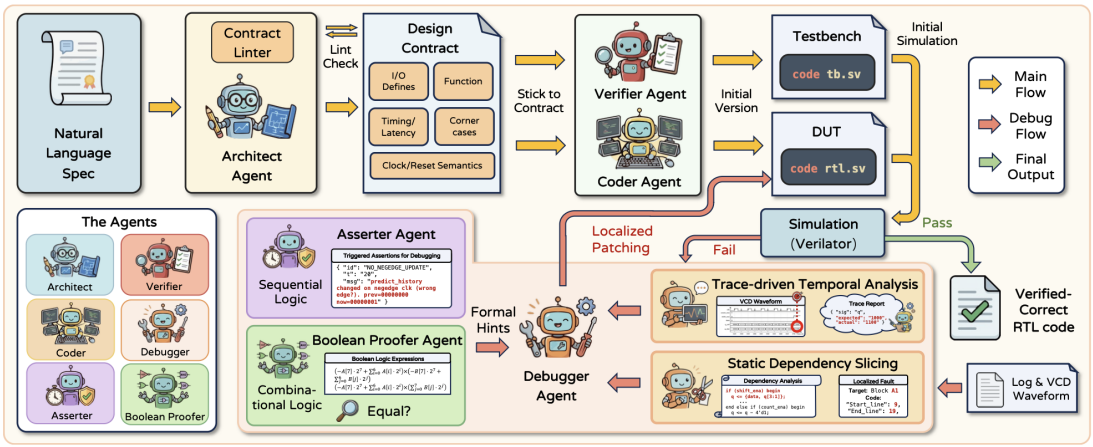


Figure 3.6: Picture from [42] paper: agents representation

Design Contract Generation The first stage of the Veri-Sure workflow, consists of generating a Design Contract, a structured representation of the design specification that serves as the foundation for all subsequent stages. Rather than

relying solely on the original natural language description, the framework prompts a language model to transform the specification into a machine-readable contract that explicitly captures the expected behavior of the design. By converting an often ambiguous textual specification into a structured representation, the Design Contract provides a common semantic reference that can be consistently interpreted by all agents participating in the workflow.

A key objective of this intermediate representation is to mitigate the semantic drift. During iterative code generation and debugging, language models may gradually deviate from the original design intent, as they focus on fixing individual verification failures. Since every stage of the workflow has access to the same Design Contract, modifications can always be validated against the original requirements, ensuring that local repairs remain consistent with the intended functionality instead of merely satisfying the current verification results.

The Design Contract acts as the shared source of truth for the entire framework, guiding both RTL generation and verification while providing a stable reference during the iterative debugging process.

Generation and Verification Once the design Contract is generated, the workflow is divided in two parallel flows:

- The RTL Generation Agent uses the contract to produce the RTL implementation of the required module
- The Verification Generator Agent uses the contract to produce the verification collateral, including testbenches, assertions, and coverage monitors.

The use of a single shared representation of the design intent ensures that both the generated RTL and the verification collateral are consistent with the original specification, reducing the likelihood of semantic drift and improving the overall reliability of the generated design.

Once the generation phase is complete, the generated RTL is compiled and executed together with the testbench produced. The result of the simulation is a first-level of project validation. If all tests pass, the design is considered functionally correct and the workflow terminates. Otherwise, all logs and information are collected and passed to the Iterative Debugging stage, which will the guide the next iterations.

Iterative Debugging The objective of this stage is to identify the root cause of the failure and apply targeted modifications while preserving the portions of the implementation that are already correct.

To achieve this, the framework combines multiple sources of diagnostic information. Besides simulation logs and compiler messages, the debugging agents analyze

execution waveforms to localize the temporal region in which the failure first occurs. Formal verification techniques are then employed to validate the suspected error locations and determine whether the observed behavior violates the constraints defined in the Design Contract.

Based on this analysis, the debugging agent produces **localized code patches** that modify only the relevant sections of the RTL implementation. This patch-based repair strategy reduces unnecessary changes to correct logic, mitigating semantic drift and lowering the risk of introducing new functional regressions during subsequent iterations. After each modification, the updated design is recompiled and verified again, and the debugging loop continues until all verification checks are satisfied or a predefined iteration limit is reached.

Strengths and Limitations

Veri-Sure extends previous LLM-based RTL generation frameworks by introducing a structured multi-agent workflow that tightly integrates design generation, verification, and debugging. Its main contribution lies in the use of a shared Design Contract to preserve the original design intent throughout the development process and in its patch-based debugging strategy, which performs localized code modifications instead of repeatedly regenerating the entire design. These features make the framework more closely resemble industrial hardware development workflows.

Despite these advantages, Veri-Sure remains primarily focused on RTL generation and functional repair. The framework assumes that the generated verification collateral is sufficiently comprehensive to expose design errors and does not address broader engineering activities such as repository navigation, long-term task planning, or interaction with complex software development environments. While it represents an important step toward agentic hardware design, it remains a domain-specific framework rather than a general-purpose coding agent.

3.3.4 EvolVE

Recent agentic frameworks have demonstrated that iterative generation and debugging can significantly improve RTL code generation, but they still rely on a relatively local refinement process, where each iteration attempts to repair the current implementation based on verification feedback. Such approaches may converge to suboptimal solutions and often struggle to escape local optima when the initial implementation is fundamentally flawed.

EvolVE [43] is an evolutionary framework that tries to address this limitation by approaching RTL generation as a search problem over the space of candidate implementations. Instead of repeatedly refining a single design, EvolVE maintains multiple candidate solutions and progressively explores the design space through

an iterative generate-evaluate-evolve cycle, as can be seen in Figure 3.7. This strategy enables the framework to balance exploration and exploitation, gradually improving both functional correctness and hardware quality.

The framework never adopts a patch based repair strategy, like most other frameworks, but it generates complete implementations at each iteration, using the information gathered in previous iterations to guide the generation of new candidates.

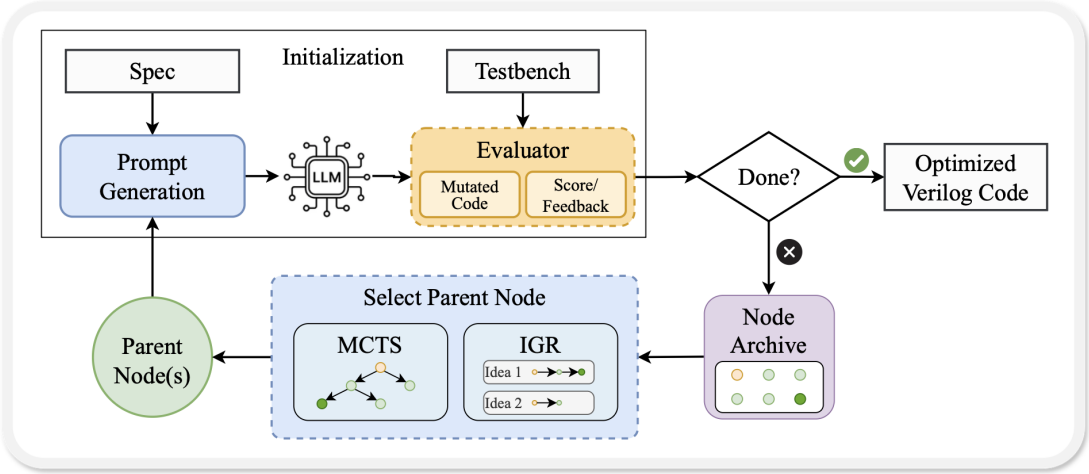


Figure 3.7: Picture from Evolve [43]: workflow representation

Evolutionary Workflow

EvolVE main idea is to approach RTL code generation as an evolutionary project. Instead of maintaining only the latest implementation, EvolVE stores every explored solution as a node in an archive of candidate implementations. Each node is represented as

$$N_i = (V_i, S_i, F_i),$$

where V_i denotes the generated RTL implementation, S_i is its evaluation score, and F_i represents the diagnostic feedback collected during verification. Rather than patching an existing implementation in place, each iteration selects one of these nodes and uses it as the starting point for generating a **new candidate**, which is then evaluated and added to the archive. In this way, the search process progressively builds a tree of increasingly refined implementations while preserving previously explored alternatives.

This means that instead of a single implementation to be iteratively corrected, the framework maintains a population of candidate solutions that are evolved

progressively through a generate-evaluate-evolve cycle. Each iteration uses the information gathered in the previous ones to guide the generation of new complete implementations. EvolVE does not explore the entire design space. Instead, it incrementally builds an archive of candidate implementations, where each new candidate is generated by refining a previously evaluated solution.

Initial Generation The first step of the EvolVE workflow is the generation of an initial population of candidate implementations. The framework uses an LLM to generate an initial set of candidates that are the initial population of the evolutive process.

Each generated RTL design is evaluated and stored as a node in the search archive together with its score and diagnostic feedback. These nodes form the starting point of the evolutionary process and provide the initial search frontier from which subsequent candidates will be generated.

Evolutionary Search The core of the EvolVE framework is the evolutionary search process, which iteratively refines the population of candidate implementations. At each iteratino, EvolVE selects one or more promising candidates and uses a language model to generate new variants based on them, using the feedback of previous evaluations. There are two main search strategies proposed:

- Idea-Guided-Refinement (IGR): the model is prompted to generate new idea that can improve the current solution, favoring the exploration of different implementations.

- Monte-Carlo Tree Search (MCTS): is used to sistematically search for promising candidates in the design space, balancing exploration and exploitation to identify high-quality solutions.

Evaluation and Refinement Each newly generated candidate is evaluated through the Structured Testbench Generation (STG) framework, which automatically produces structured testbenches and executes functional simulation. The resulting score and diagnostic feedback are associated with the corresponding node in the archive and are subsequently exploited by the search strategy to select the most promising candidate for the next iteration. The evolutionary process continues until a functionally correct implementation is found or the predefined computation budget is exhausted.

Strengths and Limitations

EvolVE, by maintaining an archive of candidate implementations and leveraging test-time compute, is able to progressively improve solution quality without requiring additional model training or fine-tuning. The integration of complementary search strategies, such as IGR and MCTS, further enables a balance between exploration of new design alternatives and exploitation of promising solutions.

Despite these advantages, EvolVE still relies on repeated full-code regeneration instead of localized code modifications, which may lead to unnecessary computation and reduced efficiency as the design complexity increases. Moreover, the framework remains focused on the RTL generation task and does not address broader software engineering activities, such as repository navigation, multi-file reasoning, or interaction with external development tools, that characterize modern coding agents.

3.3.5 ACE

As Large Language Models became increasingly capable of solving complex tasks through iterative reasoning and tool use, it was also recognized that improving the model itself is not always sufficient to improve the overall performance of an agentic system. Growing attention has been given to context adaptation, which is defined as the optimization of the information provided to the model at inference time without modifying its parameters. This paradigm is particularly relevant for agentic systems, where the model repeatedly interacts with the environment, receives feedback from previous actions, and accumulates knowledge throughout the execution.

Existing context adaptation techniques exhibit important limitations: most approaches rewrite the entire context after each iteration, condensing previously acquired knowledge into a new prompt or instruction. Zhang et al. [44], in their work on Agentic Context Engineering (ACE), identify two major issues with this strategy:

1. The first is **brevity bias**, namely the tendency of iterative optimization to progressively produce shorter and more generic prompts, gradually removing task-specific knowledge.

2. The second is **context collapse**, where repeated full-context rewriting unexpectedly compresses the accumulated knowledge into a much smaller prompt, causing a significant degradation in performance. Rather than preserving useful experience, valuable information may therefore be discarded during the adaptation process.

ACE is designed to address these limitations by providing a framework for scalable context adaptation in both offline and online settings. Instead of treating the prompt as a static instruction, ACE models it as an evolving context playbook: a structured collection of strategies, heuristics, domain knowledge, and previously acquired experience that continuously grows and is refined throughout the execution. This design follows the intuition that modern language models with large context windows benefit from detailed, structured knowledge rather than aggressively compressed summaries.

The ACE framework adopts an agentic architecture composed of three specialized components. The **Generator** solves the current task and produces the a reasoning trajectory. The **Reflector** analyzes both successful and unsuccessful trajectories to identify reusable strategies, common failure modes, and possible improvements. Finally, the **Curator** integrates these insights into the context playbook by performing localized updates rather than rewriting the entire prompt. This separation of responsibilities mirrors the human learning process of solving a problem, reflecting on the outcome, and consolidating the acquired knowledge for future tasks.

A key innovation introduced by ACE is the concept of **incremental delta updates**. Instead of regenerating the entire context after each iteration, the playbook is represented as a collection of structured context items that can be independently added, modified, or removed. Consequently, only the relevant portions of the context are updated while the remaining knowledge is preserved. This approach substantially reduces computational cost and adaptation latency while mitigating the information loss associated with monolithic context rewriting. Furthermore, ACE introduces a grow-and-refine mechanism that progressively expands the playbook with new knowledge while periodically removing redundant entries through lightweight non-LLM refinement.

Experimental results demonstrate that ACE consistently improves the performance of agentic systems across several reasoning benchmarks while significantly reducing both adaptation latency and inference cost. Overall, the work establishes structured context evolution as an effective alternative to parameter optimization, showing that continuously refining the execution context can substantially improve the capabilities of language-model-based agents without requiring additional model training.

3.3.6 ACE-RTL

Research on LLM-based RTL-design has followed two main directions: the development of general-purpose agentic frameworks and the development of RTL-specialized models. Both approaches have significant limitations. While the specialized LLM models are highly effective in generating RTL code, they are unable to interact with

the environment and to perform iterative reasoning. On the other hand, general-purpose agentic frameworks can interact with the environment, but they are not specialized in RTL design and therefore less effective in generating high-quality designs.

ACE-RTL [45] is an agentic framework that combines a specialised model for RTL design with a frontier LLM for reasoning and planning. It is based on the ACE paradigm, and it integrates the models complementary capabilities within an iterative agentic workflow.

Architecture

ACE-RTL does not rely on a single model for the entire process, but distributes the tasks among specialized components, each developed to perform a specific function. This separation is based on the observation that a single model cannot be expected to excel at all aspects of the design process, and that proficiency in RTL generation does not necessarily translate into effective reasoning or planning capabilities. The architecture of ACE-RTL is organized into three main components: the RTL Specialist Generator, the Coordinator, and the Reflector. The resulting workflow is iterative and agentic, with the Coordinator orchestrating the interactions between the Specialist Generator and the Reflector to progressively refine the design and adapt the context based on feedback from previous iterations, until a functionally correct solution is reached.

RTL Specialist Generator is responsible for producing the RTL implementation throughout the iterative design process. Given the hardware specification, together with the context maintained by the Coordinator (After the first iteration), the Generator synthesizes a complete Verilog design that represents the current candidate solution.

Unlike previous agentic frameworks that rely exclusively on a frontier language model, ACE-RTL employs a domain-specialized RTL model as the Generator, since a model specifically trained for RTL generation can more effectively capture the detailed domain knowledge required for hardware design." At every iteration, the Generator produces a new candidate implementation that is subsequently verified and refined through the agentic workflow.

Coordinator orchestrates the interaction between the Generator and the Reflector while implementing the principles of ACE. Its primary responsibility is to maintain the evolving context shared across iterations by integrating the diagnostic feedback produced by the Reflector.

The Coordinator tracks previously attempted solutions, detected failure patterns, and successful debugging strategies, ensuring that useful information is preserved

throughout the design process. Essentially, it monitors the overall optimization progress and can trigger a **restart** when repeated iterations indicate that the search has converged to an unproductive trajectory. In this way, the Coordinator enables the framework to progressively improve the generated RTL implementation while avoiding repeated mistakes and stagnation.

Reflector is responsible for analyzing unsuccessful implementations and identifying the underlying causes of failure. After each verification step, it receives the generated RTL code together with the compilation and simulation results, including compiler diagnostics, test failures, and execution logs.

Using the reasoning capabilities of a frontier language model, the Reflector produces a structured diagnostic report that explains why the current implementation failed and suggests high-level directions for improvement. Rather than modifying the RTL code directly, it provides semantic feedback that guides the Generator during subsequent iterations.

Iterative Design Workflow

ACE-RTL workflow is organized into an iterative loop (Figure 3.8) that alternates between design generation, verification, and improvement. Starting from the original hardware specification, the framework generates a first RTL implementation that is immediately compiled and functionally verified. The result of this testing is then used as the primary source that guides the following iterations. Through a continuous cycle of generation, analysis and context updating, the framework converges to a functionally correct solution.

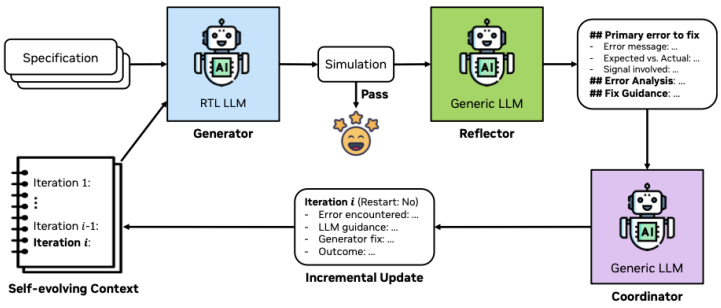


Figure 3.8: ACE-RTL iteration representation from [45]

Initial Generation The first phase of the workflow consists of the producing an initial solution. The generator receives the problem specification and generates a first complete implementation of the required modules. The main difference with

the models based on prompting, is that this implementation is only the starting point of the iterative process and its goal is not necessarily to satisfy all functional requirements

Verification and Reflection The generated code is compiled and verified within the benchmark infrastructure. Compilation errors, simulation logs and all data from the execution are collected and used as feedback. This information is then used by the Reflector, which uses its reasoning capabilities to identify the causes of failure and suggest high-level description of a specific modifications for the Generator to apply.

Context Evolution In the final phase, the Coordinator integrates the feedback from the Reflector into the evolving context. It does not just add messages to the conversation, but it updates a structured memory where are stored the important debugging information.

The updated context is then used by the Generator to produce a new RTL implementation for the next iteration.

If the Coordinator detects stagnation, characterized by the repetition of the same errors across multiple iterations, the framework can redirect the search trajectory and restart the generation process, encouraging the exploration of a different solution.

Parallel Scaling In addition to the iterative workflow executed by a single agent, a parallel scaling strategy (Figure 3.9) is introduced to further improve the probability of finding a functionally correct implementation. This approach exploits the inherently stochastic nature of Large Language Models, as independent executions starting from the same specification may generate different initial implementations and consequently explore different debugging trajectories.

Rather than relying on a single agentic workflow, ACE-RTL launches multiple independent agent instances in parallel, each independently generating its own initial implementation. Every instance executes its own iterative design loop, maintaining an independent evolving context while performing generation, verification, and reflection autonomously.

The parallel execution terminates as soon as one of the instances successfully passes functional verification. By simultaneously exploring multiple regions of the solution space, this strategy reduces the likelihood of all agents converging to the same unsuccessful trajectory. Experimental results reported by the authors show that increasing the number of parallel instances consistently improves the success rate, demonstrating that parallel scaling effectively complements the iterative context evolution process.

Unlike EvolVE, which explicitly explores multiple candidate solutions within a single evolutionary search process, ACE-RTL performs the exploration by executing multiple independent agentic workflows in parallel, each maintaining its own evolving context.

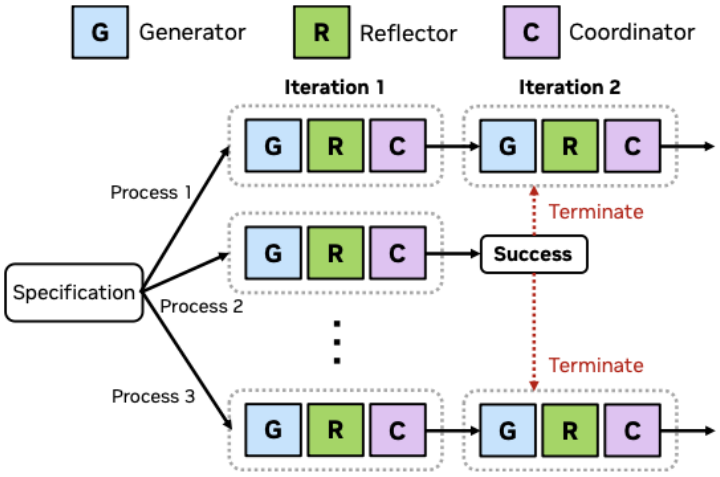


Figure 3.9: Parallel scaling representation from [45]

Strengths and Limitations

ACE-RTL represents one of the most advanced agentic frameworks for RTL generation by combining domain-specialized code generation, iterative reasoning, and context evolution within a unified architecture. The integration of a specialized Generator with a frontier LLM for reflection enables the framework to effectively exploit both hardware-specific knowledge and advanced reasoning capabilities. Furthermore, the introduction of parallel scaling allows the framework to explore multiple independent solution trajectories simultaneously, significantly improving the probability of converging to a correct implementation.

On the other hand, several aspects should be considered when interpreting the reported results. First, the agent performs its iterative refinement using the official benchmark verification infrastructure, including the reference testbench. While this design reflects a realistic hardware debugging workflow, it also provides access to verification artifacts that are unavailable to non-agentic approaches, making direct comparisons less straightforward. Moreover, the reported performance is measured using the Agent Pass Rate (APR), which evaluates whether at least one of multiple parallel agent executions successfully solves the task. Consequently, APR is conceptually closer to a pass@N metric than to the pass@1 evaluation

commonly adopted in prior RTL generation benchmarks and should therefore be interpreted accordingly when comparing ACE-RTL with earlier approaches.

Chapter 4

Methodology

Evaluating LLM-based systems for RTL code generation requires a controlled experimental environment in which different approaches can be compared under consistent conditions. This becomes particularly important for agentic systems, whose performance depends not only on the underlying language model, but also on the interaction with the repository, the available tools, and the iterative execution workflow.

For this reason, the experimental methodology of this work was designed around a common benchmark infrastructure, providing the evaluated systems with the same RTL design tasks and relying on a common verification procedure to determine their functional correctness. The CVDP benchmark was selected as the basis of this infrastructure and adapted to support both direct LLM inference and different agentic coding systems. This setup enables their behavior and performance to be evaluated within a shared and reproducible experimental framework.

4.1 Benchmark Customization

The experimental evaluation was conducted using version 1.0.2 of the CVDP benchmark, the original benchmark infrastructure was maintained as the basis of the experimental environment and subsequently extended to integrate the models and agentic systems considered in this work.

The required adaptations mainly concerned three aspects. First, the datapoints related to RTL code generation were selected, and the verification environment was adapted to support the different Cocotb versions required by their test harnesses. Cocotb is an open-source framework for verifying digital hardware designs, it allows testbenches to be written in Python and interact with a simulator to drive input signals, observe outputs, and verify the functional behavior of the design. Second, the wrappers required to integrate non-agentic models through the interface

provided by CVDP were implemented. Finally, the agentic workflow was extended to support the different coding agents considered in the evaluation and to control their access to the testbench during execution.

4.1.1 Dataset Selection

The experimental evaluation was restricted to the code-generation tasks included in CVDP, as the objective of this work is to evaluate the ability of language models and agentic systems to generate and modify RTL implementations. Datapoints belonging to the code-comprehension tasks were therefore excluded from the evaluation.

The resulting experimental dataset consists of 160 datapoints belonging to seven categories: cid003, cid004, cid005, cid012, cid013, cid014, and cid016 (their description at Table 3.1). Among them, 92 datapoints can be verified using the non-commercial toolchain, while the remaining 68 require commercial EDA tools for verification and were evaluated using Cadence Xcelium.

During the preliminary validation of the benchmark, an incompatibility was identified between the test harnesses of different datapoints and the available Cocotb versions. In particular, some test harnesses require Cocotb v1, whereas the remaining ones use Cocotb v2. To preserve the original benchmark tests without modifying their implementation, the 92 non-commercial datapoints were therefore divided into two subsets according to the required Cocotb version. The Cocotb v1 subset contains 16 datapoints, while the Cocotb v2 subset contains 76 datapoints.

This subdivision is purely an implementation requirement and does not represent a distinction between different experimental configurations. Both subsets were executed for each evaluated configuration, and their results were subsequently merged before computing the evaluation metrics. The complete distribution of the datapoints used in the experiments is reported in Table 4.1.

4.1.2 LLM Integration

The non-agentic evaluation workflow of CVDP was extended to support the LLMs considered in this work, namely Claude models, GPT and Gemini. The integration was implemented following the model interface provided by the benchmark, by defining the corresponding model factory and wrapper classes. This allowed the models to be used within the existing CVDP execution pipeline without modifying the benchmark evaluation procedure.

For each datapoint, the benchmark constructs the model input from the task specification and performs a single inference through the corresponding wrapper. The generated response is then processed using the CVDP parsing utilities to extract the RTL implementation and remove additional formatting that may be included

Category	Cocotb v1	Cocotb v2	Commercial
cid003	11	23	3
cid004	1	24	–
cid005	4	18	3
cid012	–	–	16
cid013	–	–	18
cid014	–	–	28
cid016	–	11	–
Total	**16**	**76**	**68**

Table 4.1: Distribution of the CVDP code-generation datapoints used in the experimental evaluation.

in the model output, such as Markdown code fences or other text surrounding the generated code. The resulting implementation is written to the expected design file and subsequently evaluated using the official test harness associated with the datapoint.

To distinguish failures of the generated implementation from temporary failures of the model interface, API requests were retried up to three times in case of communication or API errors. In this non-agentic pipeline, if no valid API response could be obtained after the retries, the execution was considered an infrastructure failure rather than a failure of the generated RTL code and was therefore excluded from the model performance evaluation.

4.1.3 Agent Integration

The original agentic workflow provided by CVDP was extended to integrate the four coding agents considered in this work: OpenCode, Claude Code, Claw Code, and ChipMind Agents. All agents were integrated within a common Docker-based execution infrastructure, while preserving their native agentic workflows, tool-management mechanisms, and internal reasoning strategies. Except for an additional OpenCode experiment, aimed at understanding the impact of model scaling, Claude Sonnet 4.6 was used as the underlying language model for the agentic systems.

For each datapoint, CVDP creates a dedicated workspace that is mounted inside the agent container under the `/code` directory. The workspace exposes the task specification through `prompt.json`, together with the `docs/`, `rtl/`, `verif/`, and `rundir/` directories. The `rtl/` directory contains the design files that the agent is expected to modify, while the remaining directories provide the contextual and verification collateral associated with the task. The agent operates directly on this workspace using its native file-editing and shell tools and terminates after

completing its execution.

To provide the different agents with a consistent task formulation, a common prompt was used across all integrations. In addition to the original CVDP problem specification, the prompt explicitly identifies the repository root and its directory structure and instructs the agent to inspect the available files and directly modify the RTL implementation required to solve the task. No attempt was made to standardize the internal behavior of the evaluated systems beyond this common interface. Each agent retained its native reasoning loop and tool set; consequently, agents were free to inspect files, execute shell commands, compile the RTL design, or create and execute additional tests when supported by their environment.

The integration layer differs slightly according to the interface exposed by each system. Claude Code was integrated through its Python SDK, allowing its file manipulation and shell tools to operate directly on the mounted workspace. OpenCode and Claw Code were instead executed through their respective command-line interfaces in non-interactive mode, with thin wrappers responsible for constructing the common task prompt, starting the agent, and collecting its execution output. ChipMind Agents was integrated through the driver provided with the system and treated as a black-box agent, since its internal implementation is proprietary. These differences were limited to the interface required to launch each system and did not modify their internal agentic workflows.

Once an agent terminates, its modifications to the workspace are preserved and the resulting RTL implementation is evaluated independently by the official CVDP scoring pipeline. The evaluation harness is executed only after the agent execution has completed and is therefore separated from the agentic loop. Consequently, in the standard testbench-unaware (TU) configuration, agents can use the tools available in their environment, including RTL compilers and simulators, but cannot inspect or execute the golden evaluation harness used to determine the final result.

Finally, the integration layer collects execution information produced by the agents, including their available reasoning traces, tool interactions, and execution logs. These artifacts do not affect the final pass/fail evaluation, but are retained to support the subsequent analysis of unsuccessful executions.

Testbench Awareness To investigate the effect of verification feedback on agentic RTL generation, an additional testbench-aware (TA) configuration was introduced. In this configuration, the benchmark evaluation sources and the corresponding Docker Compose configuration are additionally exposed to the agent through `/code/src` and `/code/docker-compose.yml`. The agent can therefore inspect the official verification code and, if autonomously selected as part of its reasoning process, execute the test harness during its iterative workflow. Importantly, CVDP does not explicitly invoke the harness on behalf of the agent during this phase: access to the verification infrastructure only makes it available as

an additional resource, while the decision of whether and how to use it remains part of the agent's native behavior. The independent CVDP evaluation is still executed after the agent terminates and remains the final criterion for determining correctness.

4.1.4 Execution Environment

The execution environment used for the final verification of the generated designs is separated from the environment in which the agents operate. After a model or an agent produces an RTL implementation, CVDP evaluates it through an independent Docker-based scoring environment containing the official verification infrastructure associated with the datapoint. This separation ensures that the final pass/fail result is determined by the benchmark rather than by the outcome of any compilation or simulation autonomously performed by the agent during its execution.

For each datapoint, CVDP dynamically generates the corresponding Docker Compose configuration according to the verification requirements of the task. The design files, documentation, verification files, runtime directory, and the official evaluation sources are mounted into the scoring environment. Unlike the agent workspace, the scoring environment therefore always has access to the complete golden test harness required to evaluate the generated implementation.

Two main verification environments were used depending on the selected datapoint. The 92 non-commercial datapoints were evaluated using open-source simulation tools and separate environments were maintained for the Cocotb v1 and Cocotb v2 subsets in order to preserve compatibility with their original test harnesses. In particular, a dedicated Docker image based on the CVDP open-source simulation environment was configured with Cocotb v1 for the 16 datapoints requiring the older version, while the remaining 76 datapoints were evaluated using the Cocotb v2 environment.

The remaining 68 datapoints require commercial EDA tools and were evaluated using a dedicated Cadence environment. A custom verification image was configured to provide the dependencies required by the CVDP harness, while the Cadence installations were mounted read-only from the host system. The commercial verification containers were additionally configured to access the Cadence license server through the host network. This setup allowed the commercial datapoints to be evaluated using Cadence Xcelium while preserving the same benchmark-level scoring workflow adopted for the non-commercial tasks.

It should be noted that exposing the golden harness to an agent does not necessarily provide the complete environment required to execute it. In particular, for commercial datapoints, the Cadence tools and license configuration are available

only within the independent scoring environment. Therefore, even in the testbench-aware configuration, agents can inspect the commercial verification harness but cannot directly reproduce the complete golden verification procedure during their execution. For non-commercial datapoints, instead, the agent environment provides the open-source simulation tools required by the corresponding harness, allowing an agent to execute the available verification procedure autonomously when operating in the testbench-aware configuration.

All experiments were executed on the same host machine, a Linux-based server with 32 cores and 125GB of RAM, using Docker to isolate individual benchmark executions. The verification environment and the agent execution environment remained logically separated throughout the evaluation: the former was responsible for producing the official benchmark result, while the latter provided the workspace and tools available to the evaluated coding agent.

4.2 Experimental Protocol

The experimental protocol was designed to evaluate the performance of coding agents on RTL code-generation tasks under consistent conditions. All systems were evaluated on the same set of CVDP datapoints and their generated implementations were assessed using the official benchmark scoring procedure described in the previous section.

The evaluation primarily focuses on agentic systems and on their ability to autonomously interact with the provided design environment, modify the RTL implementation, and exploit the tools available during execution. In addition to the standard configuration, the experimental setup also considers the effect of providing agents with access to the official verification harness, allowing the impact of testbench availability on their behavior and performance to be investigated. Non-agentic LLM generation was also evaluated as a reference point, using the standard CVDP inference workflow.

4.2.1 Evaluation Configurations

The main experimental evaluation considers four coding agents: OpenCode, Claude Code, Claw Code, and ChipMind Agents. Claude Sonnet 4.6 was used as the underlying language model for all four systems, providing a common model backbone while preserving the native architecture, tools, and execution workflow of each agent. Each system was executed independently on the selected CVDP datapoints, starting from the original files provided by the benchmark.

Claude Code, and ChipMind Agents were evaluated in both TU and TA configurations, allowing the effect of access to the golden verification harness to be investigated. Claw Code and OpenCode were instead evaluated only in the

testbench-aware configuration for time reasons. Except for the information exposed through the workspace, the two configurations preserve the same agent integration and execution environment.

In addition to the main agent comparison, a limited set of complementary experiments was performed to investigate the effect of the underlying language model. OpenCode was also evaluated using Claude Sonnet 3.7, allowing its performance to be compared with the corresponding Claude Sonnet 4.6 configuration while keeping the agentic framework unchanged. Non-agentic generations using Claude Sonnet 3.7 and Claude Sonnet 4.6 were additionally collected through the standard CVDP inference workflow and used as a reference for the experimental analysis.

4.2.2 Evaluation

Each agentic configuration was executed once for each datapoint. Agents were allowed to operate autonomously until task completion or until the termination conditions defined by the corresponding system were reached. The native execution limits of the evaluated agents were preserved in order to avoid modifying their original execution behavior.

At the end of each execution, the final state of the RTL implementation was evaluated independently using the official CVDP scoring environment. Any compilation, simulation, or verification performed autonomously by an agent during its execution was considered exclusively as internal feedback and did not contribute directly to the final outcome. A datapoint was considered successfully solved only if the final implementation passed the complete CVDP golden verification harness.

Failures resulting from the operational limits of an agent (e.g. context exceeding the output limits) were considered unsuccessful executions, as these limits are part of the behavior of the evaluated system and temporary API or infrastructure failures unrelated to the generated implementation were retried up to three times. If a valid execution could not be obtained after these attempts, the datapoint was classified as a failure of the generated RTL implementation.

Datapoints affected by issues in the benchmark infrastructure were retained in the evaluation. Such issues were identified and classified separately during the failure analysis, rather than being used to retrospectively exclude datapoints from the experimental dataset. This choice avoids modifying the evaluated dataset based on information obtained only after inspecting the unsuccessful executions.

4.3 Evaluation Metrics

The evaluation of the considered systems is primarily based on functional correctness, determined by the outcome of the official CVDP test harness. However, the non-agentic and agentic settings require slightly different evaluation metrics due to the

different nature of their execution. Non-agentic model performance is evaluated using the standard pass@k metric, whereas agentic systems are evaluated through the Agentic Pass Rate (APR).

4.3.1 Pass@k

is a metric commonly used to evaluate the functional correctness of code generated by language models. Given (n) independently generated samples for a problem, of which (c) successfully pass the functional verification, pass@k estimates the probability that at least one correct solution is obtained when (k) samples are considered. It is defined as:

$$\mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \qquad n \geq k$$

where (n) denotes the total number of generated samples, (c) the number of functionally correct samples, and (k) the number of samples considered for the evaluation. This formulation is also adopted by VerilogEval for evaluating RTL code generation.

Particular attention must be paid to the distinction between (k), which defines the metric being estimated, and (n), which represents the number of samples used to estimate it. Consequently, pass@1 can be evaluated using different values of (n).

With (n=1), each problem is evaluated from a single generated solution and therefore contributes either 1 or 0 depending on whether that solution passes the functional verification.

With (n=5), five independent solutions are generated for each problem and used to obtain a more robust estimate of pass@1.

This distinction is particularly relevant in this work because the reference CVDP results report pass@1 using (n=5), whereas the non-agentic experiments conducted in this study use a single generation per datapoint ((n=1)). Both results estimate single-sample success, rather than the probability of obtaining at least one successful solution across five attempts, which would instead correspond to pass@5. The different sampling procedures should be considered when interpreting and comparing the reported results.

4.3.2 Agentic Pass Rate

Agentic systems are evaluated using the Agentic Pass Rate (APR), defined as the fraction of datapoints successfully solved by the agent:

$$\frac{N_{\text{passed}}}{N_{\text{total}}}$$

where (N_{passed}) is the number of datapoints whose final implementation passes the official CVDP test harness and (N_{total}) is the total number of evaluated datapoints.

In the experimental setup adopted in this work, each agent is executed once for each datapoint. A datapoint is therefore considered successful if the repository produced at the end of that agentic execution passes the complete functional verification, and unsuccessful otherwise. From an aggregation perspective, APR is consequently equivalent to a pass@1 evaluation with (n=1): each datapoint contributes either 1 or 0 to the final score.

Despite this numerical correspondence, the two metrics characterize substantially different execution processes. In the non-agentic setting, a trial consists of a single model inference producing the candidate RTL implementation. An agentic trial represents an entire problem-solving trajectory, potentially involving multiple model calls, file inspections and modifications, tool invocations, and iterative refinement based on intermediate feedback. The APR metric measures the probability of successfully completing a task through a complete agentic execution rather than the correctness of an individual model generation. This distinction must be considered when comparing non-agentic and agentic performance, even when both metrics are computed from a single trial per datapoint.

4.4 Failure Analysis

In addition to the quantitative evaluation of functional correctness, a failure analysis was conducted to investigate the causes preventing agentic systems from successfully completing the assigned RTL design tasks. The analysis considers the unsuccessful agentic executions of the testbench-unaware non-commercial configuration, and aims to identify the root cause responsible for each failure, rather than relying exclusively on the final error reported by the verification harness.

The analysis was performed in two stages: First, information from the CVDP execution was automatically collected and used to obtain an initial characterization of each failure, this included the outcome of the official verification harness, compiler and simulator outputs, and the errors produced during the evaluation. The information was then complemented by a manual inspection of the failed executions. Depending on the failure, this inspection considered the final RTL implementation produced by the agent together with its execution trajectory, including the available reasoning traces, file modifications, tool invocations, and intermediate compilation or simulation results.

Each unsuccessful execution was assigned a single root cause. When multiple errors were observed within the same execution, the classification was based on the earliest error considered responsible for preventing the agent from successfully

completing the task, rather than on secondary errors resulting from it. The classification taxonomy was not defined a priori, but was progressively developed during the analysis as recurring failure patterns emerged from the inspected executions.

The resulting taxonomy consists of nine failure categories as seen at Table 4.2 These categories distinguish failures caused by the generated RTL implementation from failures related to the interaction with the task environment, limitations of the verification toolchain, benchmark issues, and the behavior of the agentic workflow itself.

Particular attention was given to failures originating from the benchmark rather than from the evaluated agent. Issues found in the CVDP verification infrastructure were classified as *Benchmark issues* and retained in the experimental results. These datapoints were not re-evaluated after modifying the corresponding test harness, since the presence of a benchmark issue does not imply that the generated implementation would otherwise have been correct: an execution may contain independent design errors occurring before the benchmark-related issue is reached. Maintaining these datapoints also avoids retrospectively modifying the evaluated dataset based on information obtained during the failure analysis.

This enables the analysis to go beyond their aggregate pass rates and investigate whether differences in performance are associated with different failure modes and problem-solving behaviors.

Table 4.2: Operational definitions of failure categories

Category	Operational definition
Functional	RTL compiles and executes, but its behavior does not satisfy the functional specification.
Interface	A port or signal is missing, misnamed, or of the wrong width, causing an immediate elaboration crash or attribute/access error before functional simulation can be evaluated.
Compilation	The generated Verilog/SystemVerilog contains a syntax or semantic error, or references a missing submodule, so the simulator (Icarus Verilog) refuses to compile the design.
Pre-Simulation	The simulation run produces no usable output (missing `result.xml`, empty `sim.log`) before any test outcome is recorded; the underlying root cause is not reconstructible from the available aggregated data.
Benchmark	The failure originates in the CoCoTB testbench/harness itself (e.g., a faulty clock generator, misuse of the `to_unsigned` API on 1-bit signals), not in the agent-generated RTL.
Filename	The agent writes the design under an unexpected filename or extension (e.g., `.v` instead of `.sv`), so the harness cannot locate the file to simulate it.
Icarus	The RTL uses a valid SystemVerilog construct that Icarus Verilog does not support (e.g., constant selects inside `always_*` blocks), causing a simulator-specific failure unrelated to RTL correctness.
Runtime	A signal remains in an undefined X/Z state during a check that requires a defined 0/1 value, causing the simulation to fail during execution rather than at compile time.
Loop Failure	The agent's iterative loop terminates without completing (e.g., API timeout, rate limiting, max-turns reached, doom loop) before producing a final verdict, so no RTL-level root cause can be evaluated.

Chapter 5

Results

5.1 Experimental Setup

The non-agentic evaluation was performed on 488 datapoints using Claude Sonnet 3.7 and Claude Sonnet 4.6, of which 301 datapoints can be evaluated exclusively using open-source verification tools, while the remaining 187 require commercial EDA tools. In this setting, a single generation was produced for each datapoint, corresponding to a pass@1 evaluation. Evaluating two successive generations of the same model family also enables the analysis of the impact of model scaling on RTL task performance.

The agentic evaluation includes Claude Code, OpenCode, Claw Code, and ChipMind Agents. The considered dataset comprises a total of 160 datapoints, of which 92 can be evaluated exclusively using open-source verification tools, while the remaining 68 require commercial EDA tools. The main experiments use Claude Sonnet 4.6 as the underlying language model, while OpenCode was additionally evaluated using Claude Sonnet 3.7, enabling the impact of model scaling to be investigated within an agentic workflow as well. OpenCode was configured with a maximum execution time of 600 seconds per task, whereas Claude Code, Claw Code, and ChipMind Agents were limited to a maximum of 50 agentic iterations. None of the evaluated executions reached the maximum iteration limit.

All experiments were performed using the same Docker-based CVDP infrastructure, ensuring a consistent execution and evaluation environment across the evaluated systems. The agents operated interacted directly with the files available within each datapoint repository. Depending on the experimental configuration, agents were evaluated under either *testbench-aware* or *testbench-unaware* conditions.

During an agentic execution, the system can inspect and modify the RTL files and autonomously invoke the tools available in its environment. Any verification performed by the agent during this process serves exclusively as internal feedback

for its problem-solving workflow and does not determine the final outcome of the task. At the end of each execution, the final state of the repository is independently evaluated using the official CVDP test harness. The outcome of each datapoint is therefore determined exclusively by this final evaluation.

5.2 Agentic Systems Comparison

At the aggregate level, considering both commercial and non-commercial datapoints, the three agentic systems achieve comparable performance in the benchmark unaware configuration (look at Table 5.1): Claude Code obtains the highest overall pass rate at 39.4%, followed by ChipMind at 38.1% and OpenCode at 36.3%. The difference between the best and worst performing systems is limited to 3.1% APR points, indicating that none of the evaluated agents exhibits a substantial advantage.

5.2.1 Task Difficulty and Categories

The relatively small differences in aggregate performance, conceal pronounced category-dependent differences (the task categories can be seen at Table 5.5). OpenCode achieves the highest pass rate on RTL Modification (cid004), reaching 64.0%, compared with 60.0% for Claude Code and 56.0% for ChipMind. Claude Code shows the strongest performance on Assertion Generation (cid014), with a pass rate of 42.9%, compared with 28.6% for ChipMind and 25.0% for OpenCode. ChipMind obtains significantly higher results on both testbench-related categories, reaching 18.8% on Testbench Stimulus Generation (cid012) and 11.1% on Testbench Checker Generation (cid013), although the overall performance of all systems remains low on these tasks.

For other categories, the differences between agents are considerably smaller. Performance on Natural-Language Specification to RTL (cid003) ranges from 43.2% to 45.9%, while all three systems obtain identical pass rates on Specification and Reuse to RTL (cid005) and RTL Debugging/Bug Fixing (cid016), at 48.0% and 54.5%, respectively. These results show that no agent consistently outperforms the others across all task categories: the differences between the evaluated systems are more evident in their category-specific behavior than in their aggregate pass rates.

A clear relationship emerges between task difficulty and failure frequency. Across the three agents, unsuccessful executions account for 29% of the agent–task evaluations involving easy problems, increasing to 48% for medium problems and 65% for hard problems. The same general trend is visible at the individual-agent level, with hard problems producing the highest failure rate for all three systems. More interestingly, task difficulty is also associated with a change in the nature of the observed failures. Only 27% of failures on easy problems are classified as

Table 5.1: Testbench-unaware performance for commercial and non tasks

Category	**# DP**	**OpenCode**	**ChipMind**	**Claude Code**
cid003	37	45.9%	43.2%	43.2%
cid004	25	64.0%	56.0%	60.0%
cid005	25	48.0%	48.0%	48.0%
cid012	16	0.0%	18.8%	6.3%
cid013	18	0.0%	11.1%	5.6%
cid014	28	25.0%	28.6%	42.9%
cid016	11	54.5%	54.5%	54.5%
Overall	160	**36.3%**	**38.1%**	**39.4%**

RTL-related, compared with 37% for medium problems and 67% for hard problems. While failures on easy and medium tasks are predominantly associated with execution-related causes, failures on hard problems are more frequently attributable to the generated RTL itself.

Table 5.2: Failure patterns across CVDP difficulty levels for the three agentic configurations.

Difficulty	OpenCode	Claude Code	ChipMind	Overall
Easy	4 (24%)	4 (24%)	7 (41%)	15 (29%)
Medium	26 (47%)	29 (53%)	24 (44%)	79 (48%)
Hard	12 (60%)	12 (60%)	15 (75%)	39 (65%)

5.2.2 Commercial and Non-Commercial Dataset Differences

A more pronounced difference emerges when separating the non-commercial and commercial subsets of the benchmark (see Table **??**). On the 92 non-commercial datapoints, OpenCode achieves the highest pass rate at 54.3%, followed by Claude Code at 51.1% and ChipMind at 50.0%, as shown at Table 5.3. Performance decreases substantially on the 68 commercial datapoints (as can be seen at Table 5.4), where Claude Code achieves 23.5%, ChipMind 22.1%, and OpenCode 11.8%.

The decrease is observed for all three systems, but its magnitude differs considerably. Compared with the non-commercial subset, the pass rate decreases by 27.6% for Claude Code and 27.9% for ChipMind, while OpenCode exhibits a substantially larger reduction of 42.5%. The relative ranking of the systems changes across the two subsets: OpenCode achieves the highest performance on the non-commercial datapoints but the lowest performance on the commercial ones, whereas Claude Code obtains the highest pass rate on the commercial subset.

These results also clarify the aggregate performance reported: the overall ranking on the complete dataset (in Table 5.1) is influenced by the substantially lower success rates obtained on the commercial datapoints (presented in Table 5.4), and therefore does not fully reflect the behavior observed on the non-commercial subset (presented in Table 5.3). The difference between the two subsets cannot be attributed exclusively to the use of commercial verification tools since their task composition also differs: in particular, Testbench Stimulus Generation (cid012), Testbench Checker Generation (cid013), and Assertion Generation (cid014) are represented only in the commercial subset. The observed performance gap reflects both the increased difficulty of the commercial datapoints and differences in the

types of tasks included in the two subsets. This could be partially explained by the need in the commercial task to use a specific tool, of which the model could have not seen a lot of examples in the training phase.

Table 5.3: Testbench-unaware performance on the non-commercial subset of CVDP.

Category	# DP	OpenCode	ChipMind	Claude Code
cid003	34	50.0%	44.1%	44.1%
cid004	25	64.0%	56.0%	60.0%
cid005	22	50.0%	50.0%	50.0%
cid016	11	54.5%	54.5%	54.5%
Overall	**92**	**54.3%**	**50.0%**	**51.1%**

Table 5.4: Testbench-unaware performance on the commercial subset across CVDP task categories.

Category	# DP	OpenCode	ChipMind	Claude Code
cid003	3	0.0%	33.3%	33.3%
cid005	3	33.3%	33.3%	33.3%
cid012	16	0.0%	18.8%	6.3%
cid013	18	0.0%	11.1%	5.6%
cid014	28	25.0%	28.6%	42.9%
Overall	**68**	**11.8%**	**22.1%**	**23.5%**

5.2.3 Testbench Awareness

The testbench-aware evaluation is available only for the non-commercial subset, so the comparison between the two configurations is restricted to the same 92 datapoints. Table 5.6 reports the performance of ChipMind and Claude Code under both testbench-unaware (TU) and testbench-aware (TA) conditions.

The access to the testbench brings an improvement to both systems, although with substantially different magnitudes. ChipMind increases its pass rate from 50.0% in the TU configuration to 53.3% in TA, corresponding to an improvement of a 3.3%, while Claude Code exhibits a considerably larger increase: from 51.1% to 62.0%, a difference of 10.9%. While the two agents achieve similar performance in the TU setting, a larger gap emerges between them when the testbench is made available.

The effect of testbench access is also strongly category-dependent. For Natural-Language Specification to RTL (cid003), ChipMind improves from 44.1% to 52.9%,

Table 5.5: CVDP task categories considered in the evaluation.

Category	**Task Type**
cid003	Natural-Language Specification to RTL
cid004	RTL Modification
cid005	Specification and Reuse to RTL
cid012	Testbench Stimulus Generation
cid013	Testbench Checker Generation
cid014	Assertion Generation
cid016	RTL Debugging / Bug Fixing

while Claude Code increases from 44.1% to 64.7%. On RTL Modification (cid004), the changes are more limited: ChipMind increases from 56.0% to 60.0%, while Claude Code remains unchanged at 60.0%, this is understandable since in this context the code already contains the information on what should the system do, and thus the context made available from the testbench is less useful than in other categories.

A different behavior is observed for Specification and Reuse to RTL (cid005). Claude Code improves from 50.0% to 59.1%, whereas ChipMind decreases from 50.0% to 45.5%. Similarly, on RTL Debugging and Bug Fixing (cid016), Claude Code increases from 54.5% to 63.6%, while ChipMind remains unchanged at 54.5%. Therefore, although testbench awareness is associated with higher aggregate performance for both systems, its effect is neither uniform across task categories nor consistent across agents. In particular, Claude Code appears to benefit more systematically from the additional testbench information than ChipMind.

Table 5.6: Comparison of testbench-unaware (TU) and testbench-aware (TA) performance on the 92 non-commercial datapoints.

Category	**# DP**	**ChipMind**		**Claude Code**	
		TU	**TA**	**TU**	**TA**
cid003	34	44.1%	52.9%	44.1%	64.7%
cid004	25	56.0%	60.0%	60.0%	60.0%
cid005	22	50.0%	45.5%	50.0%	59.1%
cid016	11	54.5%	54.5%	54.5%	63.6%
Overall	**92**	**50.0%**	**53.3%**	**51.1%**	**62.0%**

The overlap between the datapoints solved by the two agents provides insight into the effect of testbench awareness. In the TU configuration, ChipMind and Claude Code successfully solve 43 of the 92 datapoints in common. Given their respective totals of 46 and 47 successful datapoints, only 3 are solved exclusively by ChipMind and 4 exclusively by Claude Code, while 42 datapoints are solved by neither system.

In the TA configuration, the number of datapoints solved by both agents increases only marginally, from 43 to 44. However, the number of datapoints solved by neither agent decreases substantially, from 42 to 30. At the same time, the number of agent-specific successes increases: 5 datapoints are solved exclusively by ChipMind and 13 exclusively by Claude Code.

Although the number of datapoints successfully solved by both agents is nearly identical across the two configurations, the composition of these common successes differs (see Table 5.7). ChipMind and Claude Code jointly solve 43 of the 92 datapoints (46.7%) in the TU configuration and 44 (47.8%) in the TA configuration.

Only 34 datapoints are jointly solved under both conditions. While a substantial portion of the jointly solved datapoints remains consistent across the two settings, testbench visibility also appears to change the set of tasks on which the agents simultaneously succeed. This difference is not apparent from the aggregate joint-pass rates alone, which remain almost unchanged between TU and TA, but could be explained by the fact that systems are still variable and the model they use are intrinsically nondeterministic, more runs per experiment could be useful to understand the true impact of the testbench on the cluster of problems that the agents are able to solve.

Table 5.7: Comparison of datapoints jointly solved by ChipMind and Claude Code across the testbench-unaware (TU) and testbench-aware (TA) configurations.

Joint-pass set	**# DP**
TU	43 (46.7%)
TA	44 (47.8%)
Both TU and TA	**34**

5.2.4 Impact of Model Scaling

The availability of results for two successive generations of the Claude Sonnet model family enables the analysis of how improvements in the underlying language model affect both direct generation and agentic workflows.

In the non-agentic setting, Claude Sonnet 3.7 achieves a pass@1 of 33.89%, while Claude Sonnet 4.6 reaches 37.30%, corresponding to an improvement of 3.41 percentage points. Since the two models are evaluated under the same non-agentic setting, this increase provides a direct measure of the effect of model scaling on single-inference generation.

A larger improvement is observed within the OpenCode agentic workflow. Both model versions are evaluated on the same 92 non-commercial datapoints, with OpenCode using Claude Sonnet 3.7 achieving a pass rate of 42.39%, compared with 54.3% when using Claude Sonnet 4.6. Replacing the underlying model therefore results in an increase of 11.91 percentage points.

The absolute performance values of the non-agentic and agentic settings are not directly comparable, as the two evaluations are conducted on different sets of CVDP datapoints. The observed improvement is substantially larger for OpenCode (+11.91 percentage points) than for non-agentic generation (+3.41 percentage points). This suggests that, in the evaluated configuration, improvements in the underlying language model have a greater impact when the model operates within an iterative agentic workflow. Stronger model capabilities can influence

multiple stages of the execution process, including reasoning, tool interaction, code modification, and iterative refinement.

Table 5.8: Impact of model scaling from Claude Sonnet 3.7 to Claude Sonnet 4.6.

Setting	Sonnet 3.7	Sonnet 4.6	Improvement
Non-agentic	33.89%	37.30%	+3.41 pp
OpenCode	42.39%	54.3%	+11.91 pp

5.2.5 Time and Cost

In addition to functional correctness, the practical applicability of agentic systems also depends on the execution time and monetary cost required to complete the benchmark. Since ChipMind and Claude Code were both evaluated on the complete set of 160 datapoints, their resource consumption can be directly compared.

As shown in Table 5.9, Claude Code requires more time and higher monetary cost than ChipMind to complete the same evaluation set. This difference is relevant because the functional advantage of Claude Code in the testbench-unaware configuration is small: as reported in Table 5.1, Claude Code achieves an aggregate pass rate of 39.4%, compared with 38.1% for ChipMind. This slightly higher pass rate of Claude Code is accompanied by a substantially higher resource cost.

These results suggest that evaluating agentic workflows only in terms of final pass rate can be misleading. In practical settings, the cost of each successful datapoint also matters. Under the evaluated testbench-unaware configuration, ChipMind provides a more favorable trade-off between functional performance and resource consumption, while Claude Code achieves a small performance improvement at a noticeably higher cost.

Table 5.9: Execution time, monetary cost, and pass rate for the complete testbench-unaware evaluation.

Agent	Time	Avg	Cost	Avg.	Pass rate
ChipMind	22.2 h	8.3 min	$113.43	$0.71	38.1%
Claude Code	30.6 h	11.5 min	$178.43	$1.12	39.4%

5.3 Failure Analysis

The failure analysis was applied to the unsuccessful executions of the three agentic systems in the testbench-unaware non-commercial configuration. 133 failed executions were analyzed, corresponding to 51 distinct CVDP problems (see Table 5.10).

Each failed execution was assigned to one of the nine root-cause categories defined in Table 4.2. The resulting distribution provides a more detailed view of agent performance than aggregate pass rates alone, distinguishing failures attributable to the generated RTL from those originating from the execution environment, verification infrastructure, or agentic workflow. It must be considered though that some non-RTL failures may hied later RTL errors.

Table 5.10: Failed executions included in the failure analysis.

Scope	Count	Description
Total failures	133	Failed agent executions analyzed
Distinct problems	51	Unique CVDP datapoints involved
OpenCode	42	Failed executions
ChipMind	46	Failed executions
Claude Code	45	Failed executions

5.3.1 Overall Failure Patterns

Of the 133 unsuccessful executions, 40 problems (78%) result in a failure for all three evaluated agents. This overlap indicates that failures are largely concentrated on the same benchmark problems rather than being associated with disjoint sets of agent-specific weaknesses.

A similar pattern emerges from the distribution of root causes. Functional failures and benchmark issues constitute the two main categories for all three systems, while the remaining failure modes occur considerably less frequently (see Table 5.11). Despite differences in their overall pass rates, the three agents therefore exhibit broadly similar failure profiles.

To distinguish failures directly attributable to the generated design, from those originating elsewhere in the evaluation pipeline, the root causes were grouped into RTL and non-RTL failures (see Table 5.12). Functional and compilation failures were considered RTL-related, while failures originating from the benchmark, simulator, execution environment, file handling, interface, or agentic loop were classified as non-RTL.

RTL-related failures account for 50% of OpenCode failures, 46% of ChipMind failures, and 38% of Claude Code failures (see Table 5.12). Consequently, approximately 56% of the 133 unsuccessful executions overall are associated with a root cause that cannot be directly attributed to an error in the generated RTL.

This distinction should not, however, be interpreted as implying that the RTL produced in the remaining executions was functionally correct. Several non-RTL failures occur before functional verification can be completed. In these cases, the identified root cause explains why the execution failed, but does not establish whether additional errors would have emerged later in the execution.

Table 5.11: Distribution of root causes among unsuccessful executions. Percentages are computed over the total number of failures for each agent.

Failure category	**OpenCode**	**Claude Code**	**ChipMind**	**Overall**
Functional	50.0%	35.6%	45.7%	43.6%
Benchmark	35.7%	31.1%	30.4%	32.3%
Filename	7.1%	6.7%	6.5%	6.8%
Pre-Simulation / Incomplete	4.8%	6.7%	8.7%	6.8%
Icarus	0.0%	6.7%	8.7%	5.3%
Runtime	0.0%	8.9%	0.0%	3.0%
Compilation	0.0%	2.2%	0.0%	0.8%
Interface	2.4%	2.2%	0.0%	1.5%
Total	**42**	**45**	**46**	**133**

Table 5.12: Distribution of RTL-related and not, failures..

Agent	**RTL-related**	**Total failures**
OpenCode	50.0%	42
Claude Code	37.8%	45
ChipMind	45.7%	46
Overall	**44.4%**	**133**

5.3.2 Failure Mechanisms

Functional failures appear to happen for similar reasons across the three agents (see Table 5.13). Value mismatches represent the main functional failure mode, accounting for 13 cases for both OpenCode and ChipMind and 9 for Claude Code. Timing-related errors, such as outputs being produced one or more cycles earlier

or later than expected, constitute the second recurring pattern, while FSM state errors and propagation of uninitialized values occur less frequently.

The similarity of these distributions across the agents, suggests that the evaluated systems do not exhibit clearly distinct functional failure modes. Instead, independently generated implementations frequently encounter the same classes of RTL correctness problems.

Table 5.13: Distribution of functional failure mechanisms. Percentages are computed over the functional failures of each agent.

Mechanism	OpenCode	Claude	ChipMind	Overall
Value / computation mismatch	13 (61.9%)	9 (56.3%)	13 (61.9%)	35 (60.3%)
Latency off-by-N	4 (19.0%)	3 (18.8%)	3 (14.3%)	10 (17.2%)
FSM state error	3 (14.3%)	2 (12.5%)	3 (14.3%)	8 (13.8%)
X-propagation / uninitialized signal	1 (4.8%)	2 (12.5%)	2 (9.5%)	5 (8.6%)
Total	**21**	**16**	**21**	**58**

5.3.3 Benchmark Issues

A substantial part of the unsuccessful executions was caused by the benchmark and verification environment rather than by an error directly attributable to the generated RTL. Benchmark issues constitute one of the two dominant root-cause categories across all evaluated agents, accounting for 15 failures in OpenCode and 14 failures in both ChipMind and Claude Code (see Table 5.14 for the errors rundown).

The most frequent benchmark-related issue originates from incompatibilities with CoCoTB v2. In 11 failed executions, distributed across six distinct problems, the testbench invokes the `to_unsigned` method on single-bit signals. Under CoCoTB v2, these signals are represented as `Logic` objects rather than `LogicArray` objects, causing the testbench to raise an `AttributeError` before functional verification can be completed. A second CoCoTB-related issue affects the clock generation infrastructure. Five failures across two problems terminate with a `ValueError` caused by clock periods that are not accepted by the clock generator used by the corresponding testbench, since CoCoTBv2 only accepts periods divisible by 2 if ‘period_high‘ is not given

Another recurring source of benchmark failures concerns the configuration of the design and its verification environment. Seven failures across three problems are caused by inconsistencies between the expected module or source file and the

`VERILOG_SOURCES` configuration provided by the dataset. In these cases, the harness attempts to compile a source or instantiate a module that does not correspond to the implementation expected by the task. A related issue was observed in three executions of a single problem, where parameter names assumed during elaboration do not match those of the referenced submodule.

Other failures expose implicit requirements of the verification harness that are not communicated by the task specification. Eight failures across three problems occur because the testbench directly accesses internal design signals using a specific naming convention, which was not explicated in the specification. Similarly, three failures originate from a problem in which the specification does not constrain the width of a signal while the testbench assumes a fixed width, producing a mismatch between otherwise plausible interpretations of the task.

Other three failures are associated with a problem for which contextual information required by the specification is not actually made available to the agent. In this case, the execution cannot reproduce the design assumptions expected by the verification environment because part of the information referenced by the task is absent from the provided context.

A further benchmark issue affects `elevator_control_0004`. In this datapoint, the testbench explicitly sets the simulation outcome to `None`, causing all three evaluated agents to fail independently of the generated RTL. Since the failure is directly encoded in the verification logic, these three executions were classified as benchmark issues.

Overall, these results show that a failed benchmark execution cannot always be interpreted as evidence of an agent error. In several cases, execution terminates before the functional behavior of the generated design can be evaluated. As described in the methodology, these datapoints are nevertheless retained as failures in the reported benchmark results, since removing or correcting them retrospectively would modify the evaluated dataset.

5.4 Discussion and Practical Implications

The results show that LLM-based agentic workflows' behavior is far from uniformly reliable. Performance depends strongly on the interaction between the datapoint, the available testbench feedback, and the execution infrastructure. In particular, the failure analysis suggests that many unsuccessful runs are not caused by a complete inability to generate RTL, but by more specific issues such as interface mismatches, incorrect combinatorial or sequential behavior, latency errors, and toolchain failures.

A first implication is that pass rate alone is not sufficient to evaluate these systems, since two agents may obtain similar final pass rates, while failing for

Table 5.14: Recurring benchmark issues identified during the failure analysis.

Benchmark issue	**# Failures**	**# Problems**
CoCoTB v2 `to_unsigned` incompatibility on 1-bit signals	11	6
Undocumented internal signal naming required by the testbench	8	3
`VERILOG_SOURCES` / module configuration mismatch	7	3
CoCoTB v2 clock generation incompatibility	5	2
Parameter naming mismatch during elaboration	3	1
Required context not provided to the agent	3	1
Signal width unspecified by the task but fixed by the testbench	3	1
Hard-coded `Simulation: None` in the testbench	3	1

different reasons.

The analysis by difficulty confirms that harder datapoints lead to higher failure rates across all agents. This trend is expected, but the per-agent breakdown is more informative than the aggregate result: it shows whether the increase in difficulty affects all agents similarly or whether some agents degrade more sharply. This matters in practice because an agent that is competitive on easy tasks may not be suitable for more complex datapoints without stronger feedback, better decomposition, or additional verification support.

The functional failure mechanisms highlights a practical limitation of current workflows. The dominant category is value or computation mismatch, which means that the agents often generate designs that compile and run but implement the wrong behavior.

These results suggest that agentic LLMs as of now are more useful as assisted design and debugging tools than as fully autonomous RTL generation systems. They can accelerate exploration and produce useful candidate implementations, but the generated designs still require systematic verification. The most promising use cases are therefore workflows where the agent is embedded in a tight feedback loop with tests, simulators, and clear failure diagnostics.

Overall, the main practical implication is that reliable agentic hardware generation requires more than a strong language model. It requires robust harnesses, clear diagnostics, and evaluation metrics that distinguish between process failures and design failures. The agent should be evaluated not only on whether it eventually passes, but also on how it uses feedback, how often it reaches executable RTL, how expensive the run is, and which classes of failures remain unresolved.

Chapter 6

Conclusions

This thesis evaluated LLM-based agentic workflows on the CVDP benchmark with the goal of understanding not only whether agents can solve hardware design datapoints, but also how and why they fail. The work combined pass-rate analysis with a failure analysis that distinguished agentic-loop limitations from datapoint-level failures and infrastructure-related issues. This distinction is essential in hardware-generation workflows, because a failed run may be caused by incorrect RTL, by an incomplete or unstable agent loop, or by problems in the surrounding toolchain.

Across the evaluated agentic configurations, the results showed low and uneven pass rates. Performance varied across agents, benchmark categories, and difficulty levels. Testbench-aware settings generally improved the results, but the improvement was not uniform across agents: Claude Code benefited more clearly from testbench exposure than ChipMind. This suggests that different workflows may differ not only in model capability, but also in how effectively they inspect the environment, use verification feedback, and adapt their generated RTL after observing test failures. The analysis by difficulty showed that failure rates increase on harder datapoints, confirming that the agents struggle more as the required functional and temporal reasoning becomes more complex.

The results show that current agentic workflows still have limited reliability on CVDP. Although CVDP was designed to be more challenging than overly simple HDL-generation benchmarks, its datapoints are still far from real industrial use cases, where agents would need to navigate huge repositories, existing module hierarchies, build systems, project-specific constraints, and complex verification environments. Therefore, the low pass rates observed in this work are significant: the agents already struggle in a controlled benchmark setting, before being exposed to the full complexity of real hardware-development flows.

The failure analysis also shows that pass/fail results alone are not sufficient to characterize agentic performance. Many unsuccessful runs are not simple syntax

failures, but functional failures in designs that are plausible enough to compile or execute. Common mechanisms include value or computation mismatches, latency errors, FSM-state mistakes, and uninitialized-signal behaviour. These failures highlight that the main challenge is not only generating syntactically valid RTL, but correctly capturing the intended temporal and functional semantics of the design.

The comparison across agents further indicates that different workflows fail in different ways. In particular, the effect of testbench exposure is not uniform across agents, suggesting that some systems may be better at exploiting concrete verification feedback than others, or may be more prone to exploring the environment even when not explicitly directed to do so.

Understanding the differences between agents requires looking beyond final pass rates and analyzing the internal behavior of each agent: which tools are used, how often tests are run, how logs are interpreted, how many files are inspected or modified, and whether the agent converges toward a fix or repeatedly makes speculative changes.

Overall, the main conclusion of this work is that current LLM-based agentic workflows are not yet reliable autonomous RTL-generation systems. Their current value lies more in assisted exploration, debugging, and candidate solution generation than in unsupervised hardware design. Future work should therefore focus first on correcting and isolating benchmark issues, extending the analysis of commercial datapoints, and studying in more detail how testbench exposure affects different agents. A more complete evaluation should also include richer measurements of agent behavior, such as tool usage, number of test executions, file inspection patterns, debugging iterations, cost, and recovery behavior after failures.

Appendix A

Functional description of a CVDP datapoint

```
id: cvdp_agentic_arithmetic_progression_generator_0001
categories:
  - cid016
  - easy
system_message: |-
  You are a language model that has the following file operations available at your disposal:
    - **List files in a directory** by running one of the following commands:
      - `ls`
      - `tree`
    - **Read files** by using:
      - `cat <filename>`
    - **Write files** by using:
      - `echo <content> > <filename>`
    - **Compile Verilog** by using `iverilog` such as:
      - `iverilog -o <output_filename>.out -g2012 <verilog_code_file> <verilog_testbench_file>`
    - **Run Simulation** by using:
      - `vvp <output_filename>.out`
    - **Find current working directory** by using:
      - `pwd`
    - **Update the contents of a text file from a old content to new content**
      - `sed -i "problematic_line_number s/problematic_statement/non_problematic_statement/" Buggy_RTL_code.sv`
    - **To access a specific line of the file**
       - `awk 'NR==line_number' file_name.sv`

  You will be given a prompt and your task is to understand it and solve the given issue by using the above-mentioned commands as needed. In the final step, you should create a Linux patch highlighting the necessary file updates to achieve the targeted goal.

    You will solve the problem step by step using the following approach of
    - thought (thinking process of the step you're going to take
    - action (the command you will be running to get more details/context that's helpful to solve the problem)
```

```
    - observation (the output from the action you will observe based on which you
    will take your next step)

    The last step will be the final output summary and the patch itself in the
    following format
    - thought (the summary of what you did and some introduction of the patch file
     itself)
    - patch (a Linux-based patch that needs to be applied to reach the relevant
    solution)

    The patch file should only be applied to a single file to reach the required
    solution.
prompt: |
  ### **Bug Fix Request in Arithmetic progression generator RTL**

  I have the Arithmetic progression generator RTL in the current directory, and I
    need assistance in fixing the following bugs:

  #### **Identified Bugs:**
  1. **Overflow Handling:** Overflow occurs in `out_val` and `counter` when the
    input values reach their maximum limits.
  2. **Missing Condition for Sequence Length Zero:** The design lacks a check for
    a sequence length of `0`, leading to incorrect behavior when `0` is applied,
    as operations continue instead of being skipped.

  Could you help resolve these bugs in the RTL?
context:
  rtl/arithmetic_progression_generator.sv: |-
    module arithmetic_progression_generator #(
        parameter DATA_WIDTH = 16,  // Width of the input data
        parameter SEQUENCE_LENGTH = 10 // Number of terms in the progression
    )(
        clk,
        resetn,
        enable,
        start_val,
        step_size,
        out_val,
        done
    );
      // ----------------------------------------
      // - Local parameter definition
      // ----------------------------------------

        localparam WIDTH_OUT_VAL = $clog2(SEQUENCE_LENGTH) + DATA_WIDTH; // Bit
    width of out_val to prevent overflow

      // ----------------------------------------
      // - Interface Definitions
      // ----------------------------------------
        input logic clk;                          // Clock signal
        input logic resetn;                       // Active-low reset
        input logic enable;                       // Enable signal for the
    generator
        input logic [DATA_WIDTH-1:0] start_val;   // Start value of the sequence
        input logic [DATA_WIDTH-1:0] step_size;   // Step size of the sequence
        output logic [WIDTH_OUT_VAL-1:0] out_val; // Current value of the sequence
        output logic done;                        // High when sequence generation
     is complete


```

```
      // ----------------------------------------
      // - Internal signals
      // ----------------------------------------
        logic [WIDTH_OUT_VAL-1:0] current_val;  // Register to hold the current value
        logic [$clog2(SEQUENCE_LENGTH)-1:0] counter;  // Counter to track sequence length

      // ----------------------------------------
      // - Procedural block
      // ----------------------------------------
        always_ff @(posedge clk or negedge resetn) begin
            if (!resetn) begin
                current_val <= 0;
                counter <= 0;
                done <= 1'b0;
            end else if (enable) begin
                if (!done) begin
                    if (counter == 0) begin
                        current_val <= start_val; // Initialize with start value
                    end else begin
                        current_val <= current_val + step_size; // Compute next term
                    end

                    if (counter < SEQUENCE_LENGTH - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        done <= 1'b1; // Mark completion
                    end
                end
            end
        end

      // ----------------------------------------
      // - Combinational Assignments
      // ----------------------------------------
        assign out_val = current_val;

    endmodule
patch:
  rtl/arithmetic_progression_generator.sv: ''
harness:
  docker-compose.yml: |+
    services:

      01-new-tb:
        image: __OSS_SIM_IMAGE__
        volumes:
          - ./src/:/src/:ro
        working_dir : /code/rundir
        env_file    : ./src/.env
        command     : pytest -s -o cache_dir=/rundir/harness/.cache /src/test_runner.py -v

  src/.env: |-
    SIM             = icarus
    WAVE            = False
    TOPLEVEL_LANG   = verilog
    VERILOG_SOURCES = /code/rtl/arithmetic_progression_generator.sv
    TOPLEVEL        = arithmetic_progression_generator
```

```
    MODULE          = test_arithmetic_progression_generator
    PYTHONPATH      = /src
    HASH            = 89e25e0bcc604d170ede28f3649c3623cb94c207
  src/harness_library.py: |2+

    from cocotb.triggers import FallingEdge, RisingEdge, Timer
    import random

    async def reset_dut(reset, duration_ns = 10):
        # Restart Interface
        reset.value = 1
        await Timer(duration_ns, units="ns")
        reset.value = 0
        await Timer(duration_ns, units="ns")
        reset.value = 1
        await Timer(duration_ns, units='ns')
        reset._log.debug("Reset complete")


    async def dut_init(dut):
        # iterate all the input signals and initialize with 0
        for signal in dut:
            if signal._type == "GPI_NET":
                signal.value = 0

  src/test_arithmetic_progression_generator.py: |-
    # File: arithmetic_progression_generator.py

    import cocotb
    from cocotb.clock import Clock
    from cocotb.triggers import FallingEdge, RisingEdge, ClockCycles, Timer
    import harness_library as hrs_lb
    import random
    import time
    import math

    def clog2(N):
        return math.ceil(math.log2(N))

    @cocotb.test()
    async def test_arithmetic_progression_generator(dut):

        # Randomly execute this statement in one of the iterations
        MIN_CLOCK_PERIOD = 4
        # clock_period_ns = random.randint(MIN_CLOCK_PERIOD, 15)  # For example,
    10ns clock period
        clock_period_ns = 10  # For example, 10ns clock period
        cocotb.start_soon(Clock(dut.clk, clock_period_ns, units='ns').start())
        print("[INFO] Clock started.")

        # Initialize DUT
        await hrs_lb.dut_init(dut)

        # Apply reset
        await hrs_lb.reset_dut(dut.resetn, clock_period_ns)
        await RisingEdge(dut.clk)
        await RisingEdge(dut.clk)

        # Extract parameters from the DUT
        DATA_WIDTH = int(dut.DATA_WIDTH.value)
        SEQUENCE_LENGTH = int(dut.SEQUENCE_LENGTH.value)
```

```
196         if SEQUENCE_LENGTH == 0:
197             SEQUENCE_LENGTH_MOD = 1
198             EXPECTED_WIDTH_OUT_VAL = 1 + (DATA_WIDTH)
199         else :
200             SEQUENCE_LENGTH_MOD = SEQUENCE_LENGTH
201             EXPECTED_WIDTH_OUT_VAL = clog2(SEQUENCE_LENGTH_MOD) + (DATA_WIDTH)
202         WIDTH_OUT_VAL = int(dut.WIDTH_OUT_VAL.value)
203 
204         print(f"DATA_WIDTH= {DATA_WIDTH}, SEQUENCE_LENGTH= {SEQUENCE_LENGTH_MOD},
    WIDTH_OUT_VAL={WIDTH_OUT_VAL} ")
205 
206         reset_system = 0
207         # if random.choice([True, False]):
208         #     reset_system = 1
209         #     N_cycles_reset = random.randint(2, 5)
210         #     Positive_delta = random.randint(3, 10)
211         #     # Generate start_cycle mostly greater than SEQUENCE_LENGTH
212         #     if random.random() < 0.8:  # 80% chance to be greater
213         #         start_cycle_reset = random.randint(SEQUENCE_LENGTH_MOD + 2 ,
    SEQUENCE_LENGTH_MOD + Positive_delta)
214         #     else:  # 20% chance to be less
215         #         start_cycle_reset = random.randint(1, SEQUENCE_LENGTH_MOD - 1)
216         #     print(f"Reset will be given at {start_cycle_reset + 1} cycle for {
    N_cycles_reset} cycles!")
217 
218         Dessert_enable = 0
219         # if random.choice([True, False]):
220         #     Dessert_enable = 1
221         #     N_cycles = random.randint(1, 3)
222         #     start_cycle = random.randint(1, SEQUENCE_LENGTH_MOD-1)
223         #     print(f"Enable will be deasserted at {start_cycle + 1} cycle for {
    N_cycles} cycles !")
224 
225         # Test-specific variables
226         MAX_VALUE =  (1 << DATA_WIDTH) - 1
227         start_val = random.randint(1, MAX_VALUE)  # Example start value
228         step_size = random.randint(1, MAX_VALUE)  # Example step size
229         if random.choice([True, False]):
230             start_val = MAX_VALUE  # Example start value
231             step_size = MAX_VALUE  # Example start value
232             print(f"Overflow check !")
233             print(f"WIDTH_OUT_VAL = {WIDTH_OUT_VAL}, EXPECTED_WIDTH_OUT_VAL = {
    EXPECTED_WIDTH_OUT_VAL}")
234 
235         cycle_num = random.randint( SEQUENCE_LENGTH_MOD + 2, 100)
236         cycle = 0
237         expected_value = 0
238         expected_value_s1 = 0
239         expected_done = 0
240         expected_done_s1 = 0
241         counter = 0
242         reset = 0
243 
244         for cycle in range(cycle_num):  # Run the test for random number of cycles
245             ##################################################################
246             ######### Applying reset to the system randomly
247             ##################################################################
248             dut.resetn.value = 1
249             if reset_system == 1 :
250                 #reset applied for N cycles after start_cycle
251                 reset = 0
```

```
252                 if cycle >= start_cycle_reset and cycle < start_cycle_reset +
    N_cycles_reset  :
253                     reset = 1
254                     dut.resetn.value = 0
255                     print(f"Reset applied for {N_cycles_reset} cycles!")
256                     expected_value = 0
257                     expected_value_s1 = 0
258                     expected_value_s2 = 0
259                     expected_done = 0
260                     expected_done_s1 = 0
261                     expected_done_s2 = 0
262                     counter = 0
263             ##################################################################
264             ######### Controlling enable signal randomly
265             ##################################################################
266             enable = 1
267             if Dessert_enable == 1 :
268                 #valid in 0 for N cycles after start_cycle
269                 if cycle >= start_cycle and cycle < start_cycle + N_cycles  :
270                    enable = 0
271                    print(f"Enable deasserted for {N_cycles} cycles!")
272             dut.enable.value = enable
273             dut.start_val.value = start_val
274             dut.step_size.value = step_size
275
276             ##################################################################
277             ######### Verification function
278             ##################################################################
279             if enable == 1 and not reset and SEQUENCE_LENGTH > 0:
280                 if counter < SEQUENCE_LENGTH :
281                     if counter == 0 :
282                         expected_value = start_val
283                         expected_done = 0
284                         counter = counter + 1
285                     else :
286                         expected_value += step_size
287                         expected_done = 0
288                         counter = counter + 1
289                 else :
290                     expected_done = 1
291             else :
292                 expected_value = expected_value
293                 expected_done = expected_done
294                 counter = counter
295
296             ##################################################################
297             ######### Clock rise edge
298             ##################################################################
299             expected_value_s2 = expected_value_s1
300             expected_done_s2 = expected_done_s1
301             await RisingEdge(dut.clk)
302             expected_value_s1 = expected_value
303             expected_done_s1 = expected_done
304
305             ##################################################################
306             ######### Actual RTL module
307             ##################################################################
308             actual_value =dut.out_val.value.to_unsigned()
309             actual_done = dut.done.value.to_unsigned()
310
311             ##################################################################
```

```
312             ######### Assertions
313             ##################################################################
314             ##Assertion to check data out, assertion to check overflow
315             assert actual_value == expected_value_s2, f"Error at step {i}:
    expected {expected_value_s2}, got {int(dut.out_val.value)}"
316             ##Assertion to check done
317             assert actual_done == expected_done, "Done signal not asserted after
    sequence completion"
318             ##Assertion to check val_out width
319             assert WIDTH_OUT_VAL == EXPECTED_WIDTH_OUT_VAL, "Wrong calculation of
    WIDTH_OUT_VAL"
320             ##Assertion to check reset
321             if reset == 1 :
322                 assert actual_value == expected_value_s2 == 0 , f"Error at step {i
    }:At reset, expected {expected_value_s2}, got {actual_value}"
323                 assert actual_done == expected_done == 0 , f"Error at step {i}:At
    reset, expected_done {expected_done}, got {actual_done}"
324
325             print(f"[DEBUG] Cycle {cycle+1}/{cycle_num}: start_val = {hex(
    start_val)}")
326             print(f"[DEBUG] Cycle {cycle+1}/{cycle_num}: step_size = {step_size}")
327             print(f"[DEBUG] Cycle {cycle+1}/{cycle_num}: enable = {enable}")
328             print(f"[DEBUG] Cycle {cycle+1}/{cycle_num}: expected_value = {hex(
    expected_value_s2)}, expected_done = {expected_done}")
329             print(f"[DEBUG] Cycle {cycle+1}/{cycle_num}: actual_value   = {hex(
    actual_value)}, actual_done   = {actual_done}")
330             print(f"\n")
331
332   src/test_runner.py: |
333     import os
334     from cocotb_tools.runner import get_runner
335     import pytest
336     import random
337
338     # Fetch environment variables for simulation setup
339     verilog_sources = os.getenv("VERILOG_SOURCES").split()
340     toplevel_lang   = os.getenv("TOPLEVEL_LANG")
341     sim             = os.getenv("SIM", "icarus")
342     toplevel        = os.getenv("TOPLEVEL")
343     module          = os.getenv("MODULE")
344     wave            = bool(os.getenv("WAVE"))
345
346     # Runner function
347     def runner(DATA_WIDTH: int=0, SEQUENCE_LENGTH: int=0, start_val: int=0,
    step_size: int=0, enable: int=0):
348         # Plusargs to pass simulation parameters enable
349         plusargs = [
350             f'+start_val={start_val}',
351             f'+step_size={step_size}',
352             f'+enable={enable}'
353         ]
354
355         parameters = {
356             "DATA_WIDTH": DATA_WIDTH,
357             "SEQUENCE_LENGTH": SEQUENCE_LENGTH
358         }
359
360         # Debug information
361         print(f"[DEBUG] Running simulation with DATA_WIDTH={DATA_WIDTH},
    SEQUENCE_LENGTH={SEQUENCE_LENGTH}")
362         print(f"[DEBUG] Start Value: {start_val}, Step Size: {step_size}")
```

```
        print(f"[DEBUG] Parameters: {parameters}")

        # Configure the simulation runner
        runner = get_runner(sim)
        runner.build(
            sources=verilog_sources,
            hdl_toplevel=toplevel,
            parameters=parameters,
            always=True,
            clean=True,
            waves=True,
            verbose=True,
            timescale=("1ns", "1ns"),
            log_file="sim.log"
        )

        runner.test(hdl_toplevel=toplevel, test_module=module, waves=wave,
    plusargs=plusargs)

    # Pytest parameterization
    @pytest.mark.parametrize("DATA_WIDTH", [random.randint(4, 32)])
    @pytest.mark.parametrize("SEQUENCE_LENGTH", [0])
    @pytest.mark.parametrize("test", range(5))  # Run 50 tests
    def test_arithmetic_progression_generator(DATA_WIDTH, SEQUENCE_LENGTH,test):
        runner(DATA_WIDTH=DATA_WIDTH, SEQUENCE_LENGTH=SEQUENCE_LENGTH)
_jsonl_line: 13
```

Bibliography

[1] Guyue Huang et al. *Machine Learning for Electronic Design Automation: A Survey*. 2021. arXiv: 2102.03357 [eess.SP]. URL: https://arxiv.org/abs/2102.03357 (cit. on pp. 1, 4, 5).

[2] Jingyu Pan, Guanglei Zhou, Chen-Chia Chang, Isaac Jacobson, Jiang Hu, and Yiran Chen. *A Survey of Research in Large Language Models for Electronic Design Automation*. 2025. arXiv: 2501.09655 [cs.LG]. URL: https://arxiv.org/abs/2501.09655 (cit. on pp. 1, 6).

[3] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165 (cit. on pp. 1, 6).

[4] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374 (cit. on p. 1).

[5] 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*. 2023. arXiv: 2308.00708 [cs.PL]. URL: https://arxiv.org/abs/2308.00708 (cit. on pp. 1, 37).

[6] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. *AutoChip: Automating HDL Generation Using LLM Feedback*. 2024. arXiv: 2311.04887 [cs.PL]. URL: https://arxiv.org/abs/2311.04887 (cit. on p. 1).

[7] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. *RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models*. 2024. arXiv: 2311.16543 [cs.AR]. URL: https://arxiv.org/abs/2311.16543 (cit. on p. 1).

[8] Zhiteng Chao et al. *Think with Self-Decoupling and Self-Verification: Automated RTL Design with Backtrack-ToT*. 2025. arXiv: 2511.13139 [cs.AR]. URL: https://arxiv.org/abs/2511.13139 (cit. on pp. 1, 4).

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629 (cit. on pp. 1, 11, 39).

[10] Lei Wang et al. «A survey on large language model based autonomous agents». In: *Frontiers of Computer Science* 18.6 (Mar. 2024). ISSN: 2095-2236. DOI: 10.1007/s11704-024-40231-1. URL: http://dx.doi.org/10.1007/s11704-024-40231-1 (cit. on pp. 1, 11).

[11] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770 (cit. on pp. 2, 32).

[12] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. *VerilogEval: Evaluating Large Language Models for Verilog Code Generation*. 2023. arXiv: 2309.07544 [cs.LG]. URL: https://arxiv.org/abs/2309.07544 (cit. on pp. 2, 8, 26, 27).

[13] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. *RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model*. 2023. arXiv: 2308.05345 [cs.LG]. URL: https://arxiv.org/abs/2308.05345 (cit. on pp. 2, 23, 25, 29).

[14] Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da 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*. 2025. arXiv: 2506.14074 [cs.LG]. URL: https://arxiv.org/abs/2506.14074 (cit. on pp. 2, 32, 35).

[15] M. Morris Mano and Michael D. Ciletti. *Digital Design: With an Introduction to the Verilog HDL*. 5th. Pearson, 2013. ISBN: 978-0-13-277420-8 (cit. on p. 4).

[16] Luciano Lavagno, Igor L. Markov, Grant Martin, and Louis K. Scheffer, eds. *Electronic Design Automation for Integrated Circuits Handbook*. 2nd. CRC Press, 2016. ISBN: 978-1-4822-5450-1 (cit. on p. 4).

[17] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. IEEE, 2024. DOI: 10.1109/IEEESTD.2024.10458102 (cit. on p. 4).

[18] Ahmed Allam and Mohamed Shalan. *RTL-Repo: A Benchmark for Evaluating LLMs on Large-Scale RTL Design Projects*. 2024. arXiv: 2405.17378 [cs.LG]. URL: https://arxiv.org/abs/2405.17378 (cit. on p. 6).

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 6, 9).

[20] Wayne Xin Zhao et al. «A Survey of Large Language Models». In: *Frontiers of Computer Science* 20.12 (May 2026). ISSN: 2095-2236. DOI: 10.1007/s11704-026-60308-3. URL: http://dx.doi.org/10.1007/s11704-026-60308-3 (cit. on pp. 6, 8).

[21] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652 (cit. on p. 7).

[22] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. *Don't Stop Pretraining: Adapt Language Models to Domains and Tasks*. 2020. arXiv: 2004.10964 [cs.CL]. URL: https://arxiv.org/abs/2004.10964 (cit. on p. 8).

[23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. *Toolformer: Language Models Can Teach Themselves to Use Tools*. 2023. arXiv: 2302.04761 [cs.CL]. URL: https://arxiv.org/abs/2302.04761 (cit. on pp. 8, 12).

[24] Jeffrey L. Elman. «Finding Structure in Time». In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1 (cit. on p. 9).

[25] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks*. 2019. arXiv: 1909.09586 [cs.NE]. URL: https://arxiv.org/abs/1909.09586 (cit. on p. 9).

[26] anomalyco. *OpenCode Repo*. https://github.com/lisagiacob/opencode/commit/dce7eceb2855bc36a41bc49d9c56d5dcc92a8eb2. 2026 (cit. on p. 12).

[27] Anthropic. *Claude Code Documentation*. URL: https://code.claude.com/docs/en/how-claude-code-works (visited on 07/14/2026) (cit. on p. 16).

[28] Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. *Dive into Claude Code: The Design Space of Today's and Future AI Agent Systems*. 2026. arXiv: 2604.14228 [cs.SE]. URL: https://arxiv.org/abs/2604.14228 (cit. on pp. 16–18).

[29] ultraworkers. *Claw Code Repo*. https://github.com/ultraworkers/claw-code.git. 2026 (cit. on p. 17).

[30] ultraworkers. *Claow Code Repo Commit Specific*. https://github.com/ultraworkers/claw-code/tree/dc47482e4086325d1dc05d6a6351b63be7f8624d. Commit dc47482, accessed: 2026-04-30. 2026 (cit. on p. 18).

[31] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. *Executable Code Actions Elicit Better LLM Agents*. 2024. arXiv: 2402.01030 [cs.CL]. URL: https://arxiv.org/abs/2402.01030 (cit. on pp. 19, 20).

[32] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. *Benchmarking Large Language Models for Automated Verilog RTL Code Generation*. 2022. arXiv: 2212.11140 [cs.PL]. URL: https://arxiv.org/abs/2212.11140 (cit. on p. 23).

[33] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. «Chip-Chat: Challenges and Opportunities in Conversational Hardware Design». In: *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6. DOI: 10.1109/mlcad58807.2023.10299874. URL: http://dx.doi.org/10.1109/MLCAD58807.2023.10299874 (cit. on p. 23).

[34] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374 (cit. on p. 26).

[35] https://hdlbits.01xz.net/wiki/Main_Page (cit. on p. 27).

[36] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. *Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation*. 2025. arXiv: 2408.11053 [cs.AR]. URL: https://arxiv.org/abs/2408.11053 (cit. on pp. 29, 31).

[37] 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)*. IEEE, 2024, pp. 1–5. DOI: 10.1109/lad62341.2024.10691788. URL: http://dx.doi.org/10.1109/LAD62341.2024.10691788 (cit. on p. 37).

[38] Kaiyan Chang et al. «Data is all you need: Finetuning LLMs for Chip Design via an Automated design-data augmentation framework». In: *Proceedings of the 61st ACM/IEEE Design Automation Conference*. DAC '24. ACM, 2024, pp. 1–6. DOI: 10.1145/3649329.3657356. URL: http://dx.doi.org/10.1145/3649329.3657356 (cit. on p. 37).

[39] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. *BetterV: Controlled Verilog Generation with Discriminative Guidance*. 2024. arXiv: 2402.03375 [cs.AI]. URL: https://arxiv.org/abs/2402.03375 (cit. on p. 38).

[40] Yang Zhang et al. *QiMeng-SALV: Signal-Aware Learning for Verilog Code Generation*. 2025. arXiv: 2510.19296 [cs.LG]. URL: https://arxiv.org/abs/2510.19296 (cit. on p. 38).

[41] Yingqiang Ge, Wenyue Hua, Kai Mei, Jianchao Ji, Juntao Tan, Shuyuan Xu, Zelong Li, and Yongfeng Zhang. *OpenAGI: When LLM Meets Domain Experts*. 2023. arXiv: 2304.04370 [cs.AI]. URL: https://arxiv.org/abs/2304.04370 (cit. on p. 41).

[42] Jiale Liu, Taiyu Zhou, and Tianqi Jiang. *Veri-Sure: A Contract-Aware Multi-Agent Framework with Temporal Tracing and Formal Verification for Correct RTL Code Generation*. 2026. arXiv: 2601.19747 [cs.AR]. URL: https://arxiv.org/abs/2601.19747 (cit. on pp. 42, 43).

[43] Wei-Po Hsin, Ren-Hao Deng, Yao-Ting Hsieh, En-Ming Huang, and Shih-Hao Hung. *EvolVE: Evolutionary Search for LLM-based Verilog Generation and Optimization*. 2026. arXiv: 2601.18067 [cs.AI]. URL: https://arxiv.org/abs/2601.18067 (cit. on pp. 45, 46).

[44] Qizheng Zhang et al. *Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models*. 2026. arXiv: 2510.04618 [cs.LG]. URL: https://arxiv.org/abs/2510.04618 (cit. on p. 48).

[45] Chenhui Deng, Zhongzhi Yu, Guan-Ting Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. *ACE-RTL: When Agentic Context Evolution Meets RTL-Specialized LLMs*. 2026. arXiv: 2602.10218 [cs.AR]. URL: https://arxiv.org/abs/2602.10218 (cit. on pp. 50, 51, 53).

Acknowledgements

As I am closing this chapter of my life, I am realizing that I will soon need to start putting on my big-girl pants. But before doing that, I first want to acknowledge the people who, in one way or another, have helped shape me into the person I am today.

First and foremost, I need to thank my parents, whose love and support have always been unconditional. They have, of course, shaped the person I am today more than anyone else. They taught me to think for myself, to be independent, and to seize every opportunity that came my way. Most importantly, they have always done everything in their power to help me succeed. I also wish to mention my grandpa Piero, whom I loved dearly, and who would have been immensely happy to see me achieve this milestone.

I want to acknowledge my friends: the ones I still have by my side and the ones I lost along the way, the ones who have known me for what feels like a lifetime and the ones I have only recently met. There is something of each of them in me, and for that I am grateful.

I would like to thank my supervisors and the PhD students who supported me throughout this thesis. I feel lucky to have found such patient and helpful people to work with, and I am grateful for all the support they have given me over these past months.

Finally, I want to thank the people who encouraged me throughout my whole academic journey, particularly Professor Brunelli and Professor Cibinetto.

Professor Brunelli met me back when I was still in high school. Since then, he has always cheered me on, encouraged me, and helped me whenever he could. Thank you for having accompanied me all the way here.

Professor Cibinetto supervised my Bachelor's thesis and entrusted me with a project that, at the time, felt bigger than I was. While I constantly doubted whether I was capable of doing it, I never saw him waver in his belief that I could. And, in the end, I did. It was thanks to that project that I chose this Master's degree and this specialization.

Last but certainly not least, I need to thank my dog, Caffè degli Atenei, the Chinese lady who makes my favorite gyoza, and Andrea.

And now, with a little bit of all of you coming with me, I guess it is finally time to put on those big-girl pants and take on what comes next.