Review

Not peer-reviewed version

Large Language Model for Verilog Code Generation: Literature Review and the Road Ahead

Guang Yang *, Wei Zheng *, Xiang Chen, Dong Liang, Peng Hu, Yukui Yang, Shaohua Peng, Zhenghan Li, Jiahui Feng, Xiao Wei, Kexin Sun, Deyuan Ma, Haotian Cheng, Yiheng Shen, Xing Hu, Terry Yue Zhuo, David Lo

Posted Date: 10 November 2025

doi: 10.20944/preprints202511.0656.v1

Keywords:
literature review; Verilog code generation; large language models

Review

Large Language Model for Verilog Code Generation: Literature Review and the Road Ahead

Guang Yang [1,2,*], Wei Zheng [1,*], Xiang Chen [3], Dong Liang [1], Peng Hu [1], Yukui Yang [1], Shaohang Peng [1], Zhenghan Li [1], Jiahui Feng [1], Xiao Wei [1], Kexin Sun [1], Deyuan Ma [1], Haotian Cheng [1], Yiheng Shen [1,2], Xing Hu [4], Terry Yue Zhuo [5] and David Lo [6]

[1] School of Software, Northwestern Polytechnical University, Xi'an 710072, China
[2] Institute of Artificial Intelligence and Educational Big Data, Nantong Normal College, Jiangsu 226000, China
[3] School of Artificial Intelligence and Computer Science, Nantong University, Jiangsu 226000, China
[4] School of Software Technology, Zhejiang University, Zhejiang 310027, China
[5] Department of Software Systems & Cybersecurity, Monash University, Victoria 3800, Australia
[6] School of Computing and Information Systems, Singapore Management University, Singapore 188065, Singapore
* Correspondence: novelyg@outlook.com (G.W.); wzheng@nwpu.edu.cn (W.Z.)

Abstract

Code generation has emerged as a critical research area at the intersection of Software Engineering (SE) and Artificial Intelligence (AI), attracting significant attention from both academia and industry. Within this broader landscape, Verilog, as a representative hardware description language (HDL), plays a fundamental role in digital circuit design and verification, making its automated generation particularly significant for Electronic Design Automation (EDA). Consequently, recent research has increasingly focused on applying Large Language Models (LLMs) to Verilog code generation, particularly at the Register Transfer Level (RTL), exploring how these AI-driven techniques can be effectively integrated into hardware design workflows. Despite substantial research efforts have explored LLM applications in this domain, a comprehensive survey synthesizing these developments remains absent from the literature. This review fill addresses this gap by providing a systematic literature review of LLM-based methods for Verilog code generation, examining their effectiveness, limitations, and potential for advancing automated hardware design. The review encompasses research work from conferences and journals in the fields of SE, AI, and EDA, encompassing 70 papers published on venues, along with 32 high-quality preprint papers, bringing the total to 102 papers. By answering four key research questions, we aim to (1) identify the LLMs used for Verilog generation, (2) examine the datasets and metrics employed in evaluation, (3) categorize the techniques proposed for Verilog generation, and (4) analyze LLM alignment approaches for Verilog generation. Based on our findings, we have identified a series of limitations of existing studies. Finally, we have outlined a roadmap highlighting potential opportunities for future research endeavors in LLM-assisted hardware design.

Keywords: literature review; Verilog code generation; large language models

1. Introduction

Code generation has emerged as one of the most transformative applications of Large Language Models (LLMs), fundamentally reshaping software development practices across diverse programming paradigms. Following the remarkable success of models like Codex [1] and CodeLlama [2], LLM-based code generation has demonstrated unprecedented capabilities in translating natural language specifications into executable code for general-purpose programming languages such as Python, Java, and C++ [3]. This success has catalyzed extensive research exploring LLM applications beyond traditional software engineering, extending into specialized domains including scientific computing [4], web development [5], and domain-specific languages [6,7].

 doi:10.20944/preprints202511.0656.v1

However, the realm of hardware design presents fundamentally distinct challenges that differentiate it from conventional software development. The Electronic Design Automation (EDA) industry faces escalating complexity in modern integrated circuit design, where the productivity gap between hardware capabilities and design resources continues to widen [8]. Hardware Description Languages (HDLs) such as Verilog demand specialized domain expertise encompassing intricate knowledge of digital circuit design, timing constraints, and hardware architecture principles. Traditional hardware design workflows, characterized by manual coding, extensive verification cycles, and iterative refinement processes, frequently become bottlenecks in meeting the demands of rapidly evolving technological requirements. Furthermore, the hardware design industry confronts severe talent shortages, with the steep learning curve of HDLs creating barriers for entry-level engineers and limiting productivity even among experienced designers. Specifically, Verilog code generation represents a particularly compelling research direction due to several fundamental distinctions from software code generation [150]:

- **Concurrency:** hardware systems are inherently parallel, requiring simultaneous consideration of multiple signal paths unlike sequential software execution;
- **Timing constraints:** designs must satisfy strict timing requirements including clock domains, setup/hold times, and propagation delays;
- **Physical limitations:** generated code must respect area, power, and routing constraints that directly impact manufacturing feasibility;
- **Synthesizability:** verification encompasses functional correctness, timing closure, and fabrication compliance beyond traditional software testing;
- **Domain expertise:** effective generation demands deep understanding of digital logic, computer architecture, and semiconductor physics.

These unique characteristics underscore both the potential impact and inherent difficulty of applying LLMs to Verilog code generation.

Research interest in LLM-based Verilog code generation has experienced remarkable growth since its inception. Our comprehensive literature analysis reveals an explosive trajectory: beginning with a single pioneering study in 2020 [9], the field remained relatively dormant through 2022, but subsequently witnessed dramatic expansion with 6 papers in 2023, 29 papers in 2024, and an impressive 64 papers in 2025 alone, which can be seen in Figure 5. This research spans multiple disciplinary boundaries, appearing across premier venues in artificial intelligence, electronic design automation, and emerging interdisciplinary forums. However, this phenomenon has resulted in fragmented knowledge across diverse research communities with inconsistent terminologies, evaluation metrics, and optimization approaches. Despite this rapid growth, the field lacks a comprehensive systematic review unifying these scattered contributions. This gap prevents the research community from identifying promising directions, understanding how well different approaches work, establishing best practices, and avoiding repeated efforts.

To fill this critical research gap, we conducted a rigorous Systematic Literature Review (SLR) following established methodologies [10]. As shown in Figure 4, we employed the Quasi-Gold Standard (QGS) search strategy [11], combining manual searches across seven premier conferences and journals (AAAI, ACL, ICML, ICLR, NeurIPS, DAC, and TCAD) with automated searches spanning six academic databases (IEEE Xplore, ACM Digital Library, ScienceDirect, Web of Science, SpringerLink, and arXiv). Our comprehensive search yielded a substantial pool of candidate papers, which underwent systematic three-stage filtering including title/abstract screening, full-text assessment, and quality evaluation. Supplementary forward and backward snowballing identified additional relevant studies. This rigorous process yielded 102 high-quality papers (70 peer-reviewed publications and 32 rigorously assessed preprints) spanning 2020-2025, forming the empirical foundation for our analysis. Our investigation addresses four fundamental research questions examining the LLMs employed (RQ1), dataset construction and evaluation methodologies (RQ2), adaptation and optimization techniques (RQ3), and alignment approaches for human-centric requirements (RQ4).

 doi:10.20944/preprints202511.0656.v1

Related Literature Reviews. While several surveys have examined LLM applications in code generation, software engineering, and hardware design automation, our work represents the first comprehensive review focusing specifically on Verilog code generation. Table 1 summarizes how our work differs from existing literature reviews. Prior surveys fall into two primary categories: (1) *general code generation surveys* that predominantly concentrate on high-level programming languages like Python and Java but neglect hardware description languages, and (2) *broad EDA surveys* that cover multiple aspects of hardware design automation without in-depth analysis of LLM-based Verilog generation.

General code generation [12,17] and low-resource domain-specific programming language surveys [6] provide valuable insights into LLM architectures and code generation techniques, but neither addresses the unique challenges of hardware description languages such as timing constraints, synthesizability requirements, and physical design considerations. Recent EDA-focused surveys [14–16] examine broader applications of AI in electronic design automation, including various stages of the design flow. While some mention LLMs for RTL generation, they lack systematic analysis of Verilog-specific datasets, evaluation methodologies, and alignment techniques. In contrast, our survey presents the first comprehensive review of LLMs for Verilog code generation, systematically analyzing model architectures, dataset construction, evaluation methods, and alignment techniques from both hardware design and AI perspectives.

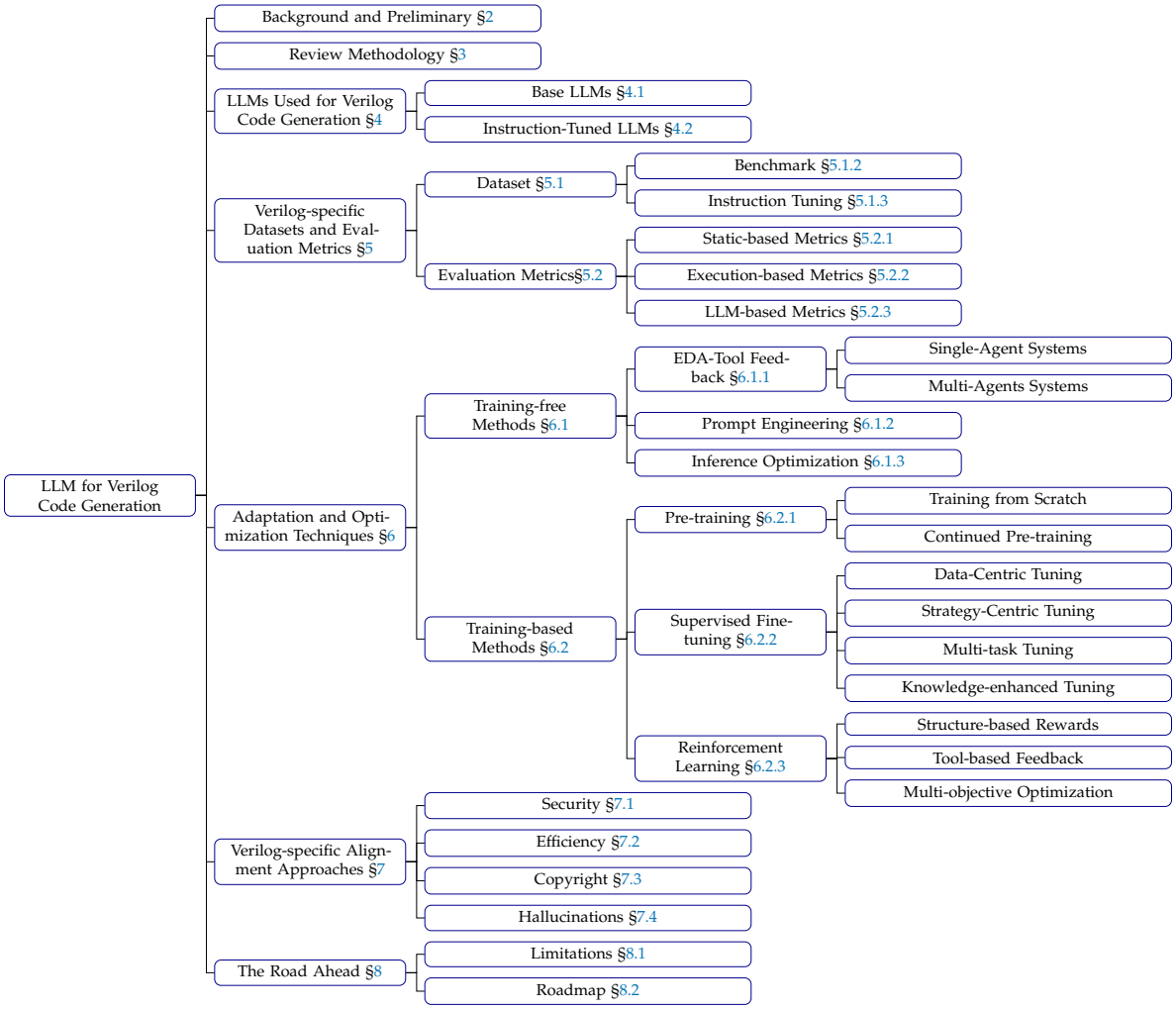


Figure 1. Structure of This Survey

In general, this study makes the following **contributions**:

 doi:10.20944/preprints202511.0656.v1

- We present the first systematic literature review on LLM-based Verilog code generation, analyzing 102 papers (70 peer-reviewed, 32 high-quality preprints) spanning 2020-2025.
- We provide comprehensive taxonomy and trend analysis of LLMs for Verilog generation (RQ1), systematically analyze dataset construction and evaluation evolution across 27 benchmarks and 34 training datasets (RQ2), investigate adaptation and optimization techniques (RQ3), and examine alignment techniques addressing human-centric requirements including security, efficiency, copyright, and hallucinations (RQ4).
- We identify key research limitations and propose a comprehensive roadmap for future directions in LLM-assisted hardware design.

Survey Structure. Figure 1 illustrates the organization of this survey. Section 2 provides essential background on Large Language Models and Verilog code generation within the EDA workflow. Section 3 details our systematic review methodology including search strategy, selection criteria, and quality assessment procedures. Section 4 examines the landscape of LLMs employed for Verilog code generation, categorizing Base LLMs and instruction-tuned models with trend analysis. Section 5 investigates dataset construction methodologies and evaluation metrics, analyzing both benchmark and instruction-tuning datasets alongside the evolution of assessment approaches. Section 6 explores adaptation and optimization techniques for enhancing LLM performance in Verilog generation tasks. Section 7 analyzes alignment approaches addressing security, efficiency, copyright, and hallucination challenges. Section 8 discusses limitations of current research and presents a roadmap for future directions. Finally, Section 10 concludes the survey with key takeaways and perspectives on the field's trajectory.

Table 1. Comparison with Related Literature Reviews

Survey	Year	Papers	Topics	Verilog Focus
General Code Generation Surveys				
Jiang et al. [12]	2024	235	General Programming Languages	✗
Joel et al. [6]	2024	111	Low-Resource and Domain-Specific Programming Languages	Partial
Chen et al. [13]	2025	601	Deep Learning-based Software Engineering	✗
EDA-focused Surveys				
Fang et al. [14]	2025	250	Layout/Netlists/HDL/Assertion Generation	Partial
He et al. [15]	2024	211	EDA Generation, Verification, and Debugging	Partial
Chen et al. [16]	2024	≈ 200	A Paradigm Shift from AI4EDA towards AI-rooted EDA (full-workflow AI4EDA)	Partial
Our Verilog-specific Survey				
Our work	**2025**	**102**	Verilog Code Generation	✓

2. Background and Preliminaries

2.1. Large Language Models

LLMs represent a major breakthrough in artificial intelligence, characterized by massive scale (billions to trillions of parameters) and transformer-based architectures [18]. Through self-supervised learning on vast text corpora, these models acquire sophisticated capabilities in understanding linguistic patterns, encoding domain knowledge, and performing complex reasoning tasks.

Since the introduction of GPT-3 [19] in 2020, LLMs have demonstrated remarkable proficiency in natural language understanding and generation across diverse domains. Modern LLMs fall into two main categories: *base models*, pretrained on large-scale general corpora to establish foundational language understanding, and *instruction-tuned models*, further refined through supervised fine-tuning or reinforcement learning from human feedback (RLHF) [20] to better follow instructions and align with human preferences.

The power of LLMs lies in their adaptability to domain-specific tasks through various techniques. *Prompt engineering* guides model behavior through carefully designed input templates; *in-context learning* enables few-shot adaptation without updating parameters; and *fine-tuning* adjusts model parameters on task-specific datasets. These capabilities have made LLMs highly effective for automating

 doi:10.20944/preprints202511.0656.v1

complex tasks, with code generation emerging as a particularly successful application that bridges natural language and formal programming languages.

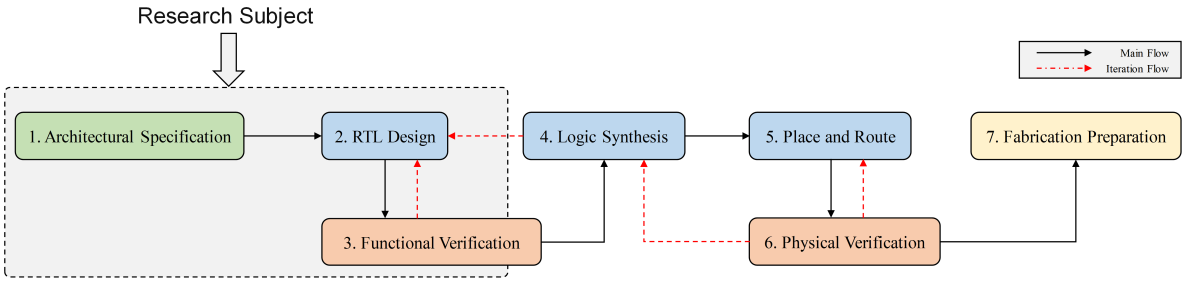


Figure 2. The EDA workflow for integrated circuit development.

2.2. Verilog Code Generation

As illustrated in Figure 2, the EDA workflow for integrated circuit development follows a systematic sequence of seven interconnected phases with iterative refinement loops. The process begins with *Architectural Specification* (Phase 1), where system-level requirements and functionalities are formally defined. This feeds into *RTL Design* (Phase 2), where HDLs like Verilog describe the hardware architecture through registers, data paths, and control logic. *Functional Verification* (Phase 3) validates behavioral correctness against specifications through simulation using testbenches, which are structured verification environments containing test vectors, stimulus generators, and monitoring logic to systematically exercise the design under test, with iteration back to RTL design when errors are detected. Subsequently, *Logic Synthesis* (Phase 4) transforms verified RTL into gate-level netlists. Following synthesis, timing simulation verifies timing correctness at the netlist level, potentially iterating with verification for optimization. *Place and Route* (Phase 5) determines physical component placement and interconnections on the silicon die, followed by *Physical Verification* (Phase 6) that checks design rules, timing constraints, and signal integrity with iterative refinement loops. Finally, *Fabrication Preparation* (Phase 7) generates GDSII files and manufacturing documentation for trial tape-out. After successful physical testing of the fabricated prototype, the design proceeds to mass production.

Natural Language Descriptions

Implement a module of a counter design that requires counting from 4 'b0000 to 4' d11. The counting can be controlled by the input signal valid_count. That is, the count is paused if valid_count is 0. The counter increments on each clock cycle when the valid_count signal is active and resets to 0 when the reset signal (rst_n) is active.

Module name:
counter_12
Input ports:
rst_n: Reset signal (active low)
clk: Clock signal
valid_count: Signal to enable counting
Output ports:
out: 4-bit output representing the current count value

Implementation and Constraint:
If the reset signal is active (!rst_n), the counter is reset to 0 by assigning the value 4'b0000 to the output register (out). If the valid_count signal is 1, the counter increments. It checks if the current count value (out) is equal to 4'd11 (maximum count value). If it is, the counter wraps around and resets to 0 by assigning the value 4'b0000 to the output register (out). Otherwise, it increments the output register by 1. If the valid_count = 0, the counter will stop, and the output register (out) remains unchanged.

Verilog Code

```
`timescale 1ns/1ps

module counter_12
(
  input rst_n,
  input clk,
  input valid_count,
  output reg [3:0] out
);
  always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
    begin
      out <= 4'b0000;
    end
    else if (valid_count)
    begin
      if (out == 4'd11)
      begin
        out <= 4'b0000;
      end
      else begin
        out <= out + 1;
      end
    end
    else begin
      out <= out;
    end
  end
endmodule
```

Figure 3. An Example of Verilog Code Generation.

Verilog code generation, as highlighted in the dashed box of Figure 2, primarily targets the first three phases of this workflow, forming a critical iterative cycle. Natural language specifications from architectural design are transformed into synthesizable Verilog code during RTL design, which is then validated through functional verification. The iterative feedback loop between verification and RTL

 doi:10.20944/preprints202511.0656.v1

design enables refinement until correctness is achieved. The quality of generated Verilog code directly impacts all downstream processes, influencing synthesis optimization opportunities, achievable clock frequencies, power consumption profiles, and ultimately manufacturing feasibility.

LLM-based Verilog code generation can be formally represented as a mapping function $f_\theta : (D, I) \rightarrow V$, where θ represents the model parameters, D denotes natural language descriptions encompassing both functional specifications (design intent and behavioral requirements) and design constraints (timing requirements, area budgets, power targets, interface protocols), I denotes optional multimodal inputs (circuit diagrams, timing waveforms, state machine diagrams), and V is the generated Verilog code. Text-only generation, which represents the majority of current approaches, simplifies to $f_\theta : D \rightarrow V$.

The correctness evaluation of generated Verilog code employs a verification function $verify : (V, S, T) \rightarrow \{pass, fail\} \times M$, where V represents the generated code, S denotes specifications or reference implementations, T represents testbenches with input-output test vectors, and M captures quality metrics. This evaluation encompasses: (1) *syntactic validation* through HDL parsers ensuring compilability; (2) *functional correctness* via simulation-based testing where $\forall t \in T : sim(V, t) = expected(t)$; (3) *semantic similarity* measuring structural correspondence with reference implementations through AST comparison or text-based metrics; and (4) *formal equivalence* proving mathematical equivalence between generated and reference designs using formal verification tools.

Figure 3 illustrates a practical example from RTLLM-v2 [21], where a natural language specification is transformed into synthesizable RTL code for a 4-bit counter (0 to 11) with pause and reset functionality. This example demonstrates essential HDL design elements: synchronous sequential logic with asynchronous reset (*always @(posedge clk)*), conditional state transitions based on control signals, and bounded counting behavior. The natural language input contains both functional specifications (counting logic, range limits) and behavioral requirements (reset handling, pause control), which must be correctly interpreted into standard Verilog constructs. In terms of our formal representation, D is the counter specification, V is the generated Verilog module, and verification would involve testbenches T that validate counting behavior under various conditions.

3. Review Methodology

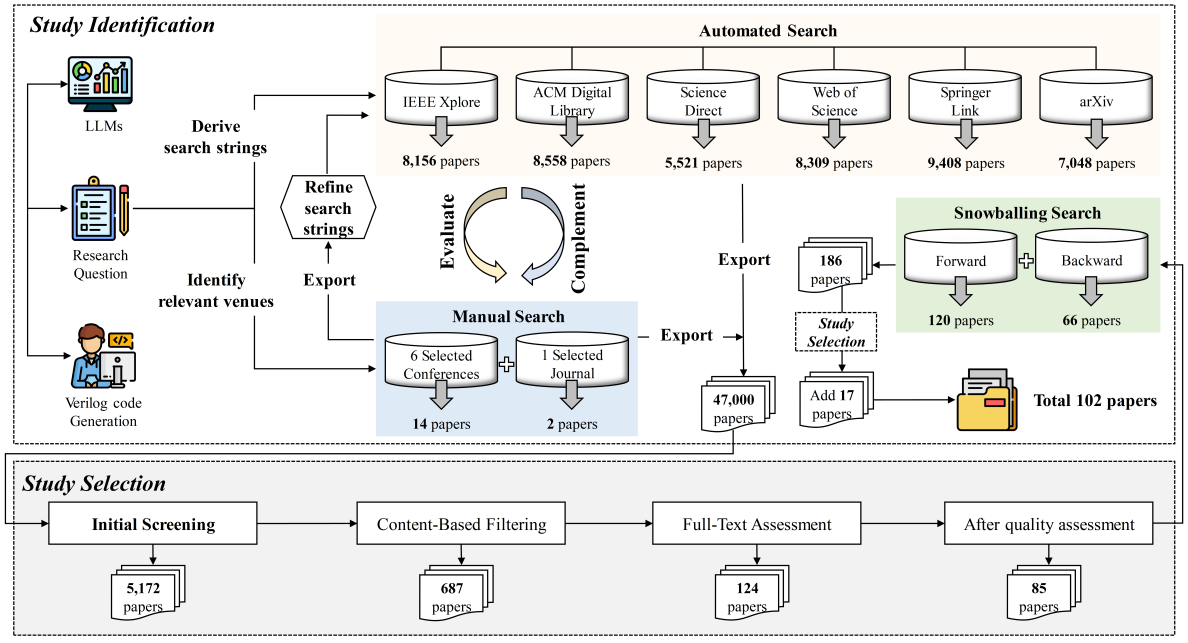


Figure 4. Study identification and selection process.

 doi:10.20944/preprints202511.0656.v1

Following the guide by Kitchenham et al. [10], which is used in most other SLRs [22–24], our methodology is organized as follows: planning the review (i.e., Section 3.1, 3.2), conducting the review (i.e., Section 3.3), and analyzing the basic review results (i.e., Section 3.4).

3.1. Research Question

To provide a comprehensive overview of LLM for Verilog code generation, it is important to fully understand the construction of Verilog domain-specific datasets, how LLMs are currently being applied in Verilog code generation, the challenges they face, and their potential future research directions in this domain. Thus, we aim to provide an SLR of the application of LLMs to Verilog code generation. This study thus aims to answer the following research questions:

RQ1: What LLMs have been employed to solve Verilog code generation tasks? Identifying the architectures of LLMs utilized in Verilog code generation is fundamental to establishing the current technological landscape and understanding model selection patterns in this emerging field. This comprehensive taxonomy enables systematic comparative analysis of different LLMs (i.e., general-purpose Base LLMs vs. domain-specific IT LLMs, open-source vs. closed-source) and their adoption trends, revealing the evolution of model preferences and the growing diversity in this rapidly expanding research domain.

RQ2: How are Verilog-specific datasets constructed and what evaluation metrics are utilized to assess LLMs? Understanding the construction methodologies of Verilog-specific datasets and the evaluation metrics employed for LLM assessment is critical, as these factors directly influence the reliability of conclusions drawn about model capabilities. This investigation is pivotal for interpreting model performance, ensuring benchmark validity, and guiding dataset optimization strategies for Verilog code generation tasks.

RQ3: What adaptation and optimization techniques are applied in Verilog code generation? Examining the adaptation and optimization techniques employed in Verilog code generation is crucial for revealing effective strategies that customize general-purpose LLMs to accommodate the unique syntax, semantics, and constraints of Verilog. This understanding is central to enhancing model performance in domain-specific tasks and advancing the field's methodological foundations.

RQ4: What alignment techniques are employed for meeting human-centric requirements? Investigating alignment techniques is essential for ensuring that LLM-generated Verilog code conforms to human intentions and values. This becomes particularly critical considering potential risks such as copyright infringement, security vulnerabilities, inefficiencies, and trustworthiness issues, which may have severe consequences when code is implemented by users without specialized programming expertise in hardware design.

3.2. Search Strategy

As shown in Fig.4, we employed the "Quasi-Gold Standard" (QGS) [11] approach for comprehensive paper search. The QGS methodology involves manual search to identify a set of relevant studies from which search strings are extracted. Subsequently, these search strings are utilized to perform automated searches, followed by snowballing techniques to further supplement the search results. Finally, we applied a series of rigorous filtering procedures to obtain the most pertinent studies for our review.

Given that the first application of LLMs to Verilog code generation emerged in 2020 [9], our search strategy specifically targets publications from 2020 to 2025, ensuring comprehensive coverage of this emerging research domain.

 doi:10.20944/preprints202511.0656.v1

Table 2. Publication venues for manual search.

Acronym	Venues
AAAI	AAAI Conference on Artificial Intelligence
ACL	Annual Meeting of the Association for Computational Linguistics
ICML	International Conference on Machine Learning
ICLR	International Conference on Learning Representations
NeurIPS	Conference on Neural Information Processing Systems
DAC	Design Automation Conference
TCAD	IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

3.2.1. Search Items

During the manual search phase, we selected eight premier conferences and journals spanning artificial intelligence, machine learning, and electronic design automation domains (i.e., AAAI, ACL, ICML, ICLR, NeurIPS, DAC, and TCAD, as detailed in Table 2), and systematically retrieved papers applying LLMs to Verilog code generation tasks. Following the manual search process, we identified 16 papers (14 papers from conferences and 2 papers from journals) that aligned with our research objectives. These 16 relevant publications constitute the foundational corpus for constructing our QGS, serving as the basis for extracting comprehensive search terms and validating our automated search strategy.

Our search string construction follows a systematic approach by combining two complementary keyword sets: one pertaining to Verilog code generation tasks and another related to LLMs. A paper is considered potentially relevant only when it contains keywords from both categories, thereby ensuring higher precision in our search results. The comprehensive search keyword sets are as follows:

- **Keywords related to Verilog code generation (12 terms):** Verilog, HDL, Hardware Description Language, RTL, Register Transfer Level, Digital Design, Hardware Design, Electronic Design Automation, EDA, Verilog Generation, FPGA, ASIC
- **Keywords related to LLMs (13 terms):** LLM, Large Language Model, Language Model, GPT, ChatGPT, Transformer, fine-tuning, prompt engineering, In-context learning, Natural Language Processing, NLP, Machine Learning, AI

Here is our used command search string:

```
(Verilog OR "Hardware Description Language" OR "Register Transfer Level" OR
    "Digital Design" OR "Hardware Design" OR "Electronic Design Automation" OR
    "EDA" OR "Verilog Generation" OR "FPGA" OR "ASIC")
AND
(LLM OR "Large Language Model" OR "Language Model" OR "GPT" OR "ChatGPT" OR
    "Transformer" OR "fine-tuning" OR "prompt engineering" OR "In-context learning"
    OR "Natural Language Processing" OR "NLP" OR "Machine Learning" OR AI)
```

It is important to note that our LLM-related keyword list encompasses broader terms such as machine learning and deep learning, which may not be exclusively associated with LLMs. This inclusive approach is deliberately adopted to maximize recall and minimize the risk of omitting potentially relevant studies, thereby broadening our search scope during the automated search process to ensure comprehensive coverage of the literature.

3.2.2. Search Databases

Following the establishment of our search strings, we conducted systematic automated searches across six widely recognized academic databases: IEEE Xplore [25], ACM Digital Library [26], ScienceDirect [27], Web of Science [28], SpringerLink [29], and arXiv [30]. Consistent with our temporal scope targeting publications from 2020 onward, we applied appropriate date filters to each database search. The automated search process yielded 8,156 papers from IEEE Xplore, 8,558 papers from ACM Digital Library, 5,521 papers from ScienceDirect, 8,309 papers from Web of Science, 9,408 papers from

 doi:10.20944/preprints202511.0656.v1

SpringerLink, and 7,048 papers from arXiv, resulting in a total of about 47,000 papers for subsequent screening and evaluation.

3.3. Search Selection

Table 3. Inclusion criteria and Exclusion criteria.

Inclusion criteria	
1)	The paper claims that an LLM is used.
2)	The paper claims that the study involves to solve Verilog code generation.
3)	The paper with accessible full text and must be written in English.
Exclusion criteria	
1)	Short papers whose number of pages is less than 5.
2)	Books, records, theses, tool demos papers, editorials, or venues not subject to a full peer-review process.
3)	The paper is a literature review or survey.
4)	The paper mentions LLMs only in future work or discussions rather than using LLMs in the approach.
5)	The paper does not involve Verilog code generation.
6)	Duplicate papers or similar studies authored by the same authors.

3.3.1. Inclusion and Exclusion Criteria

To ensure methodological rigor and systematic paper selection, we established well-defined inclusion and excl usion criteria grounded in established best practices from leading systematic literature reviews [31,32], as presented in Table 3. These criteria were meticulously designed to guarantee that selected studies directly address our research questions while maintaining objectivity and reproducibility.[1]

Our paper selection process employed a three-stage filtering approach to systematically reduce the candidate pool while preserving study quality:

Stage 1 (Initial Screening): We applied exclusion criteria 1 and 6 to eliminate short papers (< 5 pages) and duplicate publications, reducing our corpus to 5,172 papers. This preliminary filter ensured that only substantial, original contributions proceeded to detailed evaluation.

Stage 2 (Content-Based Filtering): Manual examination of publication venues, titles, and abstracts further refined our selection to 687 papers. We deliberately retained high-quality arXiv preprints to capture emerging research trends while excluding non-peer-reviewed materials such as books, keynote records, technical reports, theses, tool demonstrations, editorials, and survey papers (exclusion criteria 2-3).

Stage 3 (Full-Text Assessment): Comprehensive full-text review enabled precise relevance determination. Studies exclusively addressing Verilog code repair, optimization, or test case generation without code generation components were excluded (exclusion criterion 5), except where these activities served as auxiliary components enhancing generation performance. We also eliminated studies that mentioned LLMs only conceptually in future work sections without implementation (exclusion criterion 4). This final stage yielded 124 primary studies directly relevant to our research topic.

[1] For preprint papers released on arXiv, we performed manual quality assessments to determine their eligibility for inclusion based on methodological soundness and contribution relevance.

 doi:10.20944/preprints202511.0656.v1

3.3.2. Quality Assessment

Table 4. Checklist of Quality Assessment Criteria (QAC).

ID	Quality Assessment Criteria
QAC1	Was the study published in a prestigious venue?
QAC2	Does the study make a contribution to the academic or industrial community?
QAC3	Does the study provide a clear description of the workflow and implementation of the proposed approach?
QAC4	Are the experiment details, including datasets, baselines, and evaluation metrics, clearly outlined?
QAC5	Do the findings from the experiments strongly support the main arguments presented in the study?

To mitigate potential bias from low-quality studies and provide readers with transparent quality indicators, we implemented a comprehensive quality assessment framework. Our evaluation protocol comprises five Quality Assessment Criteria (QACs), as detailed in Table 4, designed to systematically evaluate the relevance, methodological rigor, clarity, and scholarly significance of selected studies [33].

Each quality assessment criterion was evaluated using a four-point Likert scale (0-3), where 0 indicates the lowest quality and 3 represents the highest quality. To maintain high standards while accommodating methodological diversity, we established a threshold of 12 points (representing 80% of the maximum possible score of 15) for study inclusion. For arXiv preprints lacking formal publication venues, we assigned a score of 0 for QAC1 (venue prestige) while maintaining rigorous evaluation of the remaining criteria. Preprints achieving the overall threshold score of 12 were retained to capture cutting-edge research contributions that may not yet have undergone traditional peer review but demonstrate substantial methodological and theoretical merit.

Following quality assessment, our final corpus comprised 85 studies: 55 peer-reviewed publications from prestigious venues and 30 high-quality preprints that met our established criteria. This balanced approach ensures both scholarly rigor and inclusivity of emerging research trends in the rapidly evolving field of LLM-based Verilog generation.

3.3.3. Forward and Backward Snowballing

To ensure comprehensive literature coverage and minimize the risk of overlooking relevant studies during our manual and automated search processes, we conducted systematic forward and backward snowballing procedures [34]. This supplementary search strategy involves examining both the reference lists of our selected primary studies (backward snowballing) and publications that subsequently cite these studies (forward snowballing).

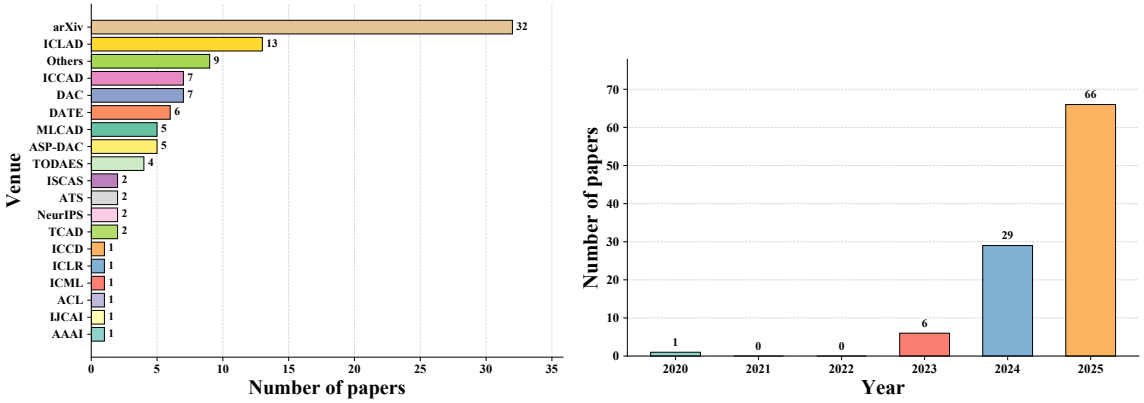
The snowballing process was implemented as follows: First, we systematically reviewed the reference lists of our 85 primary studies to identify potentially relevant publications that may have been missed in our initial search. Second, we utilized citation databases to identify papers that cite our selected studies, focusing on recent publications that might represent emerging developments in the field. The supplementary search yielded 186 additional papers, which were subsequently subjected to our established selection pipeline, including screening against inclusion and exclusion criteria, duplicate removal, and quality assessment. Following this rigorous evaluation process, we identified 15 additional relevant studies, resulting in a final corpus of 102 papers (70 peer-reviewed publications and 32 high-quality preprints) that form the foundation of our systematic literature review.

3.4. Data Extraction and Analysis

Through our comprehensive search and snowballing procedures, we ultimately obtained 102 relevant research papers. Figure 5 provides an overview of the distribution of our selected papers. As shown in Figure 5(a), 68 papers were published in peer-reviewed venues. ICCAD emerges as the most prominent venue among these, contributing 13 papers. Other significant contributing venues include DAC, DATE, and TCAD, which contributed 7, 6, and 5 papers respectively. Additionally, we observe that prominent AI-related venues such as NeurIPS, ICLR, ICML, ACL, and AAAI also contributed a substantial number of publications. This distribution indicates that LLM for Verilog code generation

 doi:10.20944/preprints202511.0656.v1

has garnered attention not only within hardware-related research communities but also within AI and machine learning research domains, reflecting the interdisciplinary nature of this emerging field.



(**a**) Distribution of papers across venues. (**b**) Distribution of papers over years.

Figure 5. Overview of the selected 102 papers' distribution.

The remaining 31 papers were published on arXiv and 1 paper were published on TechRxiv, an open-access platform serving as a repository for scholarly articles. This finding is not surprising, as many novel LLM for Verilog code generation studies are rapidly emerging, and numerous works have recently been completed and may currently be undergoing peer-review processes. Although these papers have not undergone formal peer review, we applied rigorous quality assessment procedures to all collected papers to ensure the validity and quality of our research findings. This approach enables us to include all high-quality and relevant publications while maintaining high research standards.

Figure 5(b) illustrates the temporal distribution of our selected papers. A rapid growth trend in publication numbers has been observed since 2023. There was only one relevant paper in 2020, while 2021 and 2022 saw no relevant publications. However, from 2023 to 2024, the number of papers increased dramatically from 6 to 29. Remarkably, by September 2025 alone, the number of published papers had already reached 66. This rapid growth trajectory demonstrates the increasing research interest in the LLM for Verilog code generation domain, highlighting its emergence as a significant area of investigation.

To visualize the primary content themes of our paper collection, we generated a word cloud based on the abstracts of all 102 papers, as shown in Figure 6. The most frequently appearing terms include "Verilog", "RTL", "HDL", "LLM", "dataset", "performance", "quality", "functional", "syntax", and "feedback", clearly indicating the main topics explored in these papers. The terms "Verilog", "RTL", and "HDL" emphasize the core hardware description language elements, while "LLM", "model", and "code" represent the utilization of large language models in Verilog code generation tasks. The term "dataset" underscores the critical need for constructing Verilog domain-specific datasets in LLM for Verilog code generation research. Meanwhile, terms such as "performance", "quality", "functional", "syntax", and "feedback" reflect ongoing discussions about the effectiveness and evaluation of LLMs in Verilog code generation. The word cloud provides further visual evidence that our collected literature is highly relevant to our research focus.

 doi:10.20944/preprints202511.0656.v1

Figure 6. Topics discussed in the collected papers.

4. RQ1: LLMs Employed for Verilog Code Generation

Through our systematic literature review of 102 included studies, we identify that researchers have employed a diverse range of LLMs for Verilog code generation tasks.

Statistical Scope. Our analysis encompasses all LLMs utilized in the reviewed papers, whether for *method implementation* or as *baseline comparisons*, ensuring comprehensive coverage of the LLM adoption landscape in Verilog code generation research. Based on their adaptation for Verilog tasks, we categorize these LLMs into two main types: **Base LLMs** (*general-purpose models without Verilog-specific adaptation*) and **IT LLMs** (*models specifically Instruction-Tuned for Verilog code generation*).

4.1. Analysis of Base LLMs

As shown in Table 5 and Table 6, Base LLMs represent the majority of models employed in Verilog code generation research, with 383 total mentions across all studies. Despite not being specifically designed for hardware description languages, these models demonstrate substantial capability in understanding and generating Verilog code through their general programming language comprehension.

 doi:10.20944/preprints202511.0656.v1

Table 5. Open-Source LLMs Usage in Verilog Code Generation Studies (2023-2025)

LLM Family	LLM Name	2023	2024	2025	Total
Llama Series	CodeLlama	0	7	16	23
	Llama	0	0	1	1
	Llama2	0	2	5	7
	Llama3	0	4	7	11
	Llama3.1	0	0	18	18
	Llama3.2	0	0	1	1
	Llama3.3	0	0	1	1
	LlaVa	0	2	0	2
	Subtotal	**0**	**15**	**49**	**64**
DeepSeek Series	DeepSeek-Coder	0	6	21	27
	DeepSeek-Coder-V2	0	0	5	5
	DeepSeek-R1	0	0	8	8
	DeepSeek-R1-Distill	0	0	5	5
	DeepSeek-V2	0	0	1	1
	DeepSeek-V2.5	0	0	1	1
	DeepSeek-V3	0	0	7	7
	Subtotal	**0**	**6**	**48**	**54**
Qwen Series	CodeQwen	0	3	10	13
	Qwen2.5-Coder	0	0	12	12
	Qwen3	0	0	1	1
	Qwen-vl	0	0	1	1
	QWQ	0	0	1	1
	Subtotal	**0**	**3**	**25**	**28**
CodeGen Series	CodeGen	3	2	0	5
	CodeGen2	0	4	1	5
	CodeGen2.5	0	0	2	2
	Subtotal	**3**	**6**	**3**	**12**
StarCoder Series	StarCoder	0	4	3	7
	StarCoder2	0	1	3	4
	Subtotal	**0**	**5**	**6**	**11**
Others	Various Models	2	11	22	35
Total Open-Source		**5**	**46**	**153**	**204**

Table 6. Closed-Source LLMs Usage in Verilog Code Generation Studies (2023-2025)

LLM Family	LLM Name	2023	2024	2025	Total
GPT Series	code-davinci-002	1	1	0	2
	GPT-3.5	3	16	24	43
	GPT-4	3	20	29	52
	GPT-4o	0	4	31	35
	GPT-4-turbo	0	0	2	2
	GPT-4V	0	1	1	2
	GPT-o1	0	0	6	6
	GPT-o3	0	0	4	4
	GPT-o4	0	0	3	3
	Subtotal	**7**	**42**	**100**	**149**
Claude Series	Claude 2	0	1	0	1
	Claude 3	0	2	2	4
	Claude 3.5	0	2	6	8
	Claude 3.7	0	0	5	5
	Claude 4	0	0	1	1
	Subtotal	**0**	**5**	**14**	**19**
Gemini	Gemini	0	1	4	5
Others	Various Models	0	3	3	6
Total Closed-Source		**7**	**51**	**121**	**179**

 doi:10.20944/preprints202511.0656.v1

4.1.1. Open-Source Base LLMs

Open-source Base LLMs demonstrate significant diversity and adoption, accounting for 204 mentions (53.3% of all Base LLM usage). The landscape is characterized by several key model families:

(1) Llama Family Leadership

The Llama ecosystem [35] emerges as the most successful open-source family with 64 total mentions. While CodeLlama (23 mentions) [2] leads within this family, it remains a general code generation model rather than a Verilog-specific adaptation. The rapid adoption of Llama3.1 (18 mentions) [36] demonstrates the community's preference for the latest general-purpose models.

(2) DeepSeek Series Growth

DeepSeek models [37] secure strong adoption with 54 mentions, led by DeepSeek-Coder (27 mentions) [38]. The emergence of reasoning variants like DeepSeek-R1 (8 mentions) [39] reflects growing interest in advanced reasoning capabilities for complex Verilog generation tasks.

(3) Qwen Series Growth

Qwen series shows promising growth with 28 mentions, led by CodeQwen (13 mentions) [40]. The introduction of Qwen2.5-Coder (12 mentions) [41] continues this trend toward enhanced code generation capabilities.

(4) Code-Oriented Models

Traditional code generation models like CodeGen (12 mentions) [42,43] and StarCoder (11 mentions) [44,45] maintain steady usage, though their adoption remains modest compared to general-purpose foundation models. This suggests that scale and general capability often outweigh domain-specific code training for Verilog tasks.

(5) Other Models

A substantial portion (35 mentions) represents various other open-source models (e.g., CodeGemma [46], CodeT5+ [47], Phi [48,49], Nemotron [50])[2], indicating the experimental nature of the field and researchers' willingness to explore diverse solutions.

4.1.2. Closed-Source Base LLMs

Closed-source Base LLMs demonstrate strong adoption with 179 total mentions (46.7% of all Base LLM usage), dominated by major technology companies' offerings:

(1) GPT-series LLMs

OpenAI's GPT series [51] dominates the closed-source landscape with 149 mentions (83.2% of closed-source usage). GPT-4 leads with 52 mentions, followed by GPT-3.5 (43 mentions) and GPT-4o (35 mentions). The introduction of reasoning models (GPT-o1: 6 mentions, GPT-o3: 4 mentions) reflects researchers' exploration of advanced reasoning capabilities for complex Verilog code generation tasks.

(2) Other LLMs

Anthropic's Claude series [52] shows steady growth with 19 mentions, led by Claude 3.5 (8 mentions). Google's Gemini [53] represents a newer entrant with growing adoption (5 mentions) for Verilog code generation tasks.

[2] Due to page limitations, we place the full statistical results of other models on the project homepage.

 doi:10.20944/preprints202511.0656.v1

4.1.3. Trends Analysis

The field demonstrates unprecedented expansion, with total LLM mentions growing from 12 in 2023 to 274 in 2025 (2,183% growth). This reflects the rapid maturation of LLM technology and researchers' increasing confidence in applying these models to hardware design tasks.

We first analyze the trends in LLM adoption for Verilog code generation from the perspective of Open-Source and Closed-Source, as well as the Architectural preferences of Base LLMs.

(1) Open-Source vs. Closed-Source

While both categories show strong growth, their trajectories differ significantly, with distinct model evolution patterns:

- **Open-Source LLMs:** Demonstrate accelerated growth ($5 \rightarrow 46 \rightarrow 153$ mentions) with 233% increase from 2024 to 2025. The evolution shows a clear shift from specialized code models to general-purpose foundation models:
  - **2023:** Dominated by traditional code generation models (CodeGen: 3 mentions)
  - **2024:** Transition to foundation families (Llama: 15, DeepSeek: 6, Qwen: 3 mentions)
  - **2025:** Widespread adoption of latest foundation models (Llama: 49, DeepSeek: 48, Qwen: 25 mentions)
- **Closed-Source LLMs:** Show steady expansion ($7 \rightarrow 51 \rightarrow 121$ mentions) with 137% growth rate, reflecting market diversification:
  - **2023:** Exclusively OpenAI models (GPT-3.5, GPT-4: 7 mentions)
  - **2024:** GPT dominance continues (42 mentions) while competitors emerge (Claude: 5 mentions, Gemini: 1 mention)
  - **2025:** Market expansion with new reasoning models (GPT-o1/o3) and stronger competition from Claude 3.5 and Gemini

(2) Architectural Preferences

Most employed LLMs adopt Decoder-only architectures, reflecting preference for autoregressive generation. However, some studies have explored Encoder-Decoder models (e.g., CodeT5+ [47]), demonstrating architectural diversity. Additionally, Vision-Language Models (GPT-4V, LlaVa [54]) are increasingly used for translating visual design specifications into Verilog code, indicating evolution toward multimodal input capabilities.

4.2. Analysis of IT LLMs

As shown in Table 7, we identify 34 instruction-tuned (IT) LLMs specifically built for Verilog code generation. A defining feature of this ecosystem is its comparatively high level of openness: 19 models (55.9%) provide public weights while 15 (44.1%) remain closed. This open-weight ratio exceeds typical rates in general-purpose LLMs, reflecting a stronger community emphasis on reproducibility and downstream evaluability for hardware design research.

 doi:10.20944/preprints202511.0656.v1

Table 7. Instruction-Tuned LLMs for Verilog Code Generation (Classified by Model Weight Availability).

Verilog-Specific LLM	Foundation Model	Weight	URL
Closed-Source (Model Weights)			
AutoVCoder [55]	CL/CQ/DS-Coder	Closed	-
BetterV [56]	CL/CQ/DS-Coder	Closed	-
CodeGen-Verilog [57]	CodeGen	Closed	-
CraftRTL [58]	CL/DS-Coder/StarCoder2	Closed	-
DeepRTL [59]	CodeT5+	Closed	-
DeepRTL2 [60]	Llama3.1/DS-Coder	Closed	-
FreeV [61]	Llama3.1	Closed	-
ITERTL [62]	DS-Coder/DS-Coder-v2	Closed	-
MEV-LLM [63]	CodeGen/Gemma	Closed	-
OpenRTLSet [64]	Qwen2.5	Closed	-
PyraNet [65]	CL/DS-Coder	Closed	-
RTL++ [66]	CL	Closed	-
RTLRepoCoder [67]	DS-Coder	Closed	-
ScaleRTL [68]	DS-R1-Distill-Qwen	Closed	-
Veritas [69]	Llama-3.2	Closed	-
Open-Source (Model Weights)			
DAVE [9]	GPT-2	Open	https://shorturl.asia/VWPTn
ChipGPT [70]	Llama2/Llama3	Open	https://www.modelscope.cn/profile/changkaiyan
CodeV [71]	CQ/DS-Coder/QC	Open	https://huggingface.co/zhuyaoyu
CodeV-R1 [72]	QC	Open	https://huggingface.co/zhuyaoyu
GEMMV [73]	DS-R1-Distill-Qwen/Llama3.1	Open	https://huggingface.co/bxsk2024
Hardware Phi-1.5B [74]	Phi-1.5B	Open	https://huggingface.co/KSU-HW-SEC/Hardware_Phi_30k_version
HAVEN [75]	CL/DS-Coder/CQ	Open	https://huggingface.co/yangyiyao
MG-Verilog [76]	CL	Open	https://shorturl.asia/rDNSm
Mistral-Verilog [77]	Mistral	Open	https://huggingface.co/emilgoh/mistral-verilog
Origen [78]	DS-Coder	Open	https://huggingface.co/henryen/OriGen
ReasoningV [79]	QC	Open	https://modelscope.cn/models/GipsyAI/ReasoningV-7B
RTLCoder [80]	Mistral/DS-Coder	Open	https://github.com/hkust-zhiyao/RTL-Coder
VeriCoder [81]	QC	Open	https://huggingface.co/LLM4Code/VeriCoder_Qwen14B
VeriGen [82]	CodeGen	Open	https://huggingface.co/shailja
VeriLogos [83]	DS-Coder	Open	https://huggingface.co/97kjmin/VeriLogos
VeriPrefer [84]	Mistral/CL/DS-Coder/CQ/QC	Open	https://shorturl.asia/sMj3J
VeriReason [85]	QC/CL	Open	https://shorturl.asia/yaNkf
VeriSeek [86]	DS-Coder	Open	https://huggingface.co/LLM-EDA/VeriSeek
VeriThoughts [87]	QC	Open	https://shorturl.asia/81f2a

CL: CodeLlama, DS: DeepSeek, CQ: CodeQwen, QC: Qwen2.5-Coder

4.2.1. Foundation Model

Foundation model choices reveal a clear preference for code-specialized bases. In total, 82.4% (28/34) of IT LLMs adopt coding-oriented foundations, with three clusters standing out:

- **DeepSeek-Coder ecosystem:** 32.4% market share (11 models) across variants (base, v2, R1-Distill), reflecting confidence in its code comprehension architecture
- **Qwen coding series:** 26.5% adoption (9 models) combining CodeQwen and Qwen2.5-Coder, demonstrating rapid integration of Alibaba's latest developments
- **CodeLlama variants:** 23.5% usage (8 models), maintaining strong presence despite newer alternatives

The remaining 17.6% (6/34) of models (6/34) build upon general-purpose foundations, with Llama family leading adoption (4 models) and compact models (GPT-2, Phi-1.5B) demonstrating efficiency approaches. Notably, 20.6% of projects (7 models) experiment across multiple foundation models, indicating systematic evaluation approaches rather than single-model commitment.

4.2.2. Naming Patterns

Naming conventions further highlight research intent:

- **RTL-Centric Approaches:** 5 models (14.7%) explicitly target Register Transfer Level abstraction (CraftRTL [58], DeepRTL series [59,60], RTL++ [66], RTLCoder [80], OpenRTLSet [64])
- **Verilog-Optimized Systems:** 7 models (20.6%) employ "Veri-" prefixing (VeriCoder [81], VeriGen [82], VeriLogos [83], VeriPrefer [84], VeriReason [85], VeriSeek [86], VeriThoughts [87]), emphasizing language-specific optimization
- **Reasoning-Enhanced Models:** 4 models (11.8%) explicitly incorporate advanced reasoning (CodeV-R1 [72], ReasoningV [79], VeriReason [85], VeriThoughts [87]), reflecting emerging focus on complex logical inference

 doi:10.20944/preprints202511.0656.v1

4.2.3. Trends Analysis

The current trajectory of IT LLMs for Verilog code generation exhibits three salient tendencies. First, development is consolidating around code-specialized foundations (e.g., DeepSeek-Coder, Qwen coding series, CodeLlama), alongside a sustained shift toward open weights. This configuration improves reproducibility and enables rigorous comparisons across studies.

In addition, there is a pronounced move toward explicit reasoning. Models that foreground multi-step inference (e.g., "R1-style" or "Reasoning/Thoughts" variants) aim to stabilize long-horizon logical chains and hierarchical design workflows, which better aligns with testbench-driven verification. Notwithstanding this emphasis, multi-base experimentation remains common, indicating that no single foundation dominates across all RTL tasks, datasets, and tool-chain settings.

4.3. Model Selection Insights

While our statistical analysis provides a comprehensive landscape of LLM adoption in Verilog code generation, translating these findings into actionable guidance for model selection represents a critical contribution to the different research scenarios (e.g., resource constraints, performance requirements, and research contexts).

For resource-constrained academic environments, open-source foundation models such as DeepSeek-Coder, Llama3.1, and Qwen2.5-Coder provide the optimal balance between performance, cost-effectiveness, and reproducibility. These models offer competitive Verilog generation capabilities while enabling local deployment and fine-tuning customization, though they require substantial computational infrastructure.

Performance-critical applications achieve best results with state-of-the-art closed-source models, particularly GPT variants and Claude variants. Despite higher operational costs and API dependencies, these models deliver superior generation quality and advanced reasoning capabilities essential for complex hardware design tasks. The sustained adoption of GPT-3.5 alongside newer models demonstrates that cost-performance considerations remain important across research contexts.

For domain-specific Verilog research, we recommend prioritizing the 19 available open-weight IT LLMs as foundational starting points. Among these, reasoning-enhanced variants (e.g., CodeV-R1 [72], ReasoningV [79], VeriReason [85], VeriThoughts [87]) merit particular attention, as their advanced reasoning capabilities address the increasingly complex logical inference requirements of modern hardware design challenges.

Answer to RQ1: We classify LLMs into two types: Base LLMs and IT LLMs, where base LLMs include both open-source models (204 mentions, led by Llama, DeepSeek, and Qwen series) and closed-source models (179 mentions, dominated by GPT variants). Additionally, 34 domain-specific IT LLMs have been developed, with 19 providing open weights and the majority built upon code-specialized foundations like DeepSeek-Coder and Qwen2.5-Coder.

5. RQ2: Dataset and Evaluation Metrics for Verilog Code Generation

5.1. Dataset Construction

We categorize datasets for Verilog code generation into two complementary families: (1) *Benchmark datasets* for standardized evaluation, and (2) *Instruct-tuning datasets* for supervised fine-tuning and preference/reasoning augmentation.

5.1.1. Overview

Tables 8 and 9 provide a comprehensive overview of the Verilog code generation dataset landscape. In terms of benchmarking, our analysis reveals 18 open-source benchmarks compared to 9 closed-source alternatives (2023–2025), reflecting the research community's commitment to reproducibility and transparent comparative evaluation. Similarly, in the instruction-tuning domain, open datasets

 doi:10.20944/preprints202511.0656.v1

(22 entries) also outnumber their closed-source items (12 entries), enabling standardized fine-tuning pipelines and consistent cross-method evaluation.

Across these datasets, we identify four design factors: input modality (Text vs. Text+Image vs. Text+Code), dataset availability (Open vs. Closed), testbench coverage (functional executability), and scale (ranging from tens to hundreds of thousands of samples). Although some researchers [61,88] have extracted raw Verilog corpora for pre-training[3,4], the field is mainly supported by carefully selected benchmarks for evaluation and structured instruction-style pairs for model training.

The following sections examine in detail the dataset construction methodologies and their associated quality assurance processes.

Table 8. Benchmark Datasets for Verilog Code Generation (Classified by Availability).

Dataset	Year	Input	Samples	Testbench	Available	URL
Closed-Source						
DAVE_test [9]	2020	Text	250	No	Closed	-
LLM-aided [89]	2024	Text	10	Yes	Closed	-
MCTS [90]	2024	Text	15	Yes	Closed	-
NL2Verilog [91]	2024	Text	8	Yes	Closed	-
VGV [92]	2024	Text+Image	20	Yes	Closed	-
AutoSilicon_test [93]	2025	Text	9	Yes	Closed	-
GEMMV_test [73]	2025	Text	10	Yes	Closed	-
HiVeGen_test [94]	2025	Text	4	Yes	Closed	-
ModelEval_test [95]	2025	Text	94	Yes	Closed	-
Open-Source						
ChipGPT [70]	2023	Text	8	Yes	Open	https://zenodo.org/records/7953725
VeriGen_test [88]	2023	Text	17	Yes	Open	https://github.com/shailja-thakur/VGen
VerilogEval-v1 [57]	2023	Text	156/143	Yes	Open	https://github.com/NVlabs/verilog-eval/tree/release/1.0.0
AutoChip [96]	2024	Text	138	Yes	Open	https://zenodo.org/records/10160723
ChipGPTV [97]	2024	Text+Image	30	Yes	Open	https://github.com/aichipdesign/chipgptv
CreativEval [98]	2024	Text	120	Yes	Open	https://github.com/matthewdelorenzo/CreativEval
Evaluatie_LLMs [99]	2024	Text	8	Yes	Open	https://zenodo.org/records/10947127
Hierarchical [100]	2024	Text	10	Yes	Open	https://github.com/ajn313/ROME-LLM
RTLLM-v1 [101]	2024	Text	30	Yes	Open	https://github.com/hkust-zhiyao/RTLLM/tree/v1.1
RTLLM-v2 [21]	2024	Text	50	Yes	Open	https://github.com/hkust-zhiyao/RTLLM/tree/main
RTLRepo_test [102]	2024	Text+Code	1.17k	No	Open	https://huggingface.co/datasets/ahmedallam/RTL-Repo
ArchXBench [103]	2025	Text	51	Yes	Open	https://github.com/sureshpurini/ArchXBench
CVDP [104]	2025	Text	783	Yes	Open	https://github.com/NVlabs/cvdp_benchmark
GenBen [105]	2025	Text+Image	324	Yes	Open	https://github.com/ChatDesignVerification/GenBen
RealBench [106]	2025	Text+Image	60	Yes	Open	https://github.com/IPRC-DIP/RealBench
ResBench [107]	2025	Text	56	Yes	Open	https://github.com/jultrishyyy/ResBench
VerilogEval-v2 [108]	2025	Text	156	Yes	Open	https://github.com/NVlabs/verilog-eval/tree/main/
VeriThoughts [87]	2025	Text	291	No	Open	https://huggingface.co/datasets/wilyub/VeriThoughtsBenchmark

[3] https://huggingface.co/datasets/SETH-TAMU/FreeSet-V1.0-LabUse
[4] https://huggingface.co/datasets/LLM-EDA/vgen_cpp

 doi:10.20944/preprints202511.0656.v1

Table 9. Instruct-Tuning Datasets for Verilog Code Generation (Classified by Availability).

Dataset	Year	Input	Samples	Testbench	Available	URL
Closed-Source						
DAVE_train [9]	2020	Text	5k	No	Closed	-
VerilogEval_train [57]	2023	Text	8.5k	No	Closed	-
BetterV [56]	2024	Text	Unknown	No	Closed	-
MEV-LLM [63]	2024	Text	31k	No	Closed	-
CodeV [71]	2025	Text	165k	No	Closed	-
CraftRTL [58]	2025	Text	80K	No	Closed	-
DeepRTL [59]	2025	Text	556k	No	Closed	-
DeepRTL2 [60]	2025	Text	433k	No	Closed	-
GEMMV_train [73]	2025	Text	12k	No	Closed	-
OpenRTLSet [64]	2025	Text	98K	No	Closed	-
ScaleRTL [68]	2025	Text	62k	No	Closed	-
VerilogDB [109]	2025	Text	20k	No	Closed	-
Open-Source						
VeriGen_train [88]	2023	Text	109k	No	Open	https://huggingface.co/datasets/shailja/Verilog_GitHub
AutoVCoder-Data [55]	2024	Text	1M	No	Open	https://github.com/sjtu-zhao-lab/AutoVCoder/tree/main/data
ChipGPT-FT-Data [110]	2024	Text	124k	No	Open	https://modelscope.cn/datasets/changkaiyan/chipgptseries
MG-Verilog-Data [76]	2024	Text	11k	No	Open	https://huggingface.co/datasets/GaTech-EIC/MG-Verilog
Origen-Data [78]	2024	Text	222k	No	Open	https://huggingface.co/datasets/henryen/origen_dataset_instruction
RTLCoder-Data [80,111]	2024	Text	27K	No	Open	https://github.com/hkust-zhiyao/RTL-Coder/tree/main/dataset
RTLRepo_train [102]	2024	Text+Code	2.92k	No	Open	https://huggingface.co/datasets/ahmedallam/RTL-Repo
Verilog-dateset [77]	2024	Text	68k	No	Open	https://huggingface.co/datasets/emilgoh/verilog-dataset-v2
CodeV-R1-Data [72]	2025	Text	3k	No	Open	https://huggingface.co/datasets/zhuyaoyu/CodeV-R1-dataset
DeepCircuitX [112]	2025	Text	28k	No	Open	https://drive.google.com/file/d/1Y002eJQPMbrEX7IpmzXDpFRlGl0XgUFu
Haven-KL [75]	2025	Text	62k	No	Open	https://huggingface.co/datasets/yangyiyao/HaVen-KL-Dataset
OpenCores [86]	2025	Text	834	No	Open	https://huggingface.co/datasets/LLM-EDA/opencores
PyraNet [65]	2025	Text	692k	No	Open	https://huggingface.co/datasets/bnadimi/PyraNet-Verilog
ReasoningV-Data [79]	2025	Text	5k	No	Open	https://huggingface.co/datasets/GipAI/ReaoningV
RTL++-Data [66]	2025	Text+Graph	11.7k	No	Open	https://huggingface.co/datasets/makyash/RTL-PP
VeriCoder-Origen [81]	2025	Text	126k	Yes	Open	https://huggingface.co/datasets/LLM4Code/expanded_origen_126k
VeriCoder-RTLCoder [81]	2025	Text	12k	Yes	Open	https://huggingface.co/datasets/LLM4Code/expanded_rtlcoder_12k
VeriLogos-Data [83]	2025	Text	10k	No	Open	https://huggingface.co/datasets/97kjmin/VeriLogos_Augmented_Dataset
VeriPrefer-Data [84]	2025	Text	90k	No	Open	https://huggingface.co/datasets/LLM-EDA/pyra
VeriReason-Data [85]	2025	Text	1k	Yes	Open	https://huggingface.co/Nellyw888/datasets
VeriThoughts-Data [87]	2025	Text	20k	No	Open	https://huggingface.co/datasets/wilyub/VeriThoughtsTrainSet

5.1.2. Benchmark Dataset Construction

(1) Data Sources

Benchmark datasets for Verilog code generation can be systematically categorized into four primary data sources based on their construction methodologies:

① **Template Synthesis.** Template synthesis utilizes systematic template construction to generate controlled datasets with parameterized descriptions and corresponding code templates.

For example, *DAVE* [9] first develops a structured template library containing paired English task templates and Verilog code templates with parameter placeholders, then populates these templates through systematic parameter conversion (e.g., mapping English "OR" to Verilog "|").

Although this approach ensures quality control through its structured generation process, it inherently lacks the natural diversity present in real-world implementations [113].

② **Open-Source Mining.** Open-source mining extracts code and problem statements from established educational platforms, public repositories, and community resources, leveraging verified solutions and comprehensive coverage.

VerilogEval [57] draws entirely from HDLBits [114], a Verilog learning platform, addressing multimodal elements like circuit diagrams and state graphs through two approaches: *VerilogEval-machine* uses GPT-3.5-turbo for automated description generation, creating 143 valid problems, while *VerilogEval-human* employs expert manual conversion to produce 156 high-fidelity problems. *VerilogEval-v2* [108] builds on this foundation with hybrid collections combining HDLBits resources and manual curation, targeting code completion and specification-to-RTL tasks with optimized formats and in-context learning examples.

AutoChip [96] similarly processes HDLBits data to develop executable benchmarks, refining 178 original problems to 120 code generation tasks and reconstructing testbenches by reverse engineering HDLBits' testing logic before standardizing design prompts. In contrast, *RTL-repo-test* [102] uses GitHub API to collect from public repositories, applying filters for permissive licenses, creation dates after October 2023, and repositories containing 4-24 Verilog files, then systematically selecting files and target lines for prediction tasks.

③ **Expert Curated.** Expert-curated datasets are developed by domain specialists who craft task specifications and reference implementations based on canonical designs, educational materials,

 doi:10.20944/preprints202511.0656.v1

and industrial best practices, typically featuring precise, unambiguous prompts and comprehensive testbenches.

Educational foundations appear in *VGV* [92] and *VeriGen_test* [88], which derive from course materials and HDLBits-style exercises, spanning from basic combinational logic to complex sequential circuits and finite state machines with graduated difficulty levels and expert-authored diagrams and testbenches, with VGV further incorporating visual prompt representations.

Standardized methodologies and broad module coverage characterize *RTLLM-v1/v2* [21,101] and *ChipGPTV* [97], which present comprehensive module taxonomies across arithmetic, logic/control, storage, and advanced categories. Complex hierarchical design challenges are addressed in *Model-Eval* [95], CreativEval [98], *Hierarchical* [100], and *ArchXBench* [103], which examine top-submodule structures, dependency graphs, timing constraints, and evaluation across diverse application domains.

Application-oriented benchmarks include *GEMMV* [73] and *CVDP* [104], with GEMMV offering 24 specialized tasks for AI accelerator cores evaluation, while CVDP provides comprehensive coverage across 13 task categories focused on design and verification scenarios, both emphasizing multi-scenario testing methodologies. *ResBench* [107] concentrates on real FPGA applications including machine learning acceleration, financial computing, and encryption, emphasizing resource optimization and application diversity. Specialized domain-specific accelerator evaluation is provided by *HiVeGen* [94], which employs LLM-based kernel analysis for application-driven parameter extraction, producing dataflow graphs and key attributes to guide design and verification processes.

④ **Hybrid Methods.** Hybrid methods integrate multiple data sources, typically starting with open-source mining and then applying expert modifications, LLM-based synthesis, or other enhancement techniques to improve data quality and coverage.

Open-source mining combined with expert curation is demonstrated in *AutoSilicon* [93], which extract code from open-source repositories while incorporating expert-written prompts. *AutoSilicon* targets industrial modules such as I/O controllers, processors, and accelerators, with expert-written specifications that follow RTLLM dataset description formats.

Multi-source integration is illustrated by *GenBen* [105], which combines silicon verification projects, textbooks, Stack Overflow discussions, and open-source hardware community resources through expert screening and organization into knowledge, design, debugging, and multimodal categories, featuring enhanced verification coverage and data perturbation techniques for contamination prevention and robustness improvement. Verified IP designs enhanced through expert optimization appear in *RealBench* [106], which draws from four verified open-source IP designs including OpenCores stable-stage AES encryption/decryption cores, SD card controllers, and open-source CPU cores such as Hummingbirdv2 E203, offering expert-written and optimized design specifications that ensure compliance with real engineering standards, complemented by 100% line coverage testbenches and formal verification processes for both module-level and system-level tasks.

LLM generation combined with human verification is exemplified by *VeriThoughts* [87], which integrates the MetRex dataset [115] containing 25.8K synthesizable Verilog designs with LLM-generated prompts, reasoning trajectories, and candidate Verilog code using models such as Gemini and DeepSeek-R1, followed by formal verification and human sampling verification to ensure quality and correctness.

(2) Quality Assurance

Quality control for benchmark datasets centers on three complementary approaches that ensure dataset reliability and correctness.

① **Expert Curation and Standardization.** Expert curation and standardization eliminate ambiguity and enhance reproducibility: benchmarks incorporate expert-authored specifications and reviews, alongside unified artifacts that ensure consistent interfaces and evaluation (i.e., natural language description, testbench, and reference RTL) with standardized prompt/module-header formats or structured, annotated diagrams [21,88,90,92,96,97,101,104,106].

 doi:10.20944/preprints202511.0656.v1

② **Testbench and Formal/Toolchain Checks.** Testbench and formal/toolchain checks verify functional correctness. Solutions range from reverse-engineered testbenches where platform logic remains undisclosed [96] to comprehensive coverage-driven or 100% line-coverage suites [105,106], hierarchical unit–integration testing [100], and multi-scenario/boundary testing with explicit timing constraints [95]. Formal methods address simulation limitations through equivalence checking and SAT-based proofs with self-consistency verification [87,106], while EDA pipelines deliver linting, synthesis, and PPA analysis to evaluate implementation viability (e.g., Verilator, Yosys, Design Compiler) [70,94].

③ **Contamination Control.** Contamination control maintains the fairness of evaluation. Decontamination procedures against external corpora, static/dynamic perturbations with expert verification, multi-stage quality checks with LLM-judge filters, and structured failure taxonomies with detailed logs protect against data leakage and noise [21,104–106].

5.1.3. Instruct-Tuning Dataset Construction

(1) Data Sources

Instruct-tuning datasets for Verilog code generation can be systematically categorized into three primary data source strategies based on their construction methodologies:

① **Open-Source Mining.** Similar to benchmark datasets, open-source mining in instruct-tuning datasets exclusively collect data from public repositories, educational platforms, and community resources without relying on LLM-generated augmentation.

VeriGen_train [88] demonstrates this methodology by integrating GitHub repository mining with educational resource extraction, processing 70 Verilog textbooks through OCR and text extraction to establish a comprehensive 400MB training corpus alongside GitHub-derived code modules. *VerilogDB* [109] and *RTL-repo* [102] expand this approach by incorporating multiple open-source channels including GitHub repositories, OpenCores hardware archives [116], and academic resources, achieving broad coverage through structured database organization and detailed metadata extraction.

Although this methodology guarantees data authenticity and legal compliance, it necessitates substantial preprocessing to address quality variations and lacks the semantic consistency provided by LLM-generated descriptions.

② **LLM Synthesis.** LLM synthesis approaches produce training data entirely through language model capabilities, primarily utilizing instruction-following techniques and structured synthetic data generation strategies.

RTLCoder [80,111] implements this methodology through methodical keyword pool development and instruction generation, employing GPT-3.5 to create over 27,000 instruction-code pairs from a carefully selected vocabulary of 350 hardware design keywords spanning more than 10 circuit categories. *CraftRTL* [58] utilizes multiple LLM-based synthesis techniques including Self-Instruct, OSS-Instruct, and Docu-Instruct to develop 80.1k synthetic samples, addressing challenges with non-textual representations through advanced prompt engineering and structured generation protocols. *ScaleRTL* [68] employs DeepSeek-R1 for extensive reasoning trajectory generation, producing 3.5 billion token Chain-of-Thought datasets through automated specification development and multi-step reasoning synthesis.

While this methodology provides substantial scalability and consistent generation quality, it may not fully capture the diversity and practical complexity inherent in human-authored code.

③ **Hybrid Methods.** Hybrid methods integrate open-source mining with LLM synthesis, typically starting with community-sourced code and then applying LLM-based enhancements for description generation, quality improvement, or data augmentation.

Open-source mining combined with LLM description generation is illustrated by *VerilogEval-train* [57], which extracts Verilog modules from GitHub repositories and uses GPT-3.5-turbo for automated problem description generation, creating 8,502 instruction-code pairs through systematic validation and quality filtering. *BetterV* [56] advances this approach by combining authentic GitHub-sourced Verilog data with LLM-synthesized virtual data, utilizing V2C translation tools and EDA

 doi:10.20944/preprints202511.0656.v1

verification to ensure functional correctness across all data sources. *MEV-LLM* [63] merges GitHub repository mining with ChatGPT-3.5-Turbo API for automated description generation, implementing tiered labeling strategies to create 31,104 classified samples across four complexity levels.

Multi-source integration with LLM enhancement is demonstrated by *AutoVCoder* [55], which collects 100,000 RTL modules from GitHub repositories while employing ChatGPT-3.5 for problem-code pair generation, implementing robust code filtering and correctness verification through Icarus Verilog and Python equivalence testing. *MG-Verilog* [76] combines repository mining with multi-granularity description generation using LLaMA2-70B-Chat and GPT-3.5-turbo, developing hierarchical descriptions from line-level comments to comprehensive summaries to support diverse learning objectives. *Origen* [78] implements code-to-code augmentation techniques, starting with open-source RTL libraries and leveraging Claude3-Haiku for description generation and code regeneration, with iterative compilation verification to guarantee functional correctness.

Expert knowledge integration with LLM synthesis is exemplified by *Haven* [75], which combines GitHub-sourced Verilog code with expert-curated examples from digital design textbooks and professional engineer annotations, utilizing GPT-3.5 for instruction optimization and logical enhancement to create knowledge-enriched and logic-enhanced datasets. *DeepRTL* [59] combines GitHub mining with proprietary industrial IP modules, employing GPT-4 for multi-granularity annotation through Chain-of-Thought methods while incorporating professional hardware engineer verification for industrial-grade accuracy, achieving 90% annotation precision through human evaluation validation.

(2) Quality Assurance

Quality control for instruct-tuning datasets converges on five complementary approaches that ensure training data reliability and optimize model performance.

① **Syntax and Synthesizability Verification.** Syntax and synthesizability verification confirms that training samples meet essential hardware design requirements through automated toolchain validation. *VeriGen_train* [88] utilizes Pyverilog for AST extraction and syntax validation, filtering modules to guarantee complete module-endmodule structures and proper port definitions. *AutoVCoder* [55] conducts comprehensive syntax checking via Icarus Verilog compilation, followed by Yosys synthesis verification to ensure code mappability to gate-level netlists, eliminating non-synthesizable constructs like `initial` blocks and `$display` statements. *VeriCoder* [81] extends this methodology by combining syntax validation with functional testing, using iverilog for compilation verification and automated testbench execution to ensure both syntactic correctness and behavioral validity. *DeepRTL* [59] performs CodeT5+ tokenizer compatibility checks, ensuring all modules fit within the 2048-token context window while preserving structural integrity through Pyverilog AST validation.

② **Functional Correctness Verification.** Functional correctness verification extends beyond syntax validation to confirm that generated code produces expected outputs through comprehensive testing and formal verification. *VeriCoder* [81] implements automated testbench generation using GPT-4o, creating unit tests with input stimuli and output assertions, then executing simulations through iverilog to verify functional equivalence between generated and reference implementations. *VeriThoughts* [87] applies formal equivalence checking using Yosys `miter` commands and SAT solvers to verify that LLM-generated code produces identical outputs to reference implementations across all input combinations. *ReasoningV* [79] integrates simulation-based testing with boundary case analysis, generating comprehensive test vectors covering edge cases, reset scenarios, and error conditions for robust functional validation. *VeriPrefer* [84] employs coverage-driven testing through VCS tools, achieving over 90% line coverage and comprehensive assertion validation for thorough correctness assessment.

③ **Contamination Control.** Contamination control safeguards dataset integrity by eliminating redundant samples and preventing data leakage from evaluation benchmarks. *PyraNet* [65] applies Jaccard similarity analysis on tokenized code sequences, removing samples with similarity scores above 0.9 to prevent model overfitting to redundant patterns. *CodeV* [71] implements MinHash-based

 doi:10.20944/preprints202511.0656.v1

deduplication with 128-dimensional vector mapping, calculating Jaccard similarity to filter duplicate files while maintaining dataset diversity. *VeriLogos* [83] addresses contamination through Rouge-L similarity analysis against public benchmarks (VerilogEval, RTLLM), removing samples with similarity scores exceeding 0.5 to prevent evaluation bias. *ScaleRTL* [68] utilizes 5-gram sequence analysis for contamination detection, comparing training samples against benchmark solutions to preserve evaluation integrity and prevent data leakage.

④ **Format Standardization and Length Control.** Format standardization and length control establish consistent data representation and ensure compatibility with model training requirements. *Haven* [75] implements comprehensive format normalization, standardizing signal naming conventions (e.g., `clk`, `rst_n`), module declaration structures, and indentation styles (4-space indentation) while removing redundant comments and preserving functional documentation. *OpenRTLSet* [64] applies CodeLlama tokenizer for length validation, truncating samples exceeding 8k tokens to prevent training context overflow while preserving code completeness. *VeriReason* [85] implements structured format control, standardizing specification descriptions (100-300 words), reasoning steps (150-300 words), and code segments (under 2048 tokens) to ensure consistent training input formatting. *RTL++* [66] extends format standardization to multimodal data, ensuring consistent CFG/DFG text representation and maintaining structured instruction-code-graph triplets within model context boundaries.

⑤ **Quality Filtering and Human Validation.** Quality filtering and human validation provide expert oversight to ensure annotation accuracy and domain-specific correctness. *PyraNet* [65] employs GPT-4o-mini for automated quality scoring (0-20 scale) based on coding style, efficiency, and standardization, with human validation by hardware engineers to verify score-quality consistency. *DeepRTL* [59] conducts multi-stage human validation, utilizing four professional Verilog designers for annotation accuracy assessment and three additional engineers for cross-validation, achieving 90% annotation accuracy through expert review. *VeriCoder* [81] combines automated filtering with human verification, engaging hardware engineers to validate specification-code consistency and testbench effectiveness, ensuring 92% functional correctness through expert evaluation. *Haven* [75] incorporates expert-curated examples from digital design textbooks and professional engineer annotations, implementing knowledge-enhanced and logic-enhanced dataset construction with domain expert validation for engineering practice compliance.

These quality assurance mechanisms collectively ensure that instruct-tuning datasets deliver reliable, diverse, and functionally correct training supervision, enabling effective model fine-tuning while preserving evaluation integrity and preventing common issues such as data leakage, annotation errors, and functional inconsistencies.

5.1.4. Trends Analysis

We analyze three key evolutionary aspects of datasets and their implications for evaluation methodology: (1) testbench availability as an indicator of executability, (2) input modality as benchmarks extend beyond text-only specifications, and (3) dataset scale as applications range from reasoning-focused micro-suites to comprehensive coverage collections. Based on these trends, we formulate actionable recommendations for future dataset development that better align training supervision with executable evaluation and downstream tasks.

(1) Testbench availability

Executable verification has become the standard approach for evaluation. Among open benchmarks, 16 of 18 include testbenches (with exceptions being *RTLRepo_test*, which focuses on line-level generation, and *VeriThoughts*, which employs formal equivalence via Yosys rather than simulation). In the closed-source category, all datasets except *DAVE_test* provide testbenches. By contrast, most instruct-tuning datasets lack testbenches; only select examples (e.g., *VeriCoder-Origen*, *VeriCoder-RTLCoder*, *VeriReason-Data*) incorporate executable verification. This disparity highlights the critical need for executable supervision in training protocols (including testbenches, formal specifications, and synthesis-driven feedback).

 doi:10.20944/preprints202511.0656.v1

(2) Input modality

While input modality remains primarily text-centric, benchmarks increasingly incorporate multimodal specifications (Text+Image), as evidenced in *ChipGPTV* [97], *VGV* [92], *GenBen* [105], and *RealBench* [106]. This evolution reflects practical design workflows where timing diagrams, block schematics, and waveform representations complement textual requirements. For training resources, instruction corpora remain predominantly text-only, with emerging interest in structural representations (exemplified by the Text+Graph approach in *RTL++-Data*) that reveal hierarchical organization and connectivity patterns essential for effective RTL reasoning.

(3) Dataset scale

Dataset dimensions diverge according to purpose. Benchmarks span from focused micro-scale collections (4–30 items) that emphasize logical reasoning and specification fidelity, to comprehensive mid/large-scale resources (e.g., *CVDP*: 783; *GenBen*: 324) that prioritize coverage and diversity. Instruction-tuning resources, conversely, range from specialized compact sets (e.g., *CodeV-R1-Data*) to extensive corpora (reaching hundreds of thousands to million-scale samples; e.g., *AutoVCoder-Data*, *PyraNet*). The former sharpen reasoning capabilities and alignment, while the latter facilitate broad generalization despite less rigorous executable validation. This complementary relationship between compact, high-signal benchmarks and extensive, weakly-validated training sets reflects broader methodological patterns in code-oriented language models.

Based on these observations, future dataset development should prioritize: (1) early integration of *executable supervision* (testbenches, formal specifications, synthesis signals) to reduce the gap between supervision and evaluation; (2) stronger *multimodal alignment* between training and evaluation (e.g., diagrams, specification sheets) as benchmarks incorporate visual artifacts; (3) *reasoning-enhanced training* that explicitly incorporates intermediate reasoning steps and design decision processes into fine-tuning datasets (as seen in recent efforts like CodeV-R1 [72], VeriReason [85], ScaleRTL [68], and VeriThoughts [87]) to develop models that can articulate design rationales and tackle complex hardware design problems with transparent, verifiable reasoning.

5.2. Evaluation Metrics

Assessing the quality of LLM-generated Verilog code poses substantial challenges, requiring thorough evaluation of both syntactic validity and semantic correctness. To address these assessment requirements systematically, researchers have developed diverse evaluation methodologies that can be organized into three fundamental categories:

5.2.1. Similarity-based Metrics

Similarity-based evaluation metrics assess generated Verilog code quality by comparing structural or textual similarity scores between generated code and reference implementations without executing the code.

(1) General-Purpose Metrics

The majority of evaluation approaches utilize established text similarity metrics adapted from natural language processing and general code evaluation frameworks, including:

- **Exact Match:** Evaluates complete correspondence between generated and reference code, providing binary correctness assessment [67,102]
- **Edit Distance Metrics:** Quantify minimal character-level modifications needed for alignment, including Levenshtein distance [67,102] and Ratcliff-Obershelp similarity [9]
- **N-gram Based Metrics:** Assess sequence overlap through established measures such as BLEU [60, 104,112,117], METEOR [60,112], ROUGE [60,112,117], and chrF [117] scores, originally developed for machine translation evaluation

 doi:10.20944/preprints202511.0656.v1

These metrics provide notable advantages including computational efficiency and straightforward implementation.

(2) Verilog-Specific Metrics

Traditional general-purpose metrics cannot adequately capture domain-specific semantics inherent in hardware description languages. To address this limitation, researchers developed SimEval [118], a specialized metric operating at three complementary levels: (1) *Syntactic analysis* using Abstract Syntax Tree (AST) extraction via PyVerilog for structural assessment; (2) *Semantic analysis* employing Control Flow Graph (CFG) analysis through Verilator to identify execution paths; and (3) *Functional analysis* utilizing gate-level netlist comparison via Yosys synthesis for implementation equivalence.

5.2.2. Execution-based Metrics

Execution-based evaluation metrics assess generated Verilog code quality through comprehensive compilation and runtime verification, providing assessment of functional correctness and hardware implementation efficiency.

(1) Correctness Assessment

The fundamental execution-based evaluation focuses on verifying code functionality through compilation and behavioral testing, including:

- **Syntax-pass@k:** Quantifies the proportion of generated code samples that successfully compile without syntax errors across k attempts
- **Functional-pass@k:** Determines behavioral correctness by executing generated modules against standardized testbenches with predefined expected outputs
- **Formal-Verification:** Establishes equivalence between generated code and reference implementations using formal methods via `"Yosys -equiv"`

(2) Hardware Quality Metrics

Beyond fundamental correctness, researchers increasingly evaluate hardware implementation quality through comprehensive PPA (Power, Performance, Area) analysis:

- **Power:** Measures energy efficiency of synthesized designs
- **Performance:** Analyzes timing characteristics including maximum operating frequency, critical path delays, and latency metrics
- **Area:** Quantifies resource utilization efficiency in terms of logic gates, flip-flops, memory elements, and overall silicon footprint

5.2.3. LLM-based Metrics

LLM-based evaluation metrics utilize the reasoning capabilities of language models to assess code quality, capturing semantic understanding and domain-specific knowledge beyond traditional similarity-based techniques.

(1) Model Confidence Metrics

These metrics analyze the internal confidence of language models during code generation:

- **Perplexity:** Quantifies model certainty in generated Verilog code [74,117]

(2) LLM-as-a-Judge

These methodologies employ language models as expert evaluators to assess multiple dimensions of code quality:

- **GPT Score:** Utilizes GPT-family models as evaluators to assess code quality across multiple dimensions including readability, correctness, and design best practices [59,60]

 doi:10.20944/preprints202511.0656.v1

- **VCD-RNK:** Evaluates semantic consistency between natural language specifications and generated code implementations, implementing knowledge distillation techniques to develop efficient lightweight models capable of assessing functional [119]
- **MetRex:** Utilizes language models as expert evaluators to predict the PPA of generated Verilog code [115]

5.2.4. Metrics Trends Analysis

As illustrated in Figure 7, evaluation methodologies from 2023 to 2025 demonstrate a clear evolution.

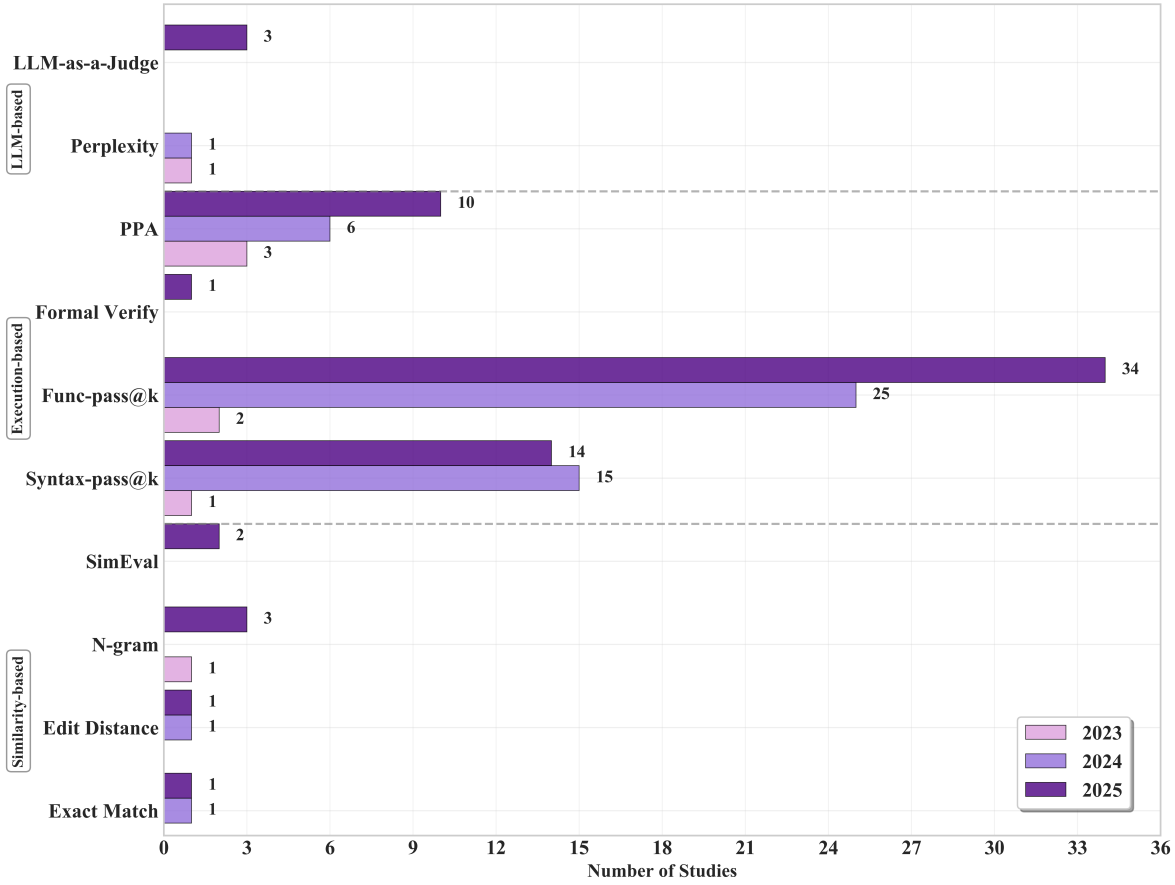


Figure 7. Usage of Evaluation Metrics Across Years (Counts from Surveyed Studies).

(1) Execution-based metrics

Execution-based metrics have become primary: *functional-pass@k* shows substantial growth from 2 instances (2023) to 25 (2024) and 34 (2025), while *syntax-pass@k* increases from 1 (2023) to 15 (2024), maintaining significant presence with 14 instances in 2025. Hardware quality assessment steadily gains importance, with *PPA* measurements increasing from 3 (2023) to 6 (2024) and 10 (2025), highlighting the research community's growing emphasis on implementation feasibility beyond basic functional correctness. Formal equivalence verification has only recently entered mainstream adoption (*formal verification*: 1 instance in 2025), largely facilitated by improvements in verification tool-chain maturity.

(2) Similarity-based metrics

Similarity-based measures continue to exist but increasingly serve supplementary functions. N-gram metrics display modest variation ($1 \rightarrow 0 \rightarrow 3$), while *Exact Match* and *Edit Distance* maintain minimal presence (0/0 in 2023, followed by 1/1 in both 2024 and 2025), highlighting their inherent limitations in capturing RTL semantic structures. By contrast, the Verilog-specific metric *SimEval* emerges in 2025 (2 instances), indicating growing recognition of the need for specialized metrics

 doi:10.20944/preprints202511.0656.v1

that effectively integrate syntax analysis, control/data flow evaluation, and functional equivalence verification.

(3) LLM-based metrics

LLM-based evaluation represents an emerging complementary approach. While use of *Perplexity* gradually diminishes ($1 \rightarrow 1 \rightarrow 0$), *LLM-as-a-Judge* appears in 2025 (3 instances), indicating a transition from basic confidence measures toward sophisticated, task-aware evaluations capable of assessing readability, specification alignment, and design best practices beyond the capabilities of traditional text similarity metrics.

Based on these observations, future metrics development should prioritize: (1) standardized benchmark integration with execution-based metrics to ensure consistent, reproducible evaluation across research efforts; (2) hardware-specific evaluation frameworks that directly measure downstream design objectives beyond functional correctness, such as PPA optimization; (3) LLM-as-a-Judge frameworks as a promising research direction, particularly for evaluating semantic correctness of generated Verilog code without reference implementations.

Answer to RQ2: (1) Verilog datasets are categorized into benchmarks and instruct-tuning collections. Construction methodologies include template synthesis, open-source mining, expert curation, LLM synthesis, and hybrid approaches, with quality assured through testbench verification, formal validation, contamination control, and format standardization. (2) Evaluation metrics have evolved from text similarity measures toward execution-based assessment, with functional-pass@k emerging as the predominant metric. Future research directions include hardware quality metrics (PPA) and LLM-as-a-Judge approaches for reference-free semantic evaluation of generated Verilog code.

6. RQ3: Adaptation and Optimization Techniques for Verilog Code Generation

Having established the foundational datasets and evaluation methodologies in the previous sections, we now examine the diverse adaptation and optimization techniques employed to enhance LLMs for Verilog code generation. These techniques can be broadly categorized into training-free and training-based approaches, each offering distinct advantages and trade-offs in terms of computational efficiency and performance gains.

6.1. Training-free Methods

Training-free methods provide immediate applicability without requiring additional model training, making them particularly attractive for practitioners with limited computational resources.

6.1.1. EDA-Tool Feedback

EDA-tool feedback represents a critical advancement in LLM-based Verilog generation, where EDA tools provide concrete error diagnostics and performance metrics to guide iterative code refinement. Unlike approaches relying solely on LLM self-correction or testbench outputs, EDA-tool feedback leverages industry-standard verification and synthesis tools to capture syntax errors, functional mismatches, timing violations, and PPA metrics. This feedback mechanism creates a closed-loop optimization system where tool-generated diagnostics inform subsequent LLM refinement iterations, progressively improving code quality from syntactic correctness to physical implementation feasibility.

Based on the architectural design of the feedback loop, we categorize these approaches into single-agent systems and multi-agent systems.

(1) Single-Agent Systems

Single-agent systems employ a unified LLM instance that processes EDA tool feedback and iteratively refines Verilog code through self-correction mechanisms, integrating multiple feedback sources into cohesive prompt engineering strategies.

 doi:10.20944/preprints202511.0656.v1

Several approaches implement hierarchical optimization stages with progressive feedback integration. VeriPPA [133] introduces dual-stage optimization combining iverilog simulator feedback for syntax/functional correctness ($V(n+1) = V(n) - E(V(n))$) with Synopsys Design Compiler PPA reports for power-performance-area optimization through VeriRectify module exploration of pipelining, clock gating, and parallel operations. The LLM-Powered RTL Assistant [138] implements automatic prompt engineering mimicking human designer workflows through three prompt types: initial prompts establishing "professional Verilog designer" roles with step-by-step planning, self-verification prompts guiding testbench generation and behavioral analysis, and self-correction prompts integrating Icarus Verilog error logs.

RTLFixer [139] specifically targets syntax errors through ReAct prompting with "think-act-observe" iterative cycles combining compiler feedback and RAG-based human expert knowledge retrieval. Paradigm-Based HDL Generation [137] emulates human expert design through structured task decomposition using three paradigm blocks: COMB blocks extract truth tables simplified by PyEDA to Sum-of-Products expressions; SEQU blocks construct state transition tables guiding "three-always-block" generation; BEHAV blocks decompose complex tasks into behavioral components.

EvoVerilog [140] integrates LLM reasoning with evolutionary algorithms through thought tree initialization and four specialized offspring operators for design space exploration. Non-dominated sorting implements multi-objective optimization balancing testbench mismatch minimization and hardware resource reduction, identifying Pareto-optimal solutions across iterations. VGV [92] leverages MMLLMs to generate Verilog from circuit diagrams through two visual prompting strategies: Basic Visual (BV) mode for direct diagram-to-code conversion and Thinking Visual (TV) mode requiring two-stage component description before generation.

These single-agent approaches demonstrate significant advancements in automated Verilog generation through systematic integration of EDA tool feedback, achieving substantial improvements in code quality, functional correctness, and design optimization.

(2) Multi-Agent Systems

Multi-agent systems employ multiple specialized LLM instances that collaborate through structured communication protocols and role-based task distribution, leveraging collective intelligence to address complex Verilog generation challenges through coordinated EDA tool feedback processing.

Several frameworks implement basic two-agent architectures with complementary roles. AIVRIL [142] establishes a dual-loop optimization system with Code Agent generating RTL code and testbenches while Review Agent analyzes compilation logs for structured correction feedback, achieving AutoReview (syntax verification) and AutoDV (functional verification) cycles with tool/LLM agnostic design supporting Claude 3.5 Sonnet, GPT-4o, and Llama3 70B. The LLM-aided Front-End Framework [89] deploys three task-specific LLMs in sequential collaboration: $h_\theta^{(1)}$ generates RTL from specifications, $h_\theta^{(2)}$ constructs corresponding testbenches, and $h_\theta^{(3)}$ performs design review based on behavioral simulation results. RTL Agent [143] implements Generator-Reviewer architecture inspired by Reflexion technology, improving the performance of GPT-4o-mini/GPT-4o through iterative "generation-test-reflection" cycles with cost-performance trade-off analysis favoring parallel width over iterative depth.

Advanced frameworks employ domain-specific agent specialization with sophisticated coordination mechanisms. MAGE [144] introduces four-agent collaboration: Testbench Generator optimizing text waveform output, RTL Generator with syntax checking, Judge Agent for simulation evaluation and scoring, and Debug Agent for iterative refinement, employing high-temperature sampling with Top-K candidate selection and normalized mismatch scoring for up to 5 syntax error iterations. VerilogCoder [145] integrates Task Planning with Task-Circuit Relationship Graph (TCRG) decomposition, Code Agent with Verification Assistant for syntax consistency, and Debug Agent utilizing three EDA tool types: Icarus Verilog syntax checker, Verilog simulator for VCD generation, and novel AST-based waveform tracing tool through Pyverilog for signal backward tracing and mismatch localization.

 doi:10.20944/preprints202511.0656.v1

CoopetitiveV [146] implements competitive-cooperative dynamics through four agents: Code Generator for initial generation, Research Agent for error analysis and correction strategies, Prosecutor Agent challenging Research strategies to refine solutions, and dual Revision Agents (code/testbench) executing optimized corrections, with Icarus Verilog and AutoBench integration supporting up to 2 iteration rounds.

Sophisticated frameworks address scalability and optimization through hierarchical organization and automated workflow discovery. Nexus [147] employs layered supervisor architecture with Root Supervisor for task decomposition, Task Supervisors for subtask management, and EDA Agents for Xilinx Vivado CLI interaction, utilizing YAML-configurable prompts and report-driven optimization targeting timing convergence through WNS/TNS analysis, resource utilization (LUT/FF/BRAM), and power consumption metrics. RTLSquad [134] implements three-stage Agent Squad with Exploration Squad (Power/Area/Performance agents debating through voting mechanisms), Implementation Squad (Programmer and Reviewer managing task checklists), and Verification Squad (Observer extracting EDA reports, Analyzer scoring with 1-5 scale evaluation), providing decision pathway prompts for interpretability and employing score-driven exploration point adjustment. VFlow [148] pioneers automated workflow optimization through Cooperative Evolutionary Population-based MCTS (CEPE-MCTS) maintaining four specialized populations (functional correctness, area optimization, timing optimization, balanced multi-objective), implementing fragment migration mechanisms and global failure experience sharing with time decay factors, supported by hardware-domain verification operators (syntax, simulation, waveform analysis, circuit optimization, hierarchical composition).

Recent frameworks focus on power-performance-area optimization and comprehensive evaluation methodologies. VeriOpt [131] establishes four-role LLM collaboration (Planner for requirement decomposition, Programmer for annotated code generation, Reviewer for step execution verification, Evaluator for EDA feedback analysis) with PPA-aware in-context learning covering power optimization (clock/power gating), performance enhancement (pipelining, retiming), and area reduction (resource sharing, FSM encoding), employing dual-stage iteration (functional correctness followed by Synopsys Design Compiler PPA optimization). AutoSilicon [93] scales to complex RTL designs through three-agent architecture: Designer Agent with task decomposition and multi-version voting, Memory Debugger Agent with persistent debugging experience and RAG-like semantic retrieval for waveform sequence queries, and Task Scheduler for dynamic dependency management, integrating Icarus Verilog for syntax/functional verification and Synopsys Design Compiler with ASAP 7nm PDK for PPA assessment. VeriMind [149] introduces comprehensive five-agent collaboration (Supervisor orchestration, Prompt Engineer for requirement optimization, Verilog Code Generator, Testbench Generator, Checker Agent for EDA tool integration) with novel pass@ARC evaluation metric combining traditional pass@k with Average Round Count through exponential penalty terms.

These multi-agent approaches demonstrate substantial advances in collaborative intelligence for Verilog generation, with architectures ranging from simple dual-agent coordination to sophisticated hierarchical workflows with specialized domain expertise.

6.1.2. Prompt Engineering

Prompt engineering approaches enhance model performance through comprehensive prompting strategies or knowledge retrieval mechanisms without requiring EDA tool feedback or model fine-tuning. These methods leverage structured/hierarchical prompting and retrieval-augmented generation to improve code quality and generation efficiency.

For the comprehensive prompting strategies, ChatModel [95] implements Hierarchical Agile Modeling flow to generate Verilog code, where specification agents convert natural language to Module Intermediate Representations and Design Architecture Graph, while modeling agents generate code with RAG-enhanced error correction. HiVeGen [94] employs a three-component framework: (1) Design Space Explorer decomposes complex designs into sub-modules with PPA-aware optimization, (2) weighted Code Library maintains dynamic module quality scores using $Code(T) = Code(\arg\max_{T_{db} \in DB}(\cos(T, T_{db}) \cdot w(T_{db})))$ where weights update as success ×1.06 and

 doi:10.20944/preprints202511.0656.v1

failure ×0.9, and (3) Runtime Parser enables real-time structural correction through natural language feedback. AoT [120] presents three-stage prompting: (1) hardware design pattern classification into combinational vs. sequential logic with optimal representation selection, (2) structured JSON-format Intermediate Representation generation decoupling circuit functionality from syntax, and (3) line-by-line pseudocode generation bridging Intermediate Representation to Verilog implementation.

For the knowledge retrieval mechanisms, VeriRAG [121] specializes in AI co-processor design through Summary-Template RAG addressing semantic fragmentation by extracting natural language summaries before code retrieval, and domain-customized prompts with explicit hardware constraints. HDLCoRe [122] combines task classification into four categories based on logic type and complexity: SC-HDL (Simple Combinational, e.g., 8-bit adder), CC-HDL (Complex Combinational, e.g., 16-bit divider), SS-HDL (Simple Sequential, e.g., basic counter), CS-HDL (Complex Sequential, e.g., pipelined multiplier), with differentiated prompting strategies, self-verification through automatic testbench generation and step-by-step simulation debugging, and two-stage heterogeneous RAG extracting three core components (high-level overview, low-level details, module ports) followed by coarse filtering and fine reranking using cross-encoders.

These approaches demonstrate performance improvements without computational overhead, with hierarchical methods suited for scalable designs, multi-stage abstraction for complex reasoning, and retrieval augmentation for knowledge-intensive scenarios.

6.1.3. Inference Optimization

Decoding optimization approaches enhance Verilog code generation quality through advanced decoding algorithms, self-consistency principles, and lightweight reranking frameworks to improve the selection of optimal Verilog code from multiple candidates.

DecoRTL [123] addresses the challenge of balancing syntax correctness and design diversity through a hybrid randomness-deterministic decoding framework. The approach employs syntax-aware temperature scaling that dynamically adjusts sampling temperature based on token types: structural tokens use $T_{base} - 0.1$ for enhanced determinism, high-impact tokens use $T_{base} + 0.1$ for controlled randomness, and other tokens maintain T_{base}. The contrastive self-consistency reranking mechanism computes adjusted scores as $score(x_{(i)}) = L_i - \lambda \cdot sim(e_i, \bar{e})$ where L_i is the original log probability and $\lambda = 0.5$ is the penalty coefficient for semantic redundancy.

VRank [124] implements a three-stage reranking framework based on self-consistency principles for deterministic optimal Verilog code selection. The approach employs execution-based clustering where candidates with identical simulation outputs across all test cases are grouped together, followed by cluster ranking using strict consistency loss $\ell_{strict}(c, c') = \max_{t \in T} 1[c(t) \neq c'(t)]$ and cluster scoring $R(c) = n - \sum_{c' \in \mathcal{C}} \ell_{strict}(c, c')$.

VCD-RNK [119] presents a distillation reranking framework that formalizes Verilog code selection as semantic alignment optimization: $\hat{y}^* = \arg\max_{\hat{y}_i \in Y_k} F_\phi(x, \hat{y}_i)$ where F_ϕ is a discriminator trained on VerilogJudge-47K dataset through dual-teacher collaborative distillation. The lightweight student model undergoes three-stage reranking: syntax pre-filtering, discriminator evaluation, and majority voting decision with score $F_\phi(x, \hat{y}_i) = \frac{\sum_{j=1}^{m} \mathbb{I}[F_\phi^{(j)}(x, \hat{y}_i) = 1]}{m}$.

These decoding optimization approaches demonstrate significant practical value while maintaining computational efficiency. The choice depends on specific requirements: syntax-aware decoding for balanced correctness-diversity trade-offs, self-consistency methods for deterministic selection from multiple candidates, and distillation-based approaches for efficient large-scale deployment scenarios.

6.2. Training-based Methods

While training-free methods offer convenience, training-based approaches typically achieve superior performance by adapting model parameters to the specific characteristics of Verilog code generation. These methods range from comprehensive pre-training to targeted fine-tuning strategies, each addressing different aspects of domain adaptation.

 doi:10.20944/preprints202511.0656.v1

6.2.1. Pre-training

Pre-training constitutes a foundational paradigm for adapting LLMs to Verilog code generation through domain-specific training methodologies. This approach optimizes the autoregressive language modeling objective:

$$\mathcal{L}_{LM} = -\sum_{i=1}^{T} \log P(x_i | x_{<i}; \theta) \tag{1}$$

where x_i denotes the i-th token, $x_{<i}$ represents the preceding context, θ encompasses model parameters, and T indicates sequence length. Two primary pre-training strategies have emerged in recent work:

(1) Training from Scratch:

Hardware Phi-1.5B [74] demonstrates training a 1.5B parameter Transformer model specifically for hardware code generation. The model employs 24 layers with 32 attention heads and incorporates Flash Attention 2 for efficiency. Training utilizes mixed precision with FSDP for distributed computing, achieving convergence after 750K iterations with careful hyperparameter tuning (learning rate 2e-4, Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$).

(2) Continued Pre-training:

FreeV [71] adopts a more resource-efficient approach by continuing pre-training of Llama 3.1 8B Instruct on Verilog-specific data. The method employs 4-bit quantization with LoRA adaptation (rank=8) to reduce computational requirements while maintaining performance. Training for one epoch with batch size 16 and gradient accumulation demonstrates effective domain adaptation.

Both approaches show that domain-specific pre-training significantly improves Verilog code generation capabilities, with the choice between training from scratch versus continued pre-training depending on available computational resources and target model size requirements.

6.2.2. Supervised Fine-tuning

Beyond pre-training, supervised fine-tuning represents a more targeted approach to model adaptation, focusing on specific downstream tasks and performance metrics. Supervised fine-tuning optimizes model parameters to minimize the cross-entropy loss on Verilog code generation task-specific datasets:

$$\mathcal{L}_{SFT} = -\sum_{(x,y)\in\mathcal{D}} \sum_{i=1}^{|y|} \log P(y_i | x, y_{<i}; \theta) \tag{2}$$

where $\mathcal{D}$ represents the fine-tuning dataset containing natural language specification-Verilog code pairs, x denotes the natural language specification, y is the target Verilog code sequence, and θ encompasses the model parameters to be optimized. Three primary fine-tuning strategies have emerged for computational efficiency and performance optimization for Verilog code generation:

Full Parameter Fine-tuning. This approach updates all model parameters θ through standard gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}_{SFT}(\theta_t) \tag{3}$$

where η represents the learning rate. While achieving optimal performance, this method requires substantial computational resources proportional to the full model size.

Low-Rank Adaptation (LoRA). LoRA introduces trainable low-rank matrices to approximate parameter updates, significantly reducing computational overhead. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA decomposes the update as:

$$W = W_0 + \Delta W = W_0 + BA \tag{4}$$

 doi:10.20944/preprints202511.0656.v1

where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and $r \ll \min(d, k)$ is the rank. The fine-tuning objective becomes:

$$\mathcal{L}_{LoRA} = -\sum_{(x,y)\in\mathcal{D}} \sum_{i=1}^{|y|} \log P(y_i | x, y_{<i}; W_0 + BA) \tag{5}$$

Quantized Low-Rank Adaptation (QLoRA). QLoRA extends LoRA by quantizing the base model to 4-bit precision while maintaining LoRA adapters in full precision. The quantized weight matrix is represented as:

$$W_{quantized} = \text{dequantize}(\text{quantize}(W_0, \text{4-bit})) + BA \tag{6}$$

This approach achieves memory efficiency comparable to 4-bit quantization while preserving the adaptation capabilities of LoRA, enabling fine-tuning of large models on resource-constrained hardware.

(1) Data-Centric Tuning

Data-centric tuning approaches improve model performance primarily through innovative dataset construction while employing standard full-parameter SFT or LoRA for training. These methods focus on enhancing training data quality, diversity, and coverage rather than modifying model architectures or training algorithms. As detailed in RQ2, dataset construction methodologies span template synthesis (DAVE [9]), open-source mining (e.g., VeriGen [88]), LLM synthesis (e.g., RTLCoder [80]), and hybrid approaches combining multiple data sources with LLM enhancement (e.g., VerilogEval [57], AutoVCoder [55], MG-Verilog [76]). Quality assurance mechanisms include syntax verification, functional correctness testing, contamination control, format standardization, and expert validation to ensure reliable training supervision.

Representative works demonstrate consistent effectiveness through full-parameter fine-tuning on carefully curated datasets. Pearce et al. [9] fine-tunes GPT-2 on template-generated data achieving 94.8% accuracy, while Thakur et al. [88] processes GitHub and textbook sources for comprehensive coverage. VerilogEval [57] employs GPT-3.5-turbo for automated description generation, and Dehaerne et al. [117] demonstrate transfer learning from general programming languages to Verilog code generation through fine-tuning. Thakur et al. [82] extends with enhanced evaluation, while OpenLLM-RTL [21] provides expanded benchmarks and datasets. Goh et al. [77] include English-to-ASIC with 68k labeled samples, CraftRTL [58] implements "correct-by-construction" generation, OpenRTLSet [64] offers 127k diverse samples, and Veritas [69] utilizes CNF-guided synthesis.

Parameter-efficient tuning approaches employ LoRA to reduce computational costs while maintaining competitive performance. VeriCoder [81] fine-tunes Qwen2.5-14B using LoRA on functionally verified datasets, while GEMMV [73] applies LoRA to GEMM-specific generation with performance awareness.

These works collectively demonstrate that both high-quality dataset curation with standard SFT and parameter-efficient approaches using LoRA can achieve competitive performance, with the latter providing substantial improvements while significantly reducing computational requirements compared to full-parameter fine-tuning.

(2) Strategy-Centric Tuning

Strategy-centric tuning approaches enhance model performance through innovative training methodologies, which employ complex training paradigms such as curriculum learning, multi-expert architectures, and adaptive sampling strategies to optimize the learning process for Verilog code generation.

MEV-LLM approach [63] introduces complexity-based specialization through full-parameter fine-tuning. The method creates specialized expert models for different complexity categories (basic, intermediate, advanced, expert-level) and trains a separate complexity classifier for inference-time routing. Similarly, MG-Verilog [76] addresses complexity diversity through multi-granularity mixed sampling during QLoRA fine-tuning, randomly sampling from different description levels (high-level

 doi:10.20944/preprints202511.0656.v1

summaries, detailed specifications, block-level descriptions, line-by-line annotations) within each training iteration to enable balanced knowledge transfer across semantic granularities.

Multiple works adopt curriculum learning with different progression strategies. AutoVCoder [55] implements two-stage LoRA fine-tuning: first on high-quality GitHub data to establish fundamental syntax, then on synthetic datasets for task-specific enhancement. DeepRTL [59] extends this to three-dimensional curriculum learning using full-parameter fine-tuning: structural progression (*line→block→module*), description granularity progression (*detailed→high-level*), and annotation quality progression (*GPT-generated→human expert*). PyraNet [65] combines curriculum learning with hierarchical loss weighting, stratifying data into six quality layers with differential weights (1.0 to 0.1) and following "high-quality-first, simple-to-complex" training order using LoRA fine-tuning.

The choice of strategy depends on specific requirements: multi-expert architectures for handling diverse complexity levels, mixed sampling for multi-granularity adaptability, progressive curriculum for systematic knowledge acquisition, and hierarchical training with loss weighting for quality-aware optimization.

(3) Multi-task Tuning

Multi-task tuning approaches enhance model capabilities by simultaneously optimizing multiple related objectives, enabling models to handle diverse aspects of Verilog code generation within a unified framework. These methods leverage shared representations across tasks while employing specialized loss functions and architectural components to address specific requirements.

Several works adopt dual-task architectures to enhance both code creation and quality control. BetterV [56] employs a generator-discriminator framework using multi-task LoRA, where the generator handles basic code generation while the discriminator provides EDA-specific guidance. Similarly, generation-repair approaches include the automated data augmentation framework [110] and OriGen [78], both using multi-task LoRA to train on code generation and error correction simultaneously. OriGen's dual-LoRA architecture separates Gen-LoRA for initial generation and Fix-LoRA for compiler feedback-driven repair.

Advanced approaches integrate multiple loss functions to optimize different code quality aspects simultaneously. RTLCoder [80] combines generation loss with contrastive loss ($\mathcal{L}_{compare} = \sum_{z_{i,k}<z_{i,\tau}} \max(s_{i,k} - s_{i,\tau} + \lambda, 0)$) using multi-task LoRA to address exposure bias. DeepRTL2 [60] extends this with full-parameter fine-tuning, integrating generation and contrastive losses for unified generative and embedding tasks. ITERTL [62] implements iterative training with generation and ranking losses ($\mathcal{L} = \alpha \times \mathcal{L}_{ce} + \beta \times \mathcal{L}_{rank}$), while Xu et al. [125] combine generation and decoding losses using hybrid full-parameter and QLoRA fine-tuning. Similarly, CodeV [71] demonstrates comprehensive multi-task capabilities by simultaneously handling code generation and Fill-in-Middle tasks across Verilog and Chisel languages using full-parameter fine-tuning.

These multi-task approaches demonstrate significant advantages over single-task training. The choice depends on specific requirements: dual-architecture frameworks for specialized optimization, multi-loss approaches for comprehensive quality enhancement and broader applicability.

(4) Knowledge-enhanced Tuning.

Knowledge-enhanced tuning approaches augment traditional fine-tuning by incorporating additional structural or reasoning information into the training process. These methods address the limitations of purely text-based training by integrating domain-specific knowledge representations to improve model understanding of Verilog code semantics and design logic.

RTL++ [66] introduces knowledge-enhanced LoRA fine-tuning by incorporating graph-structured textual descriptions derived from control flow graphs (CFG) and data flow graphs (DFG) of RTL code. The approach constructs "instruction-code-graph description" triplets, demonstrating the value of structural information for code generation quality.

Multiple works leverage DeepSeek-R1 reasoning trace distillation to construct "problem-reasoning-code" triplets that capture design, logic verification, and implementation steps. VeriThoughts [87]

 doi:10.20944/preprints202511.0656.v1

employs full-parameter fine-tuning method, while CodeV-R1 [72] extends this with DAPO optimization. ScaleRTL [68] incorporates test-time reasoning extension with iterative self-correction. ReasoningV [79] employs hybrid LoRA and full-parameter fine-tuning with adaptive reasoning mechanisms that dynamically adjust reasoning depth based on problem complexity.

These knowledge-enhanced approaches demonstrate substantial improvements over traditional fine-tuning. The integration of structural knowledge and reasoning processes addresses fundamental limitations in understanding complex hardware design logic, while adaptive mechanisms optimize performance-efficiency trade-offs.

6.2.3. Reinforcement Learning

While supervised approaches rely on static datasets, reinforcement learning methods enable dynamic optimization through interaction with the environment, offering potential for continuous improvement in code generation quality. Reinforcement learning for Verilog generation can be formalized as optimizing a policy π_θ that maximizes expected rewards:

$$J(\theta) = \mathbb{E}_{x \sim p_{data}, y \sim \pi_\theta(\cdot|x)}[R(y, y^*)] \tag{7}$$

where x represents the natural language specification, y denotes the generated Verilog code, y^* is the reference implementation, and $R(y, y^*)$ quantifies code quality through various metrics including functional correctness and structural similarity.

Contemporary research can be categorized based on reward function design and environmental feedback mechanisms:

(1) Structure-based Rewards.

VeriSeek [86] employs PPO with code-structure-guided rewards based on Abstract Syntax Tree (AST) similarity, formally defined as $r(y, \hat{y}) = 10 \times sim_{AST}(t_1, t_2)$ for parsable code, with negative penalties for invalid outputs. VeriReason [85] extends this concept using GRPO with multi-dimensional rewards that integrate structural similarity: $R(o) = 2.0$ for functionally correct code, $0.1 + 1.0 \cdot AST_{score}(o)$ for syntactically correct code, and 0 otherwise. Both approaches use LoRA adaptation to reduce computational overhead during iterative optimization.

(2) Tool-based Feedback.

Multiple approaches leverage compilation and verification tools for objective reward generation. VeriLogos [83] adopts formal equivalence checking through EDA tools, providing objective rewards that eliminate subjective biases. VeriPrefer [84] employs testbench-driven feedback with Direct Preference Optimization (DPO), where code passing more test cases is labeled as preferred. The DPO objective $\mathcal{L}_{RL} = -\mathbb{E}[\log \sigma(\beta \log \frac{\pi_\theta(y^+|x)}{\pi_\rho(y^+|x)} - \beta \log \frac{\pi_\theta(y^-|x)}{\pi_\rho(y^-|x)})]$ effectively aligns model outputs with functional correctness while avoiding reward hacking issues. CodeV-R1 [72] extends this with adaptive DAPO to reduce training time through rule-based functional equivalence checking.

(3) Multi-objective Optimization.

MCTS-augmented frameworks [90] formulate RTL generation as a Markov Decision Process with PPA-aware rewards, balancing functional correctness with power-performance-area optimization through systematic exploration of the token sequence space. This approach enables lookahead-guided decoding that considers downstream synthesis implications beyond basic functional correctness.

Empirical evidence demonstrates that RL-based approaches consistently outperform supervised fine-tuning baselines. The choice of reward mechanism depends on specific requirements: structure-based rewards for syntax correctness, tool-based feedback for functional validation, and multi-objective optimization for comprehensive design quality.

 doi:10.20944/preprints202511.0656.v1

Answer to RQ3: Adaptation and optimization techniques for LLM-based Verilog generation are categorized into training-free and training-based approaches. (1) Training-free methods include EDA tool feedback systems (encompassing single-agent iterative refinement and multi-agent collaborative architectures), prompt engineering strategies (hierarchical prompting and RAG-enhanced approaches), and inference optimization techniques (syntax-aware decoding and self-consistency reranking). (2) Training-based methods span pre-training (from-scratch and continued pre-training), supervised fine-tuning (data-centric, strategy-centric, multi-task, and knowledge-enhanced approaches), and reinforcement learning (structure-based rewards, tool-based feedback, and multi-objective optimization).

7. RQ4: Alignment Approaches for Verilog Code Generation

While LLMs demonstrate impressive capabilities in Verilog code generation as discussed in RQ3, ensuring these models produce outputs that align with human intentions, values, and requirements remains a critical challenge. Alignment in Verilog code generation extends beyond basic functionality, encompassing critical dimensions of security, efficiency, legal compliance, and reliability.

The alignment challenge for Verilog code generation presents distinct concerns due to the unique nature of hardware design. Unlike software, hardware implementations face physical constraints, resource limitations, and safety considerations when deployed in critical systems. Furthermore, since LLMs are trained on extensive data collections that may contain copyrighted content, the risk of intellectual property infringement is particularly significant in the hardware domain where IP protection has major commercial implications.

Based on our comprehensive review, we identify four essential alignment principles for Verilog code generation:

- **Security:** Ensuring generated Verilog code is free from vulnerabilities, backdoors, or other security issues that could affect system security when implemented in hardware.
- **Efficiency:** Optimizing generated code for resource usage (e.g., Performance, Power, Area), as poorly designed implementations can lead to excessive energy consumption, slower operation, or inefficient use of silicon resources.
- **Copyright:** Addressing ownership concerns by verifying that generated Verilog code does not infringe on existing patents, licenses, or copy other's designs without permission.
- **Hallucinations:** Reducing instances where LLMs produce code that appears correct but contains functional errors or logical inconsistencies that only become apparent during testing or implementation.

In the following sections, we will discuss each principle in detail, examining current methods, ongoing challenges, and promising solutions to enhance alignment in LLM-based Verilog code generation.

7.1. Security

Security concerns in LLM-generated Verilog code manifest in two primary dimensions: unintentional vulnerabilities arising from limited domain knowledge and intentional backdoors introduced through training data poisoning. Research addressing these challenges has emerged along complementary paths.

(1) Current Methods

For unintentional vulnerabilities, Saha et al. [126] developed SecFSM, which augments LLMs with structured security knowledge specifically for Finite State Machine implementations. SecFSM employs a knowledge graph architecture that formalizes hardware vulnerabilities (i.e., Dead State) and their remediation strategies, coupled with pre-analysis techniques to identify potential issues before generation. Finally, SecFSM integrates knowledge retrieval, planning, and execution to generate secure Verilog code. Similarly, Fan et al. [127] developed SecV, augmenting LLMs with Hardware-CWE

 doi:10.20944/preprints202511.0656.v1

knowledge graphs through three components: *SecV-builder* constructs security knowledge graphs via domain-specific corpus search and adaptive CoT; *SecV-verifier* optimizes graph precision through error detection and correction; *SecV-retriever* enables targeted knowledge retrieval through entity matching and graph exploration.

Regarding intentional backdoors, the RTL-Breaker [128] framework revealed critical vulnerabilities in the LLM training pipeline. This research demonstrated how malicious actors could poison training data with trigger-payload pairs, creating models that generate compromised hardware when specific triggers (typically rare words or code patterns) appear in prompts. These backdoored designs maintain functional correctness while introducing subtle harmful behaviors including performance degradation, data manipulation, and privilege escalation.

VeriContaminated [129] addresses data contamination in LLM-based hardware design, where benchmark data inadvertently appears in training datasets, inflating performance evaluations. Using CDD and Min-K% Prob detection methods, the study reveals that commercial models (GPT-3.5, GPT-4o) show 100% contamination rates while early open-source models exhibit lower rates, with contamination strongly correlating to generation performance and the TED algorithm providing effective mitigation with controlled trade-offs.

SALAD framework [130] applies machine unlearning techniques to remove sensitive hardware information from LLMs without full retraining while preserving core functionality. SALAD implements multiple unlearning algorithms (gradient-based, preference optimization, and representation misdirection) and evaluates effectiveness across four scenarios: benchmark contamination, custom designs, malicious code, and IP protection, demonstrating balanced security enhancement and performance preservation in RTL generation tasks.

(2) Ongoing Challenges

Both research directions underscore the need for specialized security alignment techniques in hardware description generation, as current evaluation frameworks remain inadequate for identifying sophisticated security vulnerabilities in LLM-generated Verilog code. ① While recent efforts have begun integrating CWE-based knowledge into LLMs (e.g., SecV), the coverage, depth, and effectiveness of these approaches remain limited, particularly for complex hardware-specific vulnerability patterns that extend beyond standard CWE categories or require deep circuit-level understanding. ② Research on backdoor attacks has primarily focused on attack methodologies while neglecting comprehensive defense strategies, creating an asymmetric understanding of security risks. ③ Other critical security concerns, such as jailbreak techniques that bypass safety guardrails and adversarial attacks that manipulate model behavior, have not been systematically investigated in the context of hardware description languages, leaving significant blind spots in security assessment frameworks.

(3) Promising Solutions

Several approaches show potential for addressing these security alignment challenges. ① Developing comprehensive security evaluation frameworks that extend beyond CWE standards to incorporate hardware-specific vulnerability patterns, coupled with systematic verification mechanisms to validate the effectiveness and completeness of security-enhanced generation approaches across diverse design scenarios. ② Implementing multi-stage verification pipelines that combine static analysis, formal verification, and simulation-based testing could enhance detection of subtle security flaws before synthesis. ③ Security-aware fine-tuning approaches that incorporate adversarial examples of hardware vulnerabilities may improve model robustness against both intentional and unintentional security weaknesses. ④ Collaborative research between hardware security experts and LLM researchers could lead to specialized red-teaming methodologies tailored to hardware description languages, enabling more systematic discovery and remediation of security vulnerabilities in generated Verilog code.

 doi:10.20944/preprints202511.0656.v1

7.2. *Efficiency*

Efficiency optimization in LLM-generated Verilog code represents a critical alignment challenge, as hardware designs must meet stringent Power, Performance, and Area (PPA) requirements for practical deployment. Unlike software where functionality alone may suffice, hardware implementations face physical constraints that demand meticulous optimization. Research in this domain has evolved along several promising directions.

(1) Current Methods

Various approaches have emerged to align LLM-generated Verilog code with PPA optimization requirements, which can be categorized into four main strategies.

For design space exploration approaches, researchers have developed methods that systematically search through possible design implementations. Delorenzo et al. [90] employed Monte Carlo Tree Search (MCTS) with a PPA-aware reward function, modeling RTL generation as a Markov Decision Process to guide exploration toward code with superior PPA characteristics, achieving up to 31.8% improvement in area-delay product while maintaining functional correctness. Similarly, Chang et al. [70] developed ChipGPT, generating multiple functionally correct candidates and selecting optimal designs based on enumerative search method, reducing area by up to 99.5% compared to naive LLM implementations.

For knowledge-enhanced approaches, Lu et al. [131] developed VeriOpt, enhancing LLMs through structured in-context learning that injects domain knowledge about PPA optimization techniques alongside synthesis reports, establishing a design-aware optimization framework that selects appropriate techniques based on specific design characteristics and constraints. Chang et al. [132] addresses data scarcity through program analysis techniques and implements "Chip Discovery Search" to combine existing design strengths, achieving area reduction by 9.8%, power reduction by 13.1%, and critical path delay reduction by 27.3%.

For design decomposition approaches, researchers have addressed complexity by breaking down problems into simpler components in different ways. Hierarchically, Tang et al. [94] introduced HiVeGen, which breaks complex designs into submodules with explicit PPA targets, enabling significant improvements for domain-specific accelerators, while Li et al. [93] proposed AutoSilicon to optimize large-scale designs through task decomposition. Sequentially, Thorat et al. [133] proposed LLM-VeriPPA, a two-stage framework separating correctness verification from PPA optimization, achieving average power reduction of 15-40% and area reduction of 10-35%. Similarly, RTLSquad [134] implements specialized agents for different optimization objectives that operate in a structured sequence, providing transparent decision-making while achieving measurable improvements like 9% dynamic power reduction in 16-bit adders.

(2) Ongoing Challenges

Despite these advances, several challenges persist in aligning LLM-generated Verilog code with efficiency requirements. ① LLMs inherently lack physical design awareness, struggling to predict how code transformations affect post-synthesis PPA metrics without external feedback mechanisms. ② Optimization techniques often involve complex trade-offs (e.g., pipelining improves performance but increases area), requiring sophisticated reasoning that exceeds current LLMs' capabilities. ③ PPA optimization remains highly context-dependent, varying across technology nodes, design constraints, and application domains, making it difficult to develop universally effective alignment strategies. ④ Computational overhead of PPA-aware approaches can be significant, with techniques like MCTS and multi-agent systems requiring substantial resources for large designs, limiting their practical application.

(3) Promising Solutions

Several approaches show potential for addressing these efficiency alignment challenges. ① Developing closed-loop optimization frameworks that incorporate EDA tool feedback directly into the

 doi:10.20944/preprints202511.0656.v1

generation process would provide LLMs with concrete signals about PPA impact. ② Implementing multi-objective optimization techniques that explicitly model PPA trade-offs could help balance competing constraints rather than optimizing single metrics in isolation. ③ Creating specialized fine-tuning datasets that pair natural language descriptions with PPA-optimized implementations would enable models to learn efficiency patterns beyond what in-context learning can provide. ④ Exploring modular approaches that combine multiple optimization strategies (e.g., hierarchical decomposition for complexity management, search-based methods for optimization, and agent-based systems for trade-off analysis) could address different aspects of the PPA challenge while controlling computational overhead.

7.3. Copyright

Copyright concerns in LLM-generated Verilog code present unique challenges at the intersection of intellectual property law and hardware design. These issues manifest in two critical dimensions: protecting the ownership of LLM-generated code and preventing LLMs from reproducing copyrighted hardware designs. Recent research has approached these challenges through complementary strategies.

(1) Current Methods

For protecting LLM-generated RTL code ownership, Wang et al. [135] introduced RTLMarker, a watermarking framework that embeds traceable signatures in generated Verilog code. RTLMarker employs 15 transformation rules operating at token and statement levels, including parameter encoding, variable name substitution, and redundant logic insertion. These transformations preserve functional correctness while maintaining watermark persistence through synthesis, enabling ownership verification at both RTL and gate-level netlist stages with detection accuracy exceeding 95% for RTL-level and 76% for netlist-level watermarks.

Regarding preventing copyright infringement, Bush et al. [61] developed FreeV, a framework focused on reducing the risk of LLMs reproducing proprietary hardware designs. FreeV includes a comprehensive dataset curation pipeline that applies three-layer filtering (license screening, per-file copyright detection, and deduplication) to remove potentially copyrighted material. Experiments demonstrated that LLMs fine-tuned on their carefully curated FreeSet dataset exhibited significantly lower infringement rates (3%) compared to models trained on conventional datasets (15%), while maintaining competitive performance on Verilog generation benchmarks.

(2) Ongoing Challenges

Despite these advances, significant copyright alignment challenges persist in LLM-generated Verilog code. ① Current watermarking techniques primarily focus on ownership protection but struggle to balance robustness against detection transparency, particularly when watermarks must survive complex hardware optimization pipelines. ② Existing copyright infringement detection mechanisms rely heavily on surface-level similarity metrics (e.g., cosine similarity), which may fail to identify functional equivalence implemented through different coding patterns. ③ The hardware domain's emphasis on design reuse and standardized interfaces creates inherent tension with copyright protection, as similar interfaces may appear across multiple designs without constituting infringement.

(3) Promising Solutions

Several approaches offer potential paths toward improved copyright alignment. ① Developing hardware-specific copyright evaluation benchmarks that incorporate both structural and functional similarity metrics would provide more nuanced assessment of potential infringement. ② Implementing hierarchical watermarking schemes that preserve ownership attribution across different abstraction levels (behavioral, RTL, gate-level) could improve robustness throughout the hardware development pipeline. ③ Creating specialized fine-tuning approaches that explicitly penalize generation of code patterns matching known proprietary designs while preserving learning of general hardware principles and public standards. ④ Exploring attribution mechanisms that automatically document the

 doi:10.20944/preprints202511.0656.v1

provenance of generated code elements, distinguishing between standard interfaces, common design patterns, and novel implementations.

7.4. Hallucinations

Hallucinations in LLM-generated Verilog code present a significant alignment challenge, manifesting as code that appears syntactically correct but contains functional errors, logical inconsistencies, or design flaws that only become apparent during testing or implementation. These issues can range from subtle logical errors to fundamental misunderstandings of hardware design principles, potentially leading to costly design iterations or hardware failures. Research addressing these challenges has emerged along several promising directions.

(1) Current Methods

Various approaches have been developed to mitigate hallucinations in LLM-generated Verilog code, which can be categorized into four main strategies.

For knowledge-enhanced approaches, researchers have developed methods that augment LLMs with domain-specific hardware design knowledge. HAVEN [75] constructed a Verilog-specific hallucination classification taxonomy and designed a three-stage mitigation pipeline with symbol interpretation, knowledge enhancement, and logic reinforcement. Similarly, HDLCoRe [122] implemented an HDL-aware Chain-of-Thought framework with self-verification mechanisms to guide generation according to hardware design principles without requiring external tools or training. ChatModel [95] introduced a multi-agent framework that standardizes design requirements into intermediate representations and uses hierarchical agile modeling to break complex designs into functional blocks with explicit verification at each stage.

For data-driven approaches, researchers have focused on enhancing training data quality to address hallucinations at their source. RTL++ [66] developed graph-enhanced instruction generation by converting RTL code into control and data flow graphs to capture structural dependencies, then fine-tuning models on this enriched representation to better understand hardware design patterns. Zhang et al. [136] systematically classified RTL code generation errors and created targeted remediation strategies including domain-specific knowledge bases and rule-guided requirement optimization. VeriReason [85] incorporated explicit reasoning paths into fine-tuning datasets, proposing a framework that combines reasoning enhancement, supervised fine-tuning, and testbench-based reinforcement learning with group relative policy optimization.

For decoding optimization approaches, researchers have modified the generation process itself to reduce hallucinations. DecoRTL [123] implemented contrast-based self-consistency decoding to penalize semantically redundant tokens and syntax-aware temperature adaptation that dynamically adjusts sampling parameters based on token type importance in RTL code. Paradigm-based generation [137] introduced expert-inspired paradigm blocks for different circuit types (sequential, combinational, behavioral) with a two-stage multi-round workflow that reuses high-quality intermediate results to minimize error propagation.

(2) Ongoing Challenges

Despite these advances, several challenges persist in addressing hallucinations in LLM-generated Verilog code. ① Current approaches often focus on specific hallucination types (syntax errors, control flow issues) without comprehensively addressing the full spectrum of hardware design vulnerabilities. ② Many techniques require extensive computational resources for verification, making them impractical for real-time code generation or large-scale designs. ③ The hardware domain's strict correctness requirements create tension with generative model capabilities, as even minor logical errors can render designs completely non-functional. ④ Hardware-specific evaluation benchmarks remain limited in their ability to systematically detect and measure subtle hallucinations, particularly those related to timing, power efficiency, and edge cases.

 doi:10.20944/preprints202511.0656.v1

(3) Promising Solutions

Several approaches show potential for addressing these hallucination challenges. ① Developing comprehensive hardware-specific verification frameworks that incorporate formal methods, simulation testing, and synthesis feedback could provide more robust hallucination detection across multiple design aspects. ② Implementing multi-stage generation pipelines that separate structural correctness from functional optimization could reduce computational overhead while maintaining high-quality output. ③ Creating specialized fine-tuning datasets that pair natural language descriptions with verified implementations would enable models to learn domain-specific patterns beyond what retrieval-based approaches can provide. ④ Exploring hybrid approaches that combine knowledge-based, structural decomposition, and decoding optimization techniques could address different aspects of the hallucination challenge simultaneously, providing more comprehensive mitigation strategies.

Answer to RQ4: Our analysis reveals that alignment of LLM-generated Verilog code requires addressing four key dimensions: security, efficiency, copyright, and hallucinations. Current approaches demonstrate promising results through knowledge enhancement, design decomposition, data-driven techniques, and decoding optimizations. However, significant challenges persist, including limited physical design awareness, computational overhead, and insufficient evaluation benchmarks. Future research should focus on developing comprehensive verification frameworks, multi-stage generation pipelines, and hybrid approaches that combine multiple alignment strategies.

8. The Road Ahead

In this section, we first discuss the limitations of current studies and then introduce a roadmap, illustrating the research trajectory shaped by prior work and highlighting future directions for exploration.

8.1. Limitations

Limitation 1: Limited Hardware Awareness. Current LLMs for Verilog generation lack intrinsic understanding of physical design constraints. Models trained primarily on software code or general Verilog syntax cannot adequately capture timing requirements, power budgets, and area optimization—constraints fundamental to hardware implementation. For instance, generated code may compile and simulate correctly but fail synthesis due to timing violations or resource conflicts. The predominant decoder-only autoregressive architectures, while effective for sequential text generation, struggle to capture Verilog's inherent parallelism and concurrent execution semantics, where multiple signal paths must be considered simultaneously.

Future research should develop hardware-aware LLMs that integrate physical design understanding through EDA tool feedback during pretraining and develop specialized architectural components for modeling concurrency and timing constraints.

Limitation 2: Data Scarcity and Quality Issues. High-quality Verilog training data remains orders of magnitude less abundant than general-purpose programming languages. As documented in RQ2, publicly available Verilog code exhibits substantial quality variation, ranging from academic exercises to production-grade IP. Many open-source repositories lack proper documentation, testbenches, or verification artifacts, making it difficult to assess functional correctness. Furthermore, data contamination concerns arise as benchmark problems may appear in training corpora, inflating reported performance metrics.

Constructing large-scale, high-quality Verilog datasets with comprehensive testbenches, verified implementations, and contamination-free benchmarks remains an urgent challenge for the community.

Limitation 3: Insufficient Benchmark Coverage. As analyzed in RQ2, most existing benchmarks contain fewer than 100 samples and focus on module-level generation tasks. Complex scenarios including hierarchical system designs, multi-clock domain implementations, cross-module dependen-

 doi:10.20944/preprints202511.0656.v1

cies, and industrial-scale projects remain largely unrepresented. Current benchmarks predominantly evaluate basic combinational and sequential logic, neglecting sophisticated patterns like pipelined datapaths, memory controllers, and protocol implementations. Additionally, many benchmarks lack comprehensive testbench coverage, making thorough functional verification difficult.

Future work should develop comprehensive benchmarks spanning multiple complexity levels, from basic modules to system-on-chip designs, with complete testbenches and PPA ground truth from actual synthesis runs. These benchmarks should incorporate complex instruction following capabilities similar to BigCodeBench [151], enabling evaluation of sophisticated multi-step design tasks and intricate specification interpretation. Additionally, cross-file and multi-module code generation benchmarks, analogous to RepoBench [152] and SWE-bench [153], are needed to assess models' abilities to maintain consistency across hierarchical designs and manage dependencies between interconnected hardware modules.

Limitation 4: Evaluation Metrics Limitations. While execution-based metrics (syntax-pass@k, functional-pass@k) have become standard, comprehensive hardware quality assessment remains limited. As shown in RQ2, only 10-15 studies reported PPA metrics due to computational expense. Critical dimensions including security vulnerabilities, synthesizability guarantees, timing closure success rates, and code maintainability lack standardized evaluation protocols. The emerging LLM-as-a-Judge approaches show promise but require rigorous validation to establish reliability and consistency with human expert judgments, particularly for domain-specific hardware correctness criteria.

Standardized evaluation frameworks integrating syntax validation, functional verification, PPA assessment, security analysis, and expert-aligned LLM judges are needed for comprehensive model comparison.

Limitation 5: Reliance on Base Models Without Domain Specialization. As documented in RQ1, the majority of research utilizes general-purpose Base LLMs without Verilog-specific adaptation. While 34 instruction-tuned models have been developed, most employ relatively lightweight fine-tuning on limited datasets. Few approaches systematically incorporate hardware design knowledge, formal verification principles, or EDA tool integration during model development. The gap between general code generation capabilities and hardware-specific requirements (timing, concurrency, synthesizability) remains substantial.

Future research should develop specialized foundation models pretrained on curated hardware design corpora with integrated reasoning capabilities and hardware-aware architectural innovations.

Limitation 6: Inadequate Alignment for Deployment. Current approaches inadequately address critical alignment dimensions identified in RQ4. Security analysis reveals limited coverage of hardware-specific CWE vulnerabilities and absent backdoor defense mechanisms. PPA optimization methods require expensive iterative feedback with synthesis tools. Copyright infringement detection remains immature, while hallucination mitigation provides limited coverage of diverse error manifestations. Moreover, no studies have integrated solutions into real industrial design workflows or demonstrated sustained collaboration with hardware engineers through interactive refinement and explainable generation.

Comprehensive alignment frameworks addressing security, efficiency, copyright, and trustworthiness, combined with deployment-ready features for human-AI collaboration, are essential for practical adoption.

8.2. Roadmap

8.2.1. Established Pathways in Current Studies.

As shown in our analysis (RQ1-RQ4), existing methods typically leverage either Base LLMs (open-source models like CodeLlama, DeepSeek-Coder, or closed-source models like GPT-4) or instruction-tuned models adapted through fine-tuning or prompting. Current approaches predominantly focus on module-level generation, utilizing datasets constructed through open-source mining or expert curation. Evaluation relies heavily on execution-based metrics (functional-pass@k), while alignment considerations and deployment features receive limited attention.

 doi:10.20944/preprints202511.0656.v1

8.2.2. Path Forward.

We outline three progressively advanced stages for exploring these new directions. In the first stage, researchers can study under-explored modules individually. In the second stage, they can investigate under-explored paths combining novel and established approaches. Finally, in the third stage, by transforming all under-explored areas into well-understood components, researchers can synthesize findings to develop next-generation LLM-based hardware design solutions. Below, we describe these stages in detail.

Stage 1: Exploring Under-explored Modules Individually. We outline promising opportunities for deeper investigation:

- *Opportunity 1: Hardware-Aware Foundation Models.* While existing studies predominantly use general-purpose LLMs (RQ1), there is substantial opportunity to develop billion-scale Verilog-specific foundation models pretrained on curated hardware design corpora. These models should integrate EDA tool feedback during pretraining to develop intrinsic understanding of physical constraints, incorporate specialized architectural components for modeling concurrency and timing, and employ hardware-aware tokenization strategies capturing domain-specific patterns. Initial efforts like ChipGPT [70] demonstrate feasibility, but systematic development of specialized foundation models with reasoning capabilities remains largely unexplored.
- *Opportunity 2: Comprehensive Benchmark Construction.* Current benchmarks suffer from limited scale and coverage (RQ2). A critical opportunity exists to construct large-scale benchmarks exceeding 1,000 samples with graduated complexity from basic modules to industrial SoC designs. These benchmarks should include complete testbench suites with comprehensive coverage metrics, verified PPA ground truth from actual synthesis runs, standardized formats enabling consistent evaluation, and contamination prevention mechanisms ensuring fair assessment. Community-driven efforts maintaining living benchmarks through continuous expert-verified additions would provide reliable progress measurement.
- *Opportunity 3: System-Level and Hierarchical Generation.* Current approaches focus predominantly on module-level generation. Substantial opportunities exist for system-level (SoC to RTL) hierarchical generation capable of managing cross-module dependencies, interface protocol compliance, and design partitioning. This includes automatic design decomposition identifying appropriate module boundaries, hierarchical planning generating top-level architectures before detailed implementation, interface synthesis ensuring protocol compatibility across modules, and IP integration reusing verified components. Recent work on hierarchical benchmarks [100,103] provides foundations for this direction.
- *Opportunity 4: Multimodal Verilog Generation.* While most studies process text-only inputs, practical hardware design involves diverse artifacts including circuit diagrams, timing waveforms, block schematics, and datasheet specifications. Vision-language models adapted for hardware design could enable designers to sketch conceptual architectures and automatically generate RTL implementations. Initial explorations like ChipGPTV [97] and VGV [92] demonstrate feasibility, but systematic development of multimodal generation frameworks integrating visual specifications with textual descriptions remains largely unexplored.
- *Opportunity 5: Closed-Loop PPA Optimization.* As identified in RQ4, current PPA optimization approaches rely on expensive iterative feedback. Opportunities exist for developing reinforcement learning algorithms with reward functions derived directly from synthesis results, implementing efficient gradient-free optimization suitable for discrete hardware design spaces, and establishing automated refinement frameworks where initial generations undergo systematic improvement based on EDA tool feedback. Integration of synthesis tools (Yosys, Design Compiler) into generation loops could enable PPA-aware generation without manual iteration.
- *Opportunity 6: Comprehensive Security Alignment.* Security analysis (RQ4) reveals limited coverage of hardware-specific vulnerabilities. Opportunities include systematic construction of security-focused datasets covering full CWE hardware taxonomy, development of specialized detection

 doi:10.20944/preprints202511.0656.v1

models identifying timing-based vulnerabilities, side-channel weaknesses, and trojan insertion points, robust backdoor defense mechanisms protecting against training data poisoning, and automated security verification integrating formal methods and simulation-based validation. Frameworks like SecFSM [126] and SecV [127] provide initial foundations.

- *Opportunity 7: Deployment-Ready Features.* No current studies integrate solutions into industrial design workflows (RQ4). Critical opportunities exist for developing interactive design tools supporting human-in-the-loop refinement, implementing explainable generation providing transparent rationales for design decisions, creating seamless IDE integration enabling designers to leverage AI assistance within existing workflows, and establishing intelligent feedback mechanisms learning from designer corrections to align with project-specific requirements. Such features are essential for transitioning from research prototypes to production-ready tools.

Stage 2: Exploring Under-explored Paths. Below, we highlight example combinations:

- Once hardware-aware foundation models are developed, researchers can apply established fine-tuning or prompting techniques to enhance system-level hierarchical generation, then extend these models to support interactive designer collaboration through explainable interfaces and iterative refinement.
- With comprehensive benchmarks established, researchers can re-evaluate existing approaches to assess true progress beyond potentially contaminated datasets. Combining new benchmarks with standardized evaluation frameworks integrating PPA assessment and security analysis would enable fair cross-method comparison.
- Researchers can leverage multimodal generation capabilities with closed-loop PPA optimization, enabling designers to sketch high-level architectures that are automatically refined through synthesis feedback to meet timing and power constraints while maintaining visual correspondence with original specifications.
- Developing hierarchical generation frameworks using hardware-aware foundation models with integrated security alignment could produce system-level designs that are both architecturally sound and free from hardware vulnerabilities, verified through automated formal checking integrated into the generation pipeline.

Paths consisting solely of under-explored modules also offer substantial opportunities. For instance, researchers could design multimodal system-level generation using specialized hardware-aware models, enhance effectiveness through closed-loop synthesis feedback, verify security through comprehensive vulnerability analysis, and deploy through IDE-integrated interactive interfaces enabling designer collaboration.

Stage 3: Synthesizing Knowledge for Next-Generation Solutions. In this most advanced stage, researchers transform under-explored aspects into well-understood areas and investigate diverse combination paths. At this stage, they can synthesize findings and analyze strategies for optimal LLM-based hardware design solution development. Leveraging accumulated experience and knowledge, they may push boundaries of LLM capabilities in automated hardware design to unprecedented heights.

Ideally, following the above roadmap, the community can develop highly effective solutions capable of performing module-level through system-level Verilog generation, utilizing comprehensive expert-curated benchmarks ensuring accurate evaluation. The ideal solution should facilitate seamless designer collaboration through interactive interfaces and explainable generation, achieve high functional correctness and PPA optimization while maintaining security and copyright compliance, integrate directly into industrial EDA workflows, and provide trustworthy expert-level assistance for real-world hardware design projects. Such solutions would fundamentally transform hardware design practices, substantially improving productivity, quality, and accessibility across the electronic design automation ecosystem.

 doi:10.20944/preprints202511.0656.v1

9. THREATS TO VALIDITY

Paper search omission. One key limitation is the possibility of omitting relevant papers during our search process for LLM-based Verilog code generation research. Despite systematically searching across six academic databases and employing both manual and automated approaches, the rapidly evolving nature of this field increases the risk of overlooking recent work, especially emerging preprints. To mitigate this concern, we implemented a comprehensive search strategy combining the Quasi-Gold Standard approach, rigorous keyword selection spanning both Verilog/HDL and LLM domains, and thorough forward and backward snowballing, ultimately identifying 102 papers (70 from peer-reviewed venues and 32 high-quality preprints).

Study selection bias. Another limitation involves potential bias in paper selection and quality assessment. Our three-stage filtering process, which reduced approximately 47,000 initial papers to 102, necessarily applied subjective judgments during quality assessment, particularly when evaluating preprints without formal peer review. To address this concern, we established transparent inclusion/exclusion criteria and a systematic quality assessment framework with five criteria evaluated on a Likert scale. Papers were required to achieve at least 80% of the maximum possible score (12/15 points), ensuring consistent standards across both published works and preprints while accommodating methodological diversity in this interdisciplinary field.

Categorization and analysis bias. Our systematic categorization of LLMs (Base vs. IT), datasets (benchmark vs. instruction-tuning), evaluation metrics (similarity-based, execution-based, LLM-based), adaptation techniques (EDA feedback, prompt engineering, inference optimization, pre-training, fine-tuning, reinforcement learning) and alignment approaches (security, efficiency, copyright, hallucination) may introduce classification bias through subjective interpretations. To minimize this bias, we conducted extensive literature review before establishing classification frameworks, referenced established taxonomies where available, and focused on extracting factual rather than interpretive information from each study. Additionally, our analysis of trends and future directions inevitably reflects our understanding of the field's trajectory, which may differ from other researchers' perspectives despite our efforts to remain objective.

10. CONCLUSION

LLMs are bringing significant changes to the field of Electronic Design Automation (EDA). The potential of these models to handle complex hardware design tasks can fundamentally reshape many Verilog development practices and tools. In this SLR, we analyzed the emerging utilization of LLMs for Verilog code generation, encompassing 102 papers published from 2020 to 2025. We examined the diverse LLMs that have been employed in Verilog code generation tasks and explored their distinct features and applications (RQ1). We then investigated the processes involved in dataset construction and evaluation metrics, emphasizing the significant role well-curated datasets play in the successful application of LLMs to solve hardware design tasks (RQ2). Following this, we investigated the various strategies utilized to optimize and assess the performance of LLMs for Verilog code generation tasks (RQ3). Lastly, we reviewed the alignment approaches for Verilog code generation where LLMs have been applied to date, shedding light on the practical considerations in security, efficiency, copyright, and hallucination mitigation (RQ4). We summarised some key existing challenges of LLM for Verilog code generation and provided a research roadmap, outlining promising future research directions in this rapidly evolving interdisciplinary field.

Acknowledgments: The authors would like to thank the editors and the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 62141208).

References

1. Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. arXiv preprint, 2021

 doi:10.20944/preprints202511.0656.v1

2. Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, et al. Code llama: Open foundation models for code. arXiv preprint, 2023
3. Qiwei Peng, Yekun Chai, Xuhong Li. HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization. In: Proceedings of Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024), 2024. 8383–8394
4. Qianhui Zhao, Fang Liu, Xiao Long, et al. On the Applicability of Code Language Models to Scientific Computing Programs. IEEE Transactions on Software Engineering, 2025, 51: 1685-1701
5. Yi Gui, Zhen Li, Yao Wan, et al. Webcode2m: A real-world dataset for code generation from webpage designs. In: Proceedings of Proceedings of the ACM on Web Conference 2025, 2025. 1834–1845
6. Sathvik Joel, Jie Wu, Fatemeh Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. ACM Transactions on Software Engineering and Methodology, 2025, Online
7. Xiaodong Gu, Meng Chen, Yalan Lin, et al. On the effectiveness of large language models in domain-specific code generation. ACM Transactions on Software Engineering and Methodology, 2025, 34: 1–22
8. Nan Wu, Yuan Xie, Cong Hao. Ai-assisted synthesis in next generation eda: Promises, challenges, and prospects. In: Proceedings of 2022 IEEE 40th International Conference on Computer Design (ICCD), 2022. 207-214
9. Hammond Pearce, Benjamin Tan, Ramesh Karri. DAVE: Deriving Automatically Verilog from English. In: Proceedings of MLCAD '20: 2020 ACM/IEEE Workshop on Machine Learning for CAD, Virtual Event, Iceland, 2020. 27–32
10. Barbara A. Kitchenham, Lech Madeyski, David Budgen. SEGRESS: Software Engineering Guidelines for REporting Secondary Studies. IEEE Transactions on Software Engineering, 2022, 49: 1273–1298
11. He Zhang, Muhammad Ali Babar, Paolo Tell. Identifying relevant studies in software engineering. Inf. Softw. Technol., 2011, 53: 625–637
12. Juyong Jiang, Fan Wang, Jiasi Shen, et al. A survey on large language models for code generation. arXiv preprint, 2024
13. Xiangping Chen, Xing Hu, Yuan Huang, et al. Deep learning-based software engineering: progress, challenges, and opportunities. Science China Information Sciences, 2025, 68: 111102
14. Wenji Fang, Jing Wang, Yao Lu, et al. A survey of circuit foundation model: Foundation ai models for vlsi circuit design and eda. arXiv preprint, 2025
15. Zhuolun He, Bei Yu. Large language models for eda: Future or mirage? In: Proceedings of Proceedings of the 2024 International Symposium on Physical Design, 2024. 65–66
16. Lei Chen, Yiqi Chen, Zhufei Chu, et al. Large circuit models: opportunities and challenges. Science China Information Sciences, 2024, 67: 200402
17. Xinyi Hou, Yanjie Zhao, Yue Liu, et al. Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 2024, 33: 1–79
18. Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention is all you need. In: Proceedings of Advances in Neural Information Processing Systems, 2017. 5998-6008
19. Tom Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In: Proceedings of Advances in Neural Information Processing Systems, 2020. 1877–1901
20. Yuntao Bai, Andy Jones, Kamal Ndousse, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. arXiv preprint, 2022
21. Shang Liu, Yao Lu, Wenji Fang, et al. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In: Proceedings of Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, 2024. 1–9
22. Xinyi Hou, Yanjie Zhao, Yue Liu, et al. Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology, 2024, 33: 1-79
23. Xin Zhou, Sicong Cao, Xiaobing Sun, et al. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. ACM Transactions on Software Engineering and Methodology, 2025, 34: 1-31
24. Xing Hu, Feifei Niu, Junkai Chen, et al. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. arXiv preprint, 2025
25. IEEE Xplore Database. https://ieeexplore.ieee.org
26. ACM Digital Library. https://dl.acm.org

 doi:10.20944/preprints202511.0656.v1

27. ScienceDirect Database. https://www.sciencedirect.com
28. Web of Science Database. https://www.webofscience.com
29. SpringerLink Database. https://link.springer.com
30. arXiv Database. https://arxiv.org
31. Meisam Abdollahi, Seyedeh Faegheh Yeganli, Mohammad Baharloo, et al. Hardware design and verification with large language models: A scoping review, challenges, and open issues. Electronics, 2024, 14: 120
32. Claudia Negri-Ribalta, Rémi Geraud-Stewart, Anastasia Sergeeva, et al. A systematic literature review on the impact of AI models on the security of code generation. Frontiers in Big Data, 2024, 7: 1386720
33. Lanxin Yang, He Zhang, Haifeng Shen, et al. Quality Assessment in Systematic Literature Reviews: A Software Engineering Perspective. Information and Software Technology, 2021, 130: 106397
34. Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of 18th International Conference on Evaluation and Assessment in Software Engineering, 2014. 1-10
35. Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. LLaMA: Open and Efficient Foundation Language Models. arXiv preprint, 2023
36. Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models. arXiv preprint, 2024
37. Xiao Bi, Deli Chen, Guanting Chen, et al. Deepseek llm: Scaling open-source language models with longtermism. arXiv preprint, 2024
38. Daya Guo, Qihao Zhu, Dejian Yang, et al. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. arXiv preprint, 2024
39. Daya Guo, Dejian Yang, Haowei Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. Nature, 2025, 645: 633–638
40. Jinze Bai, Shuai Bai, Yunfei Chu, et al. Qwen technical report. arXiv preprint, 2023
41. Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2. 5-coder technical report. arXiv preprint, 2024
42. Erik Nijkamp, Bo Pang, Hiroaki Hayashi, et al. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In: Proceedings of International Conference on Learning Representations, 2023
43. Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, et al. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv preprint, 2023
44. Raymond Li, Yangtian Zi, Niklas Muennighoff, et al. StarCoder: may the source be with you! Transactions on Machine Learning Research, 2023
45. Anton Lozhkov, Raymond Li, Loubna Ben Allal, et al. Starcoder 2 and the stack v2: The next generation. arXiv preprint, 2024
46. CodeGemma Team, Heri Zhao, Jeffrey Hui, et al. Codegemma: Open code models based on gemma. arXiv preprint, 2024
47. Yue Wang, Hung Le, Akhilesh Deepak Gotmare, et al. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In: Proceedings of EMNLP 2023-2023 Conference on Empirical Methods in Natural Language Processing, 2023. 1069–1088
48. Suriya Gunasekar, Yi Zhang, Jyoti Aneja, et al. Textbooks are all you need. arXiv preprint, 2023
49. Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, et al. Textbooks are all you need ii: phi-1.5 technical report. arXiv preprint, 2023
50. Bo Adler, Niket Agarwal, Ashwath Aithal, et al. Nemotron-4 340b technical report. arXiv preprint, 2024
51. OpenAI's GPT series models. https://platform.openai.com/docs/models
52. Anthropic's Claude series models. https://docs.claude.com/en/docs/about-claude/models/overview
53. Google's Gemini series models. https://ai.google.dev/gemini-api/docs/models
54. Haotian Liu, Chunyuan Li, Qingyang Wu, et al. Visual instruction tuning. In: Proceedings of Advances in Neural Information Processing Systems, 2023. 34892–34916
55. Mingzhe Gao, Jieru Zhao, Zhe Lin, et al. Autovcoder: A systematic framework for automated verilog code generation using llms. In: Proceedings of 2024 IEEE 42nd International Conference on Computer Design (ICCD), 2024. 162–169
56. PEI Zehua, Huiling Zhen, Mingxuan Yuan, et al. Betterv: Controlled verilog generation with discriminative guidance. In: Proceedings of International Conference on Machine Learning, 2024
57. Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, et al. Verilogeval: Evaluating large language models for verilog code generation. In: Proceedings of 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2023. 1–8

 doi:10.20944/preprints202511.0656.v1

58. Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, et al. CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair. In: Proceedings of International Conference on Learning Representations, 2025
59. Yi Liu, Changran XU, Yunhao Zhou, et al. DeepRTL: Bridging Verilog Understanding and Generation with a Unified Representation Model. In: Proceedings of International Conference on Learning Representations, 2025
60. Yi Liu, Hongji Zhang, Yunhao Zhou, et al. DeepRTL2: A Versatile Model for RTL-Related Tasks. In: Proceedings of Findings of the Association for Computational Linguistics, 2025. 6485–6500
61. Sam Bush, Matthew DeLorenzo, Phat Tieu, et al. Free and Fair Hardware: A Pathway to Copyright Infringement-Free Verilog Generation using LLMs. In: Proceedings of 2025 62nd ACM/IEEE Design Automation Conference (DAC), 2025. 1–7
62. Peiyang Wu, Nan Guo, Xiao Xiao, et al. Itertl: An iterative framework for fine-tuning llms for rtl code generation. In: Proceedings of 2025 IEEE International Symposium on Circuits and Systems (ISCAS), 2025. 1–5
63. Bardia Nadimi, Hao Zheng. A multi-expert large language model architecture for verilog code generation. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
64. Jinghua Wang, Lily Jiaxin Wan, Sanjana Pingali, et al. OpenRTLSet: A Fully Open-Source Dataset for Large Language Model-based Verilog Module Design. In: Proceedings of 2025 IEEE International Conference on LLM-Aided Design (ICLAD), 2025. 212–218
65. Bardia Nadimi, Ghali Omar Boutaib, Hao Zheng. Pyranet: A multi-layered hierarchical dataset for verilog. In: Proceedings of 2025 62nd ACM/IEEE Design Automation Conference (DAC), 2025. 1–7
66. Mohammad Akyash, Kimia Azar, Hadi Kamali. RTL++: Graph-enhanced LLM for RTL Code Generation. In: Proceedings of 2025 IEEE International Conference on LLM-Aided Design (ICLAD), 2025. 44–50
67. Peiyang Wu, Nan Guo, Junliang Lv, et al. RTLRepoCoder: Repository-Level RTL Code Completion through the Combination of Fine-Tuning and Retrieval Augmentation. arXiv preprint, 2025
68. Chenhui Deng, Yun-Da Tsai, Guan-Ting Liu, et al. ScaleRTL: Scaling LLMs with Reasoning Data and Test-Time Compute for Accurate RTL Code Generation. In: Proceedings of 2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD), 2025. 1–9
69. Prithwish Basu Roy, Akashdeep Saha, Manaar Alam, et al. Veritas: Deterministic Verilog Code Synthesis from LLM-Generated Conjunctive Normal Form. arXiv preprint, 2025
70. Kaiyan Chang, Ying Wang, Haimeng Ren, et al. Chipgpt: How far are we from natural language hardware design. arXiv preprint, 2023
71. Yang Zhao, Di Huang, Chongxiao Li, et al. CodeV: Empowering LLMs with HDL Generation through Multi-Level Summarization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2025. Early Access
72. Yaoyu Zhu, Di Huang, Hanqi Lyu, et al. CodeV-R1: Reasoning-Enhanced Verilog Generation. In: Proceedings of Advances in Neural Information Processing Systems, 2025. Accept
73. Gaoche Zhang, Dingyang Zou, Kairui Sun, et al. GEMMV: An LLM-based Automated Performance-Aware Framework for GEMM Verilog Generation. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2025, 15: 325-336
74. Weimin Fu, Shijie Li, Yifang Zhao, et al. Hardware phi-1.5 b: A large language model encodes hardware domain specific knowledge. In: Proceedings of 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), 2024. 349–354
75. Yiyao Yang, Fu Teng, Pengju Liu, et al. Haven: Hallucination-mitigated llm for verilog code generation aligned with hdl engineers. In: Proceedings of 2025 Design, Automation & Test in Europe Conference (DATE), 2025. 1–7
76. Yongan Zhang, Zhongzhi Yu, Yonggan Fu, et al. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
77. Emil Goh, Maoyang Xiang, I Wey, et al. From english to asic: Hardware implementation with large language model. arXiv preprint, 2024
78. Fan Cui, Chenyang Yin, Kexing Zhou, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In: Proceedings of Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, 2024. 1–9

 doi:10.20944/preprints202511.0656.v1

79. Haiyan Qin, Zhiwei Xie, Jingjing Li, et al. ReasoningV: Efficient Verilog Code Generation with Adaptive Hybrid Reasoning Model. arXiv preprint, 2025
80. Shang Liu, Wenji Fang, Yao Lu, et al. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024
81. Anjiang Wei, Huanmi Tan, Tarun Suresh, et al. VeriCoder: Enhancing LLM-Based RTL Code Generation through Functional Correctness Validation. arXiv preprint, 2025
82. Shailja Thakur, Baleegh Ahmad, Hammond Pearce, et al. Verigen: A large language model for verilog code generation. ACM Transactions on Design Automation of Electronic Systems, 2024, 29: 1–31
83. Kyungjun Min, Seonghyeon Park, Hyeonwoo Park, et al. Improving LLM-Based Verilog Code Generation with Data Augmentation and RL. In: Proceedings of 2025 Design, Automation & Test in Europe Conference (DATE), 2025. 1–7
84. Ning Wang, Bingkun Yao, Jie Zhou, et al. Insights from Verification: Training a Verilog Generation LLM with Reinforcement Learning with Testbench Feedback. arXiv preprint, 2025
85. Yiting Wang, Guoheng Sun, Wanghao Ye, et al. VeriReason: Reinforcement Learning with Testbench Feedback for Reasoning-Enhanced Verilog Generation. arXiv preprint, 2025
86. Ning Wang, Bingkun Yao, Jie Zhou, et al. Large language model for verilog generation with code-structure-guided reinforcement learning. In: Proceedings of 2025 IEEE International Conference on LLM-Aided Design (ICLAD), 2025. 164–170
87. Patrick Yubeaton, Andre Nakkab, Weihua Xiao, et al. Verithoughts: Enabling automated verilog code generation using reasoning and formal verification. In: Proceedings of Advances in Neural Information Processing Systems, 2025. Accept
88. Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, et al. Benchmarking large language models for automated verilog rtl code generation. In: Proceedings of 2023 Design, Automation & Test in Europe Conference Exhibition (DATE), 2023. 1–6
89. Vyom Kumar Gupta, Abhishek Yadav, Masahiro Fujita, et al. LLM-aided Front-End Design Framework For Early Development of Verified RTLs. In: Proceedings of 2024 IEEE 33rd Asian Test Symposium (ATS), 2024. 1–6
90. Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, et al. Make every move count: Llm-based high-quality rtl code generation using mcts. arXiv preprint, 2024
91. Paola Vitolo, George Psaltakis, Michael Tomlinson, et al. Natural Language to Verilog: Design of a Recurrent Spiking Neural Network using Large Language Models and ChatGPT. In: Proceedings of 2024 International Conference on Neuromorphic Systems (ICONS), 2024. 110–116
92. Sam-Zaak Wong, Gwok-Waa Wan, Dongping Liu, et al. VGV: Verilog generation using visual capabilities of multi-modal large language models. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
93. Cangyuan Li, Chujie Chen, Yudong Pan, et al. Autosilicon: Scaling up rtl design generation capability of large language models. ACM Transactions on Design Automation of Electronic Systems, 2025, 30: 1–26
94. Jinwei Tang, Jiayin Qin, Kiran Thorat, et al. Hivegen–hierarchical llm-based verilog generation for scalable chip design. In: Proceedings of 2025 IEEE International Conference on LLM-Aided Design (ICLAD), 2025. 30–36
95. Jianmin Ye, Tianyang Liu, Qi Tian, et al. ChatModel: Automating Reference Model Design and Verification with LLMs. arXiv preprint, 2025
96. Shailja Thakur, Jason Blocklove, Hammond Pearce, et al. Autochip: Automating hdl generation using llm feedback. arXiv preprint, 2023
97. Kaiyan Chang, Zhirong Chen, Yunhao Zhou, et al. Natural language is not enough: Benchmarking multi-modal generative AI for Verilog generation. In: Proceedings of Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, 2024. 1–9
98. Matthew DeLorenzo, Vasudev Gohil, Jeyavijayan Rajendran. Creativeval: Evaluating creativity of llm-based hardware code generation. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
99. Jason Blocklove, Siddharth Garg, Ramesh Karri, et al. Evaluating llms for hardware design and test. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–6
100. Andre Nakkab, Sai Qian Zhang, Ramesh Karri, et al. Rome was not built in a single step: Hierarchical prompting for llm-based chip design. In: Proceedings of Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD, 2024. 1–11

 doi:10.20944/preprints202511.0656.v1

101. Yao Lu, Shang Liu, Qijun Zhang, et al. Rtllm: An open-source benchmark for design rtl generation with large language model. In: Proceedings of 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), 2024. 722–727
102. Ahmed Allam, Mohamed Shalan. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
103. Suresh Purini, Siddhant Garg, Mudit Gaur, et al. ArchXBench: A Complex Digital Systems Benchmark Suite for LLM Driven RTL Synthesis. In: Proceedings of 2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD), 2025. 1–10
104. Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, et al. Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification. arXiv preprint, 2025
105. Gwok-Waa Wan, Yubo Wang, Sam-Zaak Wong, et al. GenBen: A Generative Benchmark for LLM-Aided Design. Arxiv, 2025
106. Pengwei Jin, Di Huang, Chongxiao Li, et al. RealBench: Benchmarking Verilog Generation Models with Real-World IP Designs. arXiv preprint, 2025
107. Ce Guo, Tong Zhao. ResBench: A Resource-Aware Benchmark for LLM-Generated FPGA Designs. In: Proceedings of Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2025. 25–34
108. Nathaniel Pinckney, Christopher Batten, Mingjie Liu, et al. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. ACM Transactions on Design Automation of Electronic Systems, 2025, 30:1–20
109. Paul E Calzada, Zahin Ibnat, Tanvir Rahman, et al. VerilogDB: The Largest, Highest-Quality Dataset with a Preprocessing Framework for LLM-based RTL Generation. arXiv preprint, 2025
110. Kaiyan Chang, Kun Wang, Nan Yang, et al. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In: Proceedings of Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024. 1–6
111. Shang Liu, Wenji Fang, Yao Lu, et al. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. In: Proceedings of 2024 IEEE LLM Aided Design Workshop (LAD), 2024. 1–5
112. Zeju Li, Changran Xu, Zhengyuan Shi, et al. DeepCircuitX: A Comprehensive Repository-Level Dataset for RTL Code Understanding, Generation, and PPA Analysis. In: Proceedings of 2025 IEEE International Conference on LLM-Aided Design (ICLAD), 2025. 204–211
113. Xiang Chen, Guang Yang, Zhan-Qi Cui, et al. Survey of State-of-the-art Automatic Code Comment Generation. Journal of Software, 2021, 32: 2118
114. HDLBits Website. https://hdlbits.01xz.net/wiki/Main_Page
115. Manar Abdelatty, Jingxiao Ma, Sherief Reda. MetRex: A Benchmark for Verilog Code Metric Reasoning Using LLMs. In: Proceedings of Proceedings of the 30th Asia and South Pacific Design Automation Conference, 2025. 995–1001
116. OpenCores Website. https://opencores.org
117. Enrique Dehaerne, Bappaditya Dey, Sandip Halder, et al. A deep learning framework for verilog autocompletion towards design and verification automation. arXiv preprint, 2023
118. Mohammad Akyash, Hadi Mardani Kamali. Simeval: Investigating the similarity obstacle in llm-based hardware code generation. In: Proceedings of Proceedings of the 30th Asia and South Pacific Design Automation Conference, 2025. 1002–1007
119. Guang Yang, Wei Zheng, Xiang Chen, et al. The Cream Rises to the Top: Efficient Reranking Method for Verilog Code Generation. arXiv preprint, 2025
120. Matthew DeLorenzo, Kevin Tieu, Prithwish Jana, et al. Abstraction-of-Thought: Intermediate Representations for LLM Reasoning in Hardware Design. arXiv preprint, 2025
121. Kangbo Bai, Peiran Yan, Lifeng Liu, et al. VeriRAG: Design AI-Specific CPU Co-processor with RAG-Enhanced LLMs. In: Proceedings of 2025 International Symposium of Electronics Design Automation (ISEDA), 2025. 76–82
122. Heng Ping, Shixuan Li, Peiyu Zhang, et al. Hdlcore: A training-free framework for mitigating hallucinations in llm-generated hdl. arXiv preprint, 2025
123. Mohammad Akyash, Kimia Azar, Hadi Kamali. DecoRTL: A Run-time Decoding Framework for RTL Code Generation with LLMs. arXiv preprint, 2025

 doi:10.20944/preprints202511.0656.v1

124. Zhuorui Zhao, Ruidi Qiu, Chao Lin, et al. Vrank: Enhancing verilog code generation from large language models via self-consistency. In: Proceedings of 2025 26th International Symposium on Quality Electronic Design (ISQED), 2025. 1–7
125. Changran Xu, Yi Liu, Yunhao Zhou, et al. Speculative Decoding for Verilog: Speed and Quality, All in One. In: Proceedings of 62nd ACM/IEEE Design Automation Conference, DAC 2025, San Francisco, CA, USA, June 22-25, 2025, 2025. 1–7
126. Ziteng Hu, Yingjie Xia, Xiyuan Chen, et al. SecFSM: Knowledge Graph-Guided Verilog Code Generation for Secure Finite State Machines in Systems-on-Chip. arXiv preprint, 2025
127. Fanghao Fan, YingJie Xia, Li Kuang. SecV: LLM-based Secure Verilog Generation with Clue-Guided Exploration on Hardware-CWE Knowledge Graph. In: Proceedings of Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, 2025. 8049–8057
128. Lakshmi Likhitha Mankali, Jitendra Bhandari, Manaar Alam, et al. Rtl-breaker: Assessing the security of llms against backdoor attacks on hdl code generation. In: Proceedings of 2025 Design, Automation & Test in Europe Conference (DATE), 2025. 1–7
129. Zeng Wang, Minghao Shao, Jitendra Bhandari, et al. VeriContaminated: Assessing LLM-driven verilog coding for data contamination. arXiv preprint, 2025
130. Zeng Wang, Minghao Shao, Rupesh Karn, et al. SALAD: Systematic Assessment of Machine Unlearing on LLM-Aided Hardware Design. arXiv preprint, 2025
131. Kimia Tasnia, Alexander Garcia, Tasnuva Farheen, et al. Veriopt: Ppa-aware high-quality verilog generation via multi-role llms. arXiv preprint, 2025
132. Kaiyan Chang, Wenlong Zhu, Kun Wang, et al. A data-centric chip design agent framework for Verilog code generation. ACM Transactions on Design Automation of Electronic Systems, 2025, 30: 1-27
133. Kiran Thorat, Jiahui Zhao, Yaotian Liu, et al. LLM-VeriPPA: Power, Performance, and Area Optimization aware Verilog Code Generation with Large Language Models. In: Proceedings of 2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD), 2025. 1–7
134. Bowei Wang, Qi Xiong, Zeqing Xiang, et al. Rtlsquad: Multi-agent based interpretable rtl design. arXiv preprint, 2025
135. Kun Wang, Kaiyan Chang, Mengdi Wang, et al. Rtlmarker: Protecting llm-generated rtl copyright via a hardware watermarking framework. In: Proceedings of Proceedings of the 30th Asia and South Pacific Design Automation Conference, 2025. 808–813
136. Jiazheng Zhang, Cheng Liu, Huawei Li. Understanding and Mitigating Errors of LLM-Generated RTL Code. arXiv preprint, 2025
137. Wenhao Sun, Bing Li, Grace Li Zhang, et al. Paradigm-based automatic hdl code generation using llms. In: Proceedings of 2025 26th International Symposium on Quality Electronic Design (ISQED), 2025. 1–8
138. Hanxian Huang, Zhenghan Lin, Zixuan Wang, et al. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. arXiv preprint, 2024.
139. Yunda Tsai, Mingjie Liu, Haoxing Ren. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. In: Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024. 1–6
140. Ping Guo, Yiting Wang, Wanghao Ye, et al. EvoVerilog: Large Langugage Model Assisted Evolution of Verilog Code. arXiv preprint, 2025.
141. Jason Blocklove, Shailja Thakur, Benjamin Tan, et al. Automatically Improving LLM-based Verilog Generation using EDA Tool Feedback. ACM Transactions on Design Automation of Electronic Systems, 2025, 30: 1-26
142. Mubashir ul Islam, Humza Sami, Pierre-Emmanuel Gaillardon, et al. AIvril: AI-Driven RTL Generation With Verification In-The-Loop. arXiv preprint, 2024.
143. Sriram Ranga, Rui Mao, Debjyoti Bhattacharjee, et al. RTL Agent: An Agent-Based Approach for Functionally Correct HDL Generation via LLMs. In: Proceedings of 2024 IEEE 33rd Asian Test Symposium (ATS), 2024. 1–6
144. Yujie Zhao, Hejia Zhang, Hanxian Huang, et al. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. In: Proceedings of 2025 62nd ACM/IEEE Design Automation Conference (DAC), 2025. 1–7
145. Chia-Tung Ho, Haoxing Ren, Brucek Khailany. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2025. 300-307
146. Zhendong Mi, Renming Zheng, Haowen Zhong, et al. CoopetitiveV: Leveraging LLM-powered Coopetitive Multi-Agent Prompting for High-quality Verilog Generation. arXiv preprint, 2025.

 doi:10.20944/preprints202511.0656.v1

147. Humza Sami, Mubashir ul Islam, Samy Charas, et al. Nexus: A Lightweight and Scalable Multi-Agent Framework for Complex Tasks Automation. arXiv preprint, 2025.
148. Yangbo Wei, Zhen Huang, Huang Li, et al. VFlow: Discovering Optimal Agentic Workflows for Verilog Generation. arXiv preprint, 2025.
149. Bardia Nadimi, Ghali Omar Boutaib, Hao Zheng. VeriMind: Agentic LLM for Automated Verilog Generation with a Novel Evaluation Metric. arXiv preprint, 2025.
150. Claudia Negri-Ribalta, Rémi Geraud-Stewart, Anastasia Sergeeva, et al. A systematic literature review on the impact of AI models on the security of code generation. Frontiers in Big Data, 2024, 7: 1386720.
151. Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In: Proceedings of International Conference on Learning Representations, 2025
152. Tianyang Liu, Canwen Xu, Julian McAuley. RepoBench: A Benchmark for Repo-Level Code Generation. In: Proceedings of International Conference on Learning Representations, 2023
153. Carlos E. Jimenez, John Yang, Alexander Wettig, et al. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In: Proceedings of International Conference on Learning Representations, 2024