WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Large Language Models for Verilog Generation and Verification: A Review of Cloud, Domain-Specialised and Resource-Constrained Approaches

Amol Dhote[1], Rajnandinee Patle[2], Mansi Borkute[3], Khushi Gumgaonkar[4], Manaswi Deogirkar[5], Dr. (Mrs.) S. W. Varade[6]

[1–6]Department of Electronics and Telecommunication Engineering, Priyadarshini College of Engineering, Nagpur, Maharashtra, India
Email: (rajnandineepatle1@gmail.com)

DOI : https://doi.org/10.55248/gengpi.07.1026.34022

ABSTRACT –

Register-transfer-level (RTL) design is a critical but labour-intensive stage in digital hardware development. Large language models (LLMs) have recently shown the ability to convert natural-language specifications into Verilog code, thereby reducing manual design effort. This paper presents a review of LLM-based Verilog generation and verification approaches, organised around deployment requirements and verification strategies. The review covers classical electronic design automation (EDA) foundations, general-purpose cloud coding assistants, Verilog-specialised models and benchmarks, feedback-driven and agentic EDA frameworks, open-source verification back ends, and resource-constrained deployment using quantisation and lightweight inference engines. The literature indicates that cloud- and GPU-based systems remain dominant, while compiler and simulator feedback can improve the syntactic validity of generated designs. However, functional correctness remains difficult because compilation alone does not verify intended hardware behaviour. Resource-constrained deployment is feasible in selected cases; for example, RTLCoder reports a 4-bit quantised model of approximately 4 GB that can operate on a laptop. Nevertheless, consistent reporting of latency, memory use, energy consumption, offline operation, privacy, and reproducibility is still limited. This review identifies major open challenges, including functional verification, benchmark quality, scarcity of high-quality Verilog training data, security of generated hardware, small-model efficiency, and standardised resource-aware evaluation. The findings suggest that reliable AI-assisted RTL design will require closer integration of LLMs with deterministic EDA tools, stronger verification oracles, improved benchmarks, and transparent evaluation practices.

Key Words: *large language models, Verilog, RTL generation, hardware verification, quantisation, electronic design automation*

1. INTRODUCTION

Digital integrated circuits are described at the register-transfer level (RTL) using hardware description languages (HDLs), among which Verilog is widely used in education and industry. Writing RTL requires an understanding of concurrency, timing, synthesis semantics, and hardware resource constraints, which differs significantly from conventional sequential programming. LLM-generated RTL may contain syntax errors, width mismatches, incorrect hardware behaviour, and other implementation issues, making compilation and functional verification important parts of the design workflow [8]–[11].

Automation has long played an important role in digital design. Logic minimisation and multi-level synthesis have reduced manual design effort, while formal methods support reasoning about circuit behaviour [1]–[3]. More recently, machine-learning methods have been applied to physical-design tasks such as macro placement [4]. Code-capable large language models (LLMs) introduce another possibility: a designer can describe a circuit in natural language and receive candidate Verilog code [5], [8]–[11].

A rapidly growing body of research examines LLM-based RTL generation. It includes benchmarks [8], [10], [11], fine-tuned Verilog models [9], [15], conversational hardware design [12], industrial domain adaptation [13], and feedback loops using EDA tools [14], [18], [33]. Practical concerns remain: cloud-hosted models may require confidential design information to be sent to external servers; high-end GPUs can be costly; and generated code may be syntactically incorrect, functionally wrong, or insecure [32]. Quantisation and lightweight inference engines have therefore increased interest in local or low-cost deployment [22]–[26].

Existing surveys have reviewed LLM-based Verilog generation and broader LLM applications in electronic design automation [34], [35]. This paper complements that work with a focused deployment- and verification-oriented narrative synthesis, comparing execution environment, verification depth, privacy assumptions, and resource requirements. It does not claim to be an exhaustive systematic review; its contribution is an engineering-oriented interpretation of representative studies and the trade-offs that affect practical use.

The main contributions of this paper are as follows:

1. A deployment- and verification-oriented taxonomy of LLM-based RTL generation approaches.
2. A critical comparison of representative methods based on execution environment, verification method, privacy, and hardware requirements.
3. A discussion of how deterministic EDA tools can complement probabilistic LLMs.
4. A consolidated list of open research challenges.

The remainder of this paper is organised as follows. Section 2 presents background on EDA and HDL semantics. Section 3 describes the review methodology. Section 4 reviews general code LLMs and cloud assistants. Section 5 discusses Verilog-specialised models and benchmarks. Section 6 reviews feedback-driven and agentic EDA frameworks. Section 7 examines open-source EDA tools as verification back ends. Section 8 reviews efficient and local deployment. Section 9 presents a comparative analysis. Section 10 discusses open challenges, and Section 11 concludes the paper.

2. BACKGROUND

2.1 Evolution of Logic Synthesis and EDA

Electronic design automation (EDA) became essential as the increasing complexity of integrated circuits made manual hardware design impractical. Early logic-synthesis research focused on Boolean-function minimisation and the development of algorithms for reducing circuit complexity. Although exact minimisation methods can have exponential worst-case complexity, heuristic and algorithmic techniques make practical optimisation possible for many designs [1].

Later, multilevel synthesis systems represented digital circuits as networks of Boolean operations and applied algebraic and Boolean transformations to improve their structure. Reduced ordered binary decision diagrams (BDDs) provide a canonical representation for Boolean functions under a fixed variable ordering and support Boolean-function manipulation and equivalence checking [2]. And-inverter graphs (AIGs) provide another graph-based representation that supports efficient logic optimisation and rewriting [3].

A typical digital-design flow begins with an RTL description and proceeds through elaboration, logic synthesis, technology mapping, placement, and routing. Simulation and formal-verification techniques can be applied at different stages to evaluate functional behaviour and identify design errors. These stages provide different forms of evidence: successful compilation or synthesis alone does not establish that a circuit implements its intended specification [27]–[31].

2.2 Machine Learning in EDA

Before the emergence of large language models (LLMs), machine-learning techniques were applied to optimisation, prediction, and decision-making problems in EDA. For example, Mirhoseini et al. formulated chip floorplanning as a reinforcement-learning problem and demonstrated competitive placement results for accelerator designs [4]. Such approaches use machine learning to optimise aspects of the physical-design process.

These methods generally operate on structured representations of a design or its physical environment, such as graphs, netlists, and placement information. In contrast, LLM-based approaches can operate at the specification and code-generation stages, accepting natural-language descriptions and producing candidate hardware description language (HDL) code. The two approaches address different parts of the design process and may be complementary rather than interchangeable.

2.3 HDL Semantics Versus Software Semantics

Verilog describes hardware using constructs that represent concurrent processes, signals, and event-driven behaviour. Statements in separate procedural blocks may execute concurrently, while synthesis tools interpret supported HDL constructs to infer hardware components such as combinational logic, registers, and interconnections. This differs from conventional software programming, where statements are generally understood as instructions executed by a processor [8]–[11], [29].

Several HDL-specific errors can affect the correctness or synthesizability of generated RTL:

- **Incomplete assignments:** In a combinational process, failing to assign an output under every relevant condition can infer an unintended latch.
- **Multiple drivers:** Driving the same signal from multiple sources can cause conflicts, unknown values, or tool-dependent errors, depending on the signal type and design context.
- **Width mismatches:** Assignments and expressions involving incompatible bit widths can produce truncation, extension, or unintended arithmetic results.
- **Loops and excessive hardware:** Synthesizable loops are commonly elaborated into hardware structures. Loops with large bounds can therefore produce unexpectedly large circuits.

- **Assignment semantics:** Incorrect use of blocking and non-blocking assignments can lead to unintended simulation behaviour or simulation–synthesis mismatches.

3. REVIEW METHODOLOGY

This paper adopts a focused narrative review approach to examine recent developments in large language model (LLM)-based Verilog generation and verification. Rather than attempting to provide an exhaustive systematic review, the study focuses on representative research works, tools, and benchmarks that illustrate the capabilities, limitations, and practical deployment requirements of LLM-assisted hardware design.

3.1 Scope of the Review

The review covers research on LLM-based RTL generation, hardware-oriented model specialisation, verification using electronic design automation (EDA) tools, and deployment under different computational constraints. Classical EDA concepts are included where they help explain the role of synthesis, simulation, and formal verification in evaluating generated hardware descriptions.

The literature is organised into the following categories:

- **General-purpose code LLMs and cloud assistants:** Approaches that use general code-generation models to produce or refine Verilog from natural-language specifications.
- **Verilog-specialised models and benchmarks:** Models adapted to hardware description languages and benchmark datasets used to assess RTL-generation performance.
- **Feedback-driven and agentic EDA frameworks:** Systems that incorporate compiler, simulator, or other EDA-tool feedback into iterative code generation and correction.
- **Open-source EDA verification tools:** Tools that can be integrated into local workflows to check syntax, compile designs, simulate behaviour, or support formal analysis.
- **Efficient and resource-constrained deployment:** Approaches involving model quantisation, lightweight inference engines, and local execution on systems with limited memory or processing capacity.

3.2 Literature Selection and Analysis

Representative publications were considered according to their relevance to LLM-assisted Verilog generation, hardware verification, model specialisation, EDA integration, or resource-efficient deployment. Benchmark studies and tool-oriented research were included where they provided evidence about generated RTL quality, verification practices, or practical execution requirements.

The selected works were examined qualitatively rather than combined through a statistical meta-analysis. The analysis focused on the purpose of each approach, its execution environment, the type of model used, the verification methods reported, and the limitations identified by the authors. Particular attention was given to distinguishing reported experimental results from broader claims about practical applicability.

This review does not claim that every publication in the field was identified or that a formal systematic screening protocol was followed. Its purpose is to compare representative approaches and develop an engineering-oriented understanding of their strengths, limitations, and deployment trade-offs.

3.3 Comparison Criteria

To support a consistent comparison across different approaches, the reviewed studies were examined using the following criteria:

- **Generation capability:** Whether the approach generates Verilog or RTL directly from natural-language descriptions, modifies existing code, or supports an iterative design workflow.
- **Model type:** Whether the system uses a general-purpose LLM, a hardware-specialised model, or a model integrated into a larger tool-assisted framework.
- **Execution environment:** Whether inference is performed through a cloud service, on a local workstation, or on resource-constrained hardware.
- **Verification strategy:** Whether evaluation includes syntax checking, compilation, simulation, testbench-based functional checking, formal verification, or feedback-driven correction.
- **Hardware and memory requirements:** The reported computing resources, model size, quantisation level, and other deployment constraints, where such information is available.
- **Privacy and offline operation:** Whether the approach supports local execution and whether the publication provides evidence about network dependence or handling of potentially sensitive design information.

- **Limitations and reproducibility:** The reported weaknesses, benchmark constraints, availability of code or models, and completeness of experimental reporting.

When a publication did not report a particular metric or deployment property, that information was treated as unavailable rather than inferred. Similarly, a reported compilation success rate was not treated as equivalent to functional correctness, because successful compilation alone does not establish that generated RTL meets its intended specification.

3.4 Synthesis of Findings

The findings were synthesised by comparing the selected approaches across the categories and criteria described above. The comparison emphasises the relationship between model capability and the surrounding design workflow, including the role of deterministic EDA tools in checking and refining LLM-generated code.

The review also considers the practical trade-offs between cloud-based capabilities and local deployment. These include computational requirements, access to external services, potential confidentiality concerns, and the availability of verification tools. Since the reviewed studies may use different benchmarks, models, hardware platforms, and evaluation procedures, their reported results are interpreted in context rather than assumed to be directly comparable.

This methodology provides the basis for the following sections, which examine general-purpose LLMs, Verilog-specialised models, feedback-driven EDA frameworks, open-source verification tools, and resource-constrained deployment.

4. GENERAL CODE LLMs AND CLOUD CODING ASSISTANTS

4.1 Foundations of Code LLMs

Modern code-generating large language models (LLMs) are largely based on the transformer architecture, which enables models to learn patterns in source code and generate contextually relevant outputs [19]. Codex, a GPT model fine-tuned on publicly available code, demonstrated the potential of LLMs for program synthesis. Its evaluation also highlighted the use of multiple generated candidates and the pass@k metric to measure the likelihood that at least one candidate satisfies a programming task [5].

The development of open-weight models has expanded access to code-generation capabilities. The LLaMA family provided foundation models that could be adapted for different applications [30]. Code Llama extended this approach with models designed for code generation, infilling, and processing longer code contexts [6]. Similarly, the Qwen2.5-Coder series offers code-focused models in multiple sizes, providing options for different computational requirements [7].

Architectural techniques also influence the practicality of deploying these models. Rotary position embeddings support the representation of token positions, while grouped-query attention can reduce key-value cache requirements compared with conventional multi-head attention configurations [20], [21]. These techniques can be useful when deploying models on systems with limited memory. However, practical resource requirements also depend on parameter count, context length, numerical precision, and inference implementation.

Although these models provide general code-generation capabilities, their performance on conventional programming tasks does not automatically establish their reliability for hardware description languages. Verilog generation requires correct handling of concurrency, bit widths, timing, reset behaviour, and synthesizable constructs. Consequently, general code LLMs must be evaluated specifically on hardware-design tasks.

4.2 Use of General Assistants for Hardware Design

Early evaluations investigated whether general-purpose code models could generate Verilog without specialised hardware-design training. Thakur et al. evaluated code-generation models on hardware-design problems and reported limitations in compilation and functional correctness. Their work also examined the benefits of fine-tuning for the evaluated tasks [8]. These findings indicate that generating syntactically plausible code is not sufficient to ensure that an RTL implementation behaves according to its specification.

Blocklove et al. explored conversational hardware co-design using a general-purpose language model. Their work demonstrated how natural-language interaction could support activities such as specification refinement, RTL generation, and testbench development [12]. However, conversational assistance does not remove the need for engineering judgement. Designers must still inspect generated code, identify incorrect assumptions, and use appropriate verification methods to evaluate the resulting design.

Security is another consideration when using code-generating models. Pearce et al. investigated security vulnerabilities in AI-generated software-code suggestions [32]. Although their study focused on software rather than hardware, it highlights the broader need to assess generated code rather than assuming that plausible output is secure. For RTL generation, relevant concerns include unintended logic, unsafe interfaces, and the possibility of incorrect or malicious hardware behaviour. The software-focused results should not, however, be interpreted as a direct measurement of hardware vulnerability rates.

Overall, general-purpose assistants can reduce the effort required to draft and modify RTL, but their outputs should be treated as candidate implementations. Hardware-specific checks and verification remain necessary before the generated code can be considered suitable for its intended use.

4.3 Strengths and Limitations of Cloud Assistants

Cloud-based assistants provide access to capable models through remote services, reducing the need for users to install and maintain large models locally. They can support rapid prototyping, explanation of unfamiliar HDL constructs, generation of alternative implementations, and assistance with testbench development. Their practical value is particularly clear when users need access to models that exceed the computational capacity of their local systems.

Nevertheless, cloud-based deployment introduces several considerations for hardware-design workflows.

- **Data confidentiality:** Design specifications, source code, and related project information may be transmitted to an external service. The actual confidentiality implications depend on the provider's data-handling policies, account configuration, and applicable organisational requirements.
- **Network dependence:** Access to a remote model requires a suitable network connection and availability of the service.
- **Cost and usage restrictions:** Pricing, quotas, and access conditions may limit repeated generation and verification cycles.
- **Reproducibility:** Model updates, service changes, and differences in generation settings can affect the outputs obtained from the same task.
- **Verification requirements:** A cloud assistant may produce code that appears correct but contains syntax errors, incorrect logic, or behaviour that does not match the specification.

These limitations do not mean that cloud assistants are unsuitable for hardware design. Rather, they indicate that model selection should consider the design environment, sensitivity of the project data, computational resources, and required verification level.

For confidential or disconnected environments, locally hosted models may offer greater control over where design data is processed. However, local deployment introduces its own challenges, including memory requirements, inference speed, model installation, and maintenance. These trade-offs are examined in later sections of this review.

In summary, general-purpose code LLMs and cloud assistants provide a useful starting point for AI-assisted RTL development. Their strengths lie in accessibility and flexible code-generation support, while their limitations highlight the need for hardware-specific evaluation and reliable verification. This provides the motivation for examining Verilog-specialised models and dedicated RTL-generation benchmarks in the next section.

5. VERILOG-SPECIALISED MODELS AND BENCHMARKS

5.1 Domain-Specialised Models for RTL Generation

General-purpose code LLMs can generate Verilog, but their performance may be limited by the specialised syntax and semantics of hardware description languages. To address this limitation, researchers have explored domain adaptation, fine-tuning, and hardware-oriented training data to improve model performance on RTL-generation tasks.

RTLCoder is an example of a model developed specifically for RTL code generation. It investigates the use of code models adapted to Verilog and explores the role of model training and evaluation in hardware-design tasks [9], [15]. Domain specialisation can help a model learn common HDL constructs, module structures, and patterns used in digital circuit descriptions. However, its effectiveness depends on the quality, diversity, and correctness of the training data.

Industrial domain adaptation provides another direction. Models can be adapted using domain-relevant code and design practices to improve their usefulness in specialised engineering workflows [13]. Nevertheless, improvements on a particular dataset do not necessarily demonstrate generalisation to unfamiliar designs, complex sequential circuits, or different synthesis environments.

Domain-specialised models should therefore be evaluated not only by their ability to produce Verilog-like output but also by whether the generated designs compile, pass functional tests, and satisfy the intended hardware specification.

5.2 Benchmarks for RTL Generation

Benchmarks provide a structured way to compare LLMs on hardware-design problems. They typically supply design descriptions or specifications and evaluate the generated RTL against expected results or predefined tests.

VerilogEval provides a benchmark for assessing LLM-generated Verilog using hardware-design tasks derived from the HDLBits instructional collection [10]. RTLLM provides another evaluation framework for examining LLM-based RTL generation across representative design tasks [11]. These benchmarks help researchers compare models using common tasks and evaluation procedures.

However, benchmark results depend on several factors, including task difficulty, prompt formulation, model configuration, the number of generated candidates, and the available test cases. Results from different benchmarks should therefore not be compared directly unless their evaluation conditions are sufficiently similar.

A useful benchmark should distinguish between at least three outcomes:

1. **Syntax or compilation success:** Whether the generated code can be processed by the selected compiler or HDL tool.
2. **Functional correctness:** Whether the generated design produces the expected behaviour for the evaluated inputs and operating conditions.
3. **Generalisation:** Whether the model can handle unfamiliar specifications and design patterns beyond those represented in its training data.

Compilation is an important initial check, but it does not guarantee functional correctness. Similarly, passing a finite set of test cases does not establish correctness for every possible input or operating condition. The strength of the conclusion depends on the coverage and quality of the verification method.

5.3 Evaluation Metrics and Limitations

Several metrics can be used to evaluate LLM-based RTL generation. Compilation or syntax success measures whether the output can be processed by the selected toolchain. Functional pass rate measures the proportion of tasks that satisfy the available tests or reference behaviour. Pass@k measures the likelihood that at least one of multiple generated candidates succeeds under a defined evaluation procedure [5].

These metrics capture different aspects of performance and should not be treated as interchangeable. For example, a model may achieve a high compilation rate but a lower functional pass rate if some generated designs implement the wrong logic. Likewise, pass@k can improve when more candidates are generated, but doing so may increase inference time and computational cost.

Benchmark-based evaluation also has limitations. Small or repetitive datasets may not represent the variety of real-world hardware designs. Testbenches may fail to exercise important corner cases, while overlap between training data and benchmark tasks can inflate reported performance. Differences in tool versions, prompt templates, decoding settings, and evaluation procedures can further affect reproducibility.

For these reasons, benchmark results should be interpreted alongside the experimental setup and verification methodology. Where possible, evaluations should report compilation success, functional correctness, candidate-generation settings, and the tools used to assess the outputs. More complex designs should also be included to examine sequential behaviour, reset handling, interfaces, and other HDL-specific requirements.

5.4 Implications for Practical RTL Development

Verilog-specialised models and benchmark frameworks represent important progress toward reliable AI-assisted hardware design. Domain adaptation can improve performance on selected RTL tasks, while benchmarks provide a more structured basis for assessing generated designs [9]–[11], [15].

However, neither specialisation nor benchmark success eliminates the need for independent verification. A model that performs well on a benchmark may still fail when faced with unfamiliar requirements, unusual timing conditions, or more complex modules. Practical deployment should therefore combine model-generated RTL with deterministic tools such as HDL compilers, simulators, and, where appropriate, formal-verification tools.

The distinction between code generation and verified hardware behaviour is central to evaluating these systems. This also motivates feedback-driven approaches, in which the results of EDA tools are used to identify errors and guide subsequent generation or correction. Such approaches are discussed in the next section.

6. FEEDBACK-DRIVEN AND AGENTIC EDA FRAMEWORKS

6.1 Motivation for Feedback-Driven RTL Generation

A major limitation of using a large language model (LLM) to generate Verilog is that the model may produce code that appears correct but fails during compilation or simulation. A single generation step cannot reliably establish whether the resulting RTL satisfies the intended hardware behaviour. Feedback-driven approaches address this limitation by combining LLM generation with electronic design automation (EDA) tools that evaluate the generated code and provide information for subsequent refinement.

AutoChip explores the use of LLM feedback for automating hardware description language generation [14]. In a feedback-driven workflow, the model first generates candidate RTL from a design prompt. The candidate is then passed to an appropriate EDA tool, such as a compiler or simulator. Errors or failed test results are used to guide another generation or correction step. This process can be repeated until the design passes the selected checks or a predefined stopping condition is reached.

The main advantage of this approach is that the model does not need to rely entirely on its own assessment of the generated code. Instead, it can use concrete tool outputs to identify problems and refine its solution. However, the quality of the result still depends on the accuracy of the feedback, the test cases provided, and the ability of the model to interpret and correct the reported issues.

6.2 Autonomous and Agent-Based EDA

Agent-based EDA frameworks extend the feedback-driven approach by allowing an LLM-powered system to interact with tools through multiple steps. Rather than generating RTL and stopping, an agent can coordinate design generation, tool execution, error analysis, and subsequent refinement. Such workflows aim to reduce manual intervention while retaining the use of established EDA tools for checking intermediate results.

ChatEDA presents an autonomous-agent approach to EDA using a large language model [18]. This direction illustrates how language models can serve as coordinators that interpret design objectives, interact with EDA tools, and use the resulting feedback to guide further actions. The broader significance is that LLMs can contribute not only to code generation but also to the organisation of multi-step design workflows.

Nevertheless, an agentic workflow should not automatically be interpreted as a fully autonomous or production-ready hardware-design system. Tool access, supported operations, execution constraints, and verification coverage determine what the agent can accomplish. Human review may still be necessary, particularly when design requirements are ambiguous or a failure could affect the reliability or security of the final hardware.

6.3 Typical Feedback-Driven Workflow

A general feedback-driven RTL generation workflow consists of the following stages:

1. **Design specification:** The user describes the required circuit, including its inputs, outputs, and expected behaviour.
2. **RTL generation:** The LLM produces an initial Verilog implementation.
3. **Compilation or elaboration:** An EDA tool checks whether the code can be processed and whether structural or language-related errors are present.
4. **Functional verification:** Simulation or another suitable verification method checks the implementation against the intended behaviour.
5. **Feedback analysis:** The system interprets compiler diagnostics or failed test results to identify potential problems.
6. **RTL refinement:** The LLM generates a corrected implementation using the available feedback.
7. **Reverification:** The revised code is checked again, and the process continues until the chosen acceptance criteria are met or the iteration limit is reached.

This workflow separates code generation from code evaluation. Compilation success establishes that the code passes particular language and elaboration checks; it does not prove functional correctness. Similarly, passing a limited set of simulation tests provides evidence for the tested cases but does not guarantee correctness for every possible input or operating condition.

6.4 Benefits and Limitations

Feedback-driven and agentic frameworks offer several potential benefits. First, they can help identify syntax errors and implementation mistakes that may not be obvious from inspecting generated code. Second, iterative refinement can reduce the effort required to correct common RTL problems. Third, tool integration provides a more structured development process than accepting the model's first response without further checking. Finally, agent-based systems can coordinate several design activities within a single workflow.

Despite these advantages, several limitations remain. Repeated model calls and tool executions can increase runtime and computational cost. A system may also become trapped in repeated correction cycles if the feedback is unclear or the model repeatedly introduces new errors. Functional verification is particularly challenging when testbenches are incomplete, design requirements are underspecified, or corner cases are overlooked. An incorrect testbench can even make a faulty implementation appear acceptable.

Another important consideration is reproducibility. The model version, generation settings, tool versions, constraints, testbench, and stopping criteria can all influence the outcome. Therefore, evaluations should report these details wherever possible. The use of automated feedback should also not be confused with formal proof: unless an appropriate formal method is applied under stated assumptions, a successful workflow does not establish complete correctness.

6.5 Implications for Practical RTL Development

Feedback-driven generation provides a practical way to combine the flexibility of LLMs with the deterministic checking capabilities of EDA tools. For educational circuits and small RTL modules, compilation and simulation feedback may be sufficient to catch many common mistakes. More complex designs may require stronger testbenches, assertion-based verification, formal equivalence checking, or additional design-rule checks.

The most appropriate role for an LLM in these workflows is therefore as a generation and refinement component within a verification-oriented design process, rather than as a replacement for established EDA methods. AutoChip and ChatEDA represent complementary directions: the former focuses on feedback-based HDL generation, while the latter explores autonomous-agent interaction with EDA tools [14], [18]. Together, these approaches motivate workflows in which generated RTL is repeatedly evaluated and improved before being considered suitable for further design stages.

Overall, feedback-driven and agentic EDA frameworks are an important step toward more reliable AI-assisted hardware development. Their practical effectiveness, however, must be assessed through clearly defined evaluation criteria, reproducible experiments, and verification methods appropriate to the intended design.

7. OPEN-SOURCE EDA TOOLS AS VERIFICATION BACK ENDS

Commercial simulators and synthesis tools provide mature capabilities for hardware development, but their use may involve licensing restrictions or additional costs. Open-source electronic design automation (EDA) tools offer practical alternatives for evaluating LLM-generated Verilog and integrating verification into automated design workflows [26]–[28], [31].

7.1 Open-Source Tools for RTL Evaluation

Icarus Verilog is a command-line Verilog compiler and simulator that supports parsing, elaboration, and simulation [27]. It can be used to check whether generated RTL is accepted by the tool and to evaluate its behaviour with a suitable testbench. Yosys provides RTL synthesis capabilities and is widely used in scripted hardware-design and research workflows [28]. For FPGA-oriented development, open-source place-and-route tools such as nextpnr can support subsequent implementation stages when used with a compatible synthesis flow [31]. Verilator is another widely used open-source tool that provides Verilog and SystemVerilog processing, linting, and simulation capabilities.

These tools are useful in LLM-assisted workflows for three main reasons. First, they can be invoked through command-line interfaces and scripts, making automated evaluation possible. Second, their outputs provide concrete diagnostic information that can be used to identify errors in generated RTL. Third, many common evaluation tasks can be performed on general-purpose computing hardware, without requiring a commercial EDA environment.

7.2 Integration with LLM-Based Workflows

Open-source EDA tools can serve as verification back ends in a feedback-driven RTL generation system. After an LLM produces Verilog, a script can pass the code to a compiler or synthesis tool and collect its output. If the tool reports an error, the diagnostic message can be provided to the LLM to support a correction attempt. If compilation succeeds, the design can proceed to simulation using a testbench that describes the expected behaviour.

This integration creates a repeatable process in which code generation and code evaluation are handled by different components. The LLM proposes a solution, while the EDA tools check specific properties of that solution. The resulting tool output can also help distinguish syntax-related failures from failures observed during functional testing.

However, open-source tools do not necessarily support identical language features or interpret every construct in the same way. Some SystemVerilog features may be unsupported or handled differently across tools and versions. Consequently, a design accepted by one tool may be rejected by another. Researchers should report tool versions, command-line options, configuration settings, and relevant language restrictions to improve reproducibility [27], [28].

7.3 Compilation, Simulation, and Functional Correctness

An important distinction must be maintained between successful compilation and functional correctness. Compilation and elaboration can identify syntax errors, unresolved references, and certain structural problems, but they cannot establish that the circuit implements the intended specification. A syntactically valid module may still contain incorrect Boolean logic, incorrect state transitions, timing-related mistakes, or boundary-condition errors.

Simulation provides additional evidence by executing the design against a testbench. The quality of this evidence depends on the testbench's correctness and coverage. If important input combinations or operating conditions are omitted, faulty behaviour may remain undetected. Assertions and formal verification methods can provide stronger checks for specified properties, while equivalence checking can be used to compare implementations under appropriate assumptions [10], [11], [27]–[29].

Therefore, a reliable evaluation process should distinguish at least three outcomes: successful compilation, successful functional testing, and successful completion of any additional verification checks. These outcomes are related but are not interchangeable. A design that compiles successfully should not automatically be classified as functionally correct.

7.4 Limitations and Practical Implications

Although open-source EDA tools make automated RTL evaluation more accessible, they do not eliminate the challenges of hardware verification. Their effectiveness depends on language support, tool configuration, testbench quality, and the properties being checked. Synthesis success alone does not prove that the generated circuit behaves as required, and passing a finite set of simulation tests cannot guarantee correctness for all possible inputs.

Tool integration also introduces engineering considerations. Automated workflows must manage source files, compiler options, error messages, simulation results, and failed execution cases. When diagnostic messages are supplied to an LLM, they may need to be shortened or structured to avoid

unnecessary context and to help the model focus on the relevant error. Repeated correction attempts should also be limited to prevent excessive execution time and computational cost.

Overall, open-source EDA tools provide an important foundation for reproducible and cost-conscious evaluation of LLM-generated RTL. Their greatest value comes from their integration into a verification-oriented workflow in which generation, compilation, simulation, and stronger verification methods are treated as distinct stages. This approach supports more meaningful assessment of AI-generated hardware while retaining the established role of EDA tools in checking design quality.

8. EFFICIENT AND LOCAL DEPLOYMENT OF LLMs

Deploying large language models (LLMs) locally can reduce dependence on cloud services and help protect sensitive design information. However, local deployment introduces constraints related to memory capacity, processing speed, model size, and power consumption. Quantisation, lightweight inference engines, and efficient model architectures are therefore important considerations when selecting an LLM for resource-constrained environments.

8.1 Quantisation

Quantisation reduces the numerical precision used to represent model parameters. For example, a model with seven billion parameters stored at 16-bit precision requires approximately 14 GB of memory for its weights alone. Additional memory is needed for runtime buffers, intermediate activations, and the key-value (KV) cache. Consequently, the total memory requirement during inference can exceed the weight-storage requirement.

Post-training quantisation reduces the precision of an already trained model without requiring complete retraining. Common approaches use 8-bit or 4-bit representations to reduce model storage and memory-bandwidth requirements [22], [23]. GPTQ is a post-training quantisation method that uses approximate second-order information to minimise quantisation error [22]. LLM.int8() enables 8-bit matrix multiplication while handling outlier features through higher-precision computation [23]. QLoRA, in contrast, combines a quantised base model with low-rank adapters to support memory-efficient fine-tuning [24]. These methods address related but distinct requirements and should not be treated as interchangeable techniques.

Quantisation can make larger models more practical on systems with limited memory. However, reducing numerical precision may affect generation quality, and the impact depends on the model, quantisation method, and task. For Verilog generation, evaluation should pay particular attention to exact syntax, signal widths, parameter values, and hardware-specific constructs. A quantised model should therefore be evaluated using both generation-quality measures and suitable compilation or functional-verification tests rather than assuming that lower memory use guarantees acceptable RTL output [15], [22]–[24].

8.2 Inference Engines and Model Formats

An inference engine provides the software needed to execute a trained model and generate outputs from user prompts. The llama.cpp project provides a C/C++ inference engine designed to support local execution of compatible language models. It supports the GGUF model format and multiple quantisation options, making it useful for investigating inference on systems without high-end GPUs [26].

GGUF is a model-file format used to store model weights and associated metadata for compatible inference implementations. Together, an appropriate model file and inference engine can support offline generation without requiring every prompt to be sent to a cloud-based service. This can be useful in environments where network connectivity, recurring API costs, or confidentiality requirements are important considerations.

Nevertheless, local execution does not automatically imply that a model will run efficiently on every device. Feasibility depends on the model's parameter count, quantisation level, available memory, context length, runtime implementation, and acceptable response time. Performance can also vary with processor architecture, memory bandwidth, software build options, and the number of execution threads. Reproducible evaluations should therefore document the model version, model-file format, quantisation level, inference-engine version, hardware configuration, and relevant runtime settings [26].

8.3 Architectural Features for Small Devices

Model architecture can also affect the resources required during inference. Grouped-query attention (GQA) reduces key-value cache requirements by sharing key and value projections across groups of query heads [21]. This can lower memory consumption, particularly when the model processes longer prompts or retains more tokens in its context.

However, reducing KV-cache memory does not eliminate the memory required to store model weights, activations, and runtime buffers. The overall requirements depend on both the model architecture and the inference workload. Longer contexts can increase memory use, so practical deployment should account for prompt length and generated output length.

Smaller code-specialised models can offer another way to reduce deployment requirements. For example, lower-parameter variants of Qwen2.5-Coder may be more suitable for constrained hardware than larger models, although their performance on complex coding tasks may differ [7]. The choice of model should therefore balance resource requirements against the quality of generated RTL.

For resource-constrained devices, model selection should be based on measured performance rather than parameter count alone. Relevant measurements include peak memory consumption, model-file size, generation speed, response latency, and the proportion of generated designs that pass compilation and functional tests.

8.4 Performance Considerations

Inference performance is influenced by memory bandwidth, processor capability, quantisation, context length, and the implementation of the inference engine. CPU-based generation can be practical for smaller quantised models, but response times may become substantial for larger models or longer outputs. GPU acceleration may improve throughput where suitable hardware is available, although it introduces additional hardware and deployment requirements.

For hardware-design applications, raw token-generation speed is not sufficient to evaluate practical performance. A model may generate Verilog quickly but require several correction attempts before producing a valid design. A more meaningful evaluation should consider the complete workflow, including prompt processing, generation, compilation, simulation, and any iterative refinement.

A useful evaluation framework should report the following measures:

1. **Memory consumption:** Peak RAM or GPU memory used during inference.
2. **Model storage:** Size of the model file and its quantisation format.
3. **Inference latency:** Time required to produce an output under specified conditions.
4. **Generation throughput:** Number of generated tokens per second, where measurable.
5. **RTL validity:** Proportion of generated designs that pass the selected compilation or elaboration checks.
6. **Functional correctness:** Proportion of designs that pass a clearly defined testbench or other functional-verification procedure.

These measures should be reported with the hardware configuration, software versions, prompt settings, and evaluation dataset. Without such details, comparisons between deployment methods may be misleading.

Overall, quantisation and lightweight inference engines provide practical options for investigating local LLM deployment. Their usefulness for Verilog generation, however, depends on the relationship between computational efficiency and hardware-specific code quality. Future evaluations should measure both resource consumption and verification outcomes to determine whether a local model provides a suitable balance of privacy, cost, speed, and reliability.

9. COMPARATIVE ANALYSIS

The reviewed approaches differ in their model architectures, deployment environments, verification strategies, and computational requirements. General-purpose code LLMs offer broad coding capabilities, while Verilog-specialised models focus more directly on hardware description languages. Feedback-driven frameworks add tool-assisted refinement, and quantised local models aim to reduce the resources required for inference. Open-source EDA tools provide a practical means of evaluating the generated RTL.

A meaningful comparison must consider not only code-generation capability but also how each approach is evaluated, the resources it requires, and the conditions under which it can be used. Table I summarises representative approaches according to these dimensions. The entries describe general characteristics of the approaches rather than results from a single controlled experiment.

9.1 Comparison of Deployment Approaches

Table -1: Comparative overview of LLM-based RTL generation approaches

Approach	Strengths	Limitations	Verification Support	Deployment
General-purpose cloud LLMs	Flexible code generation and explanation	May produce syntax errors or incorrect logic	Requires independent checking	Usually cloud-based; internet access generally required

Approach	Strengths	Limitations	Verification Support	Deployment
Verilog-specialised LLMs	Focused on Verilog syntax and RTL-generation tasks	Depends on training data, model capability, and task complexity	Benchmark evaluation; functional correctness still requires checking	Local or cloud, depending on model and resources
Benchmark-driven approaches	Enables structured comparison across models and tasks	Results depend on benchmark coverage, prompts, and evaluation criteria	Uses defined test cases and metrics where available	Requires a consistent evaluation environment
Feedback-driven and agentic EDA frameworks	Uses tool feedback to iteratively improve generated RTL	Depends on feedback quality and tool integration	May support compilation, simulation, and iterative correction	Requires compatible EDA tools and additional execution time
Open-source EDA verification tools	Enables reproducible checks using established tools	Cannot guarantee specification correctness without suitable tests	Supports compilation, simulation, and selected formal checks when configured	Often suitable for local workflows
Quantised and locally deployed LLMs	Can reduce memory requirements and enable offline use	Speed and output quality depend on the model and hardware	Requires a separate verification workflow using appropriate EDA tools	Depends on available memory, processor, and model size

Source: Qualitative synthesis of the literature reviewed in this paper.

Note: These categories are not mutually exclusive. A Verilog-specialised model may also be quantised, deployed locally, and connected to a feedback-driven EDA workflow. Benchmarks evaluate model performance, whereas EDA tools help check the generated RTL.

Three observations follow from Table I. First, cloud-hosted models provide convenient access to capable systems, whereas local models offer greater control over execution and data handling. Second, verification methods vary substantially across the literature, so results based on compilation alone should not be compared directly with results based on functional testing. Third, feedback-driven systems and open-source EDA tools can complement specialised models by providing external checks and information for refinement [10], [11], [14], [15], [18], [27], [28], [33].

9.2 Deployment Dimensions Are Coupled

Deployment is not simply a choice between cloud and local execution. A hosted model can provide access to substantial computational resources without requiring the user to maintain local accelerators. However, it may introduce service dependence, network latency, recurring costs, and data-governance considerations. A locally hosted model offers more direct control over the inference environment and can keep prompts and RTL within the user's system, provided that the complete workflow is configured accordingly. In exchange, the operator must manage hardware resources, software updates, access control, and model maintenance.

Model size and quantisation also influence deployment feasibility. A smaller or quantised model may require less memory, but reduced resource requirements do not necessarily imply equal generation quality. Similarly, a locally executed model may protect design data from transmission to an external inference service, but local execution alone does not guarantee complete privacy or security. Logs, temporary files, network services, and application permissions must also be considered.

These dimensions should therefore be assessed together. A useful deployment decision considers model capability, memory requirements, inference latency, verification support, privacy requirements, and the cost of operating the complete system rather than relying on a single label such as "local," "efficient," or "private."

9.3 Verification Depth and Benchmark Interpretation

The reviewed approaches use different levels of verification. At the first level, parsing and compilation check whether generated RTL can be processed by a particular tool. At the second level, simulation against a testbench evaluates the design's behaviour for the input sequences exercised by that testbench. At a deeper level, assertions, formal properties, or equivalence checking can evaluate specified behaviours under the assumptions and limitations of the verification setup [10], [11], [27]–[29].

This distinction is essential when comparing published results. A reported pass may indicate successful compilation in one study and functional success in another. These outcomes should not be treated as equivalent. Future evaluations should separately report compilation success, elaboration success, functional pass rate, testbench coverage where available, synthesis success, and formal-verification results where applicable.

Benchmarks such as VerilogEval and RTLLM provide defined task sets for comparing model outputs under specified conditions [10], [11]. However, benchmark performance is evidence of capability on the evaluated tasks, not proof that a model can design arbitrary hardware. Small educational circuits may not adequately represent hierarchical designs, parameterised modules, bus protocols, complex state machines, or timing-sensitive interfaces. Test suites may also omit corner cases, and reference implementations may rely on assumptions that are not explicit in the natural-language prompt.

For fair comparisons, studies should report the model version, number of generated candidates, decoding settings, retry policy, maximum output length, tool versions, and testbench details. The distinction between first-attempt success and pass@k is also important. First-attempt success measures performance without repeated candidate selection, whereas pass@k measures whether a successful candidate can be obtained from up to k attempts under the stated evaluation procedure. In feedback-driven systems, the number of refinement iterations, tool-call budget, total runtime, and final independent verification procedure should also be reported.

These details help distinguish improvements in the underlying model from gains obtained through additional sampling, repeated correction, or more extensive tool use.

9.4 Practical Selection Framework

The most appropriate approach depends on the application requirements. For confidential industrial designs, deployment should begin with the organisation's data-handling policy and threat model. Where external transmission is prohibited, an appropriately secured local or approved private deployment may be necessary. For educational tasks and rapid prototyping, a general-purpose assistant or specialised open model may be sufficient if generated RTL is independently checked.

Where local computational resources are limited, smaller models and quantisation may make inference more practical. Their suitability should be established through measurements of memory consumption, generation latency, and RTL validity on representative tasks. When reliability is the main priority, integrating compiler and simulator feedback can help identify errors, but the workflow should also include functional tests appropriate to the design.

A practical selection process should therefore consider:

1. **Design complexity:** Whether the target consists of simple combinational circuits, sequential modules, or larger hierarchical designs.
2. **Verification requirements:** Whether compilation is sufficient for initial screening or whether simulation, assertions, or formal methods are needed.
3. **Computational resources:** Available memory, processor or GPU capability, storage, and acceptable latency.
4. **Data governance:** Whether prompts and generated RTL may be processed by external services.
5. **Reproducibility:** Whether model settings, tool versions, test cases, and evaluation procedures can be documented.
6. **Total workflow cost:** The resources needed for generation, correction, and verification rather than generation alone.

Overall, no single approach is best for every application. General-purpose models offer convenience, specialised models focus on RTL generation, feedback-driven frameworks support iterative correction, and local inference methods provide options for resource-conscious deployment. The strongest practical workflow combines an appropriate model with deterministic EDA tools and a clearly defined verification strategy. This integrated approach provides a more meaningful basis for evaluating AI-assisted RTL development than model capability or compilation success alone.

10. OPEN CHALLENGES AND RESEARCH DIRECTIONS

1. **Functional Correctness:** Compilation is a necessary but insufficient step in RTL verification. Future systems should integrate automatic testbench generation, assertion synthesis, simulation, and formal equivalence checking more closely. The reliability and independence of LLM-generated testbenches must also be evaluated to ensure that they can detect errors rather than reproduce the assumptions of the generated design [10], [11].
2. **Benchmark Quality:** Larger, more diverse, and contamination-resistant benchmarks are needed to evaluate LLM-based RTL generation reliably. These benchmarks should include hierarchical designs, finite-state machines, communication interfaces, timing constraints, area requirements, and power considerations. Task difficulty, evaluation assumptions, and verification procedures should be clearly documented so that results can be compared fairly across different models [10], [11].
3. **Training Data:** Public Verilog datasets vary in quality, completeness, and licensing conditions. Curated datasets containing verified RTL designs and corresponding testbenches could improve model training and evaluation. Synthetic data may also help increase dataset size, but errors in

generated examples can be propagated during training. In addition, uncommon hardware structures and complex design patterns may remain underrepresented [9], [15], [17].

4. **Small-Model Efficiency:** Quantisation and lightweight inference engines can reduce memory requirements and make local deployment more practical. However, reducing model size may affect code-generation quality, syntax validity, and functional correctness. Future research should evaluate the trade-off between memory consumption, inference latency, energy use, and RTL quality. Resource-constrained deployment should be assessed on clearly specified hardware rather than inferred from model size alone [15], [22]–[26].
5. **Security and Trustworthiness:** LLM-generated RTL may contain unintended behaviour, insecure design choices, or vulnerabilities that are difficult to identify through syntax checking alone. Future systems should incorporate security-oriented design rules, static analysis, assertions, and formal verification where appropriate. The provenance of training data and the handling of confidential design information also require attention, particularly when cloud-hosted models are used [32].
6. **Standardised Resource-Aware Evaluation:** Studies should report model provenance, licence, quantisation method, hardware configuration, prompt and decoding settings, data-handling approach, toolchain versions, testbench provenance, and iteration budget. Correctness metrics should be reported separately, including compilation success, simulation results, and functional verification outcomes. Evaluations should cover both benchmark tasks and carefully specified designs involving sequential behaviour, reset conditions, corner cases, and interface timing. Replication across multiple hardware configurations would help determine whether local-deployment claims generalise beyond a single machine.

Addressing these challenges will require coordinated progress in model development, hardware-aware evaluation, and deterministic verification. Transparent reporting and reproducible experiments can help researchers and practitioners assess the practical capabilities and limitations of LLM-based RTL generation systems.

11. CONCLUSION

Large language models have progressed from general-purpose code generation toward specialised applications in Verilog generation and RTL design. Verilog-specialised models, dedicated benchmarks, feedback-driven frameworks, and efficient inference techniques have expanded the available approaches for AI-assisted hardware development [8]–[18], [33].

Domain adaptation can improve model performance on evaluated RTL-generation tasks, while deterministic compiler and simulator feedback can help identify and correct certain errors [9], [14], [15], [33]. However, the effectiveness of these approaches depends on model capability, verification quality, and evaluation conditions. Quantisation and lightweight inference engines also make local execution increasingly practical. For example, RTLCoder reports a 4-bit quantised model of approximately 4 GB that can operate on a laptop [15].

Despite these developments, functional verification remains a major challenge. Successful compilation does not guarantee that generated RTL implements the intended hardware behaviour. In addition, reporting of resource consumption, privacy, security, and reproducibility remains inconsistent across studies [10], [11], [34], [35].

Future progress will depend on stronger verification methods, more diverse and reliable benchmarks, systematic evaluation of quantised models, secure hardware-auditing techniques, and transparent reporting of experimental results. Integrating LLMs with deterministic EDA tools can help make the design process more reliable, but generated RTL must still be evaluated against its intended specification. With continued research in these areas, LLM-based tools have the potential to become useful assistants for accessible, efficient, and trustworthy RTL design.

REFERENCES

[1] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA, USA: Kluwer, 1984.

[2] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[3] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis,” in *Proc. 43rd ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 532–535.

[4] A. Mirhoseini et al., “A graph placement methodology for fast chip design,” *Nature*, vol. 594, no. 7862, pp. 207–212, Jun. 2021.

[5] M. Chen et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[6] B. Roziere et al., “Code Llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.

[7] B. Hui et al., “Qwen2.5-Coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.

[8] S. Thakur et al., “Benchmarking large language models for automated Verilog RTL code generation,” in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6, doi:10.23919/DATE56975.2023.10137086.

[9] S. Thakur et al., "VeriGen: A large language model for Verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 46:1–46:31, 2024, doi: 10.1145/3643681.

[10] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Invited paper: VerilogEval: Evaluating large language models for Verilog code generation," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–8, doi: 10.1109/ICCAD57390.2023.10323812.

[11] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An open-source benchmark for design RTL generation with large language model," in *Proc. 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 722–727, doi: 10.1109/ASP-DAC58780.2024.10473904.

[12] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," in *Proc. 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, 2023, pp. 1–6, doi: 10.1109/MLCAD58807.2023.10299874.

[13] M. Liu et al., "ChipNeMo: Domain-adapted LLMs for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[14] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "AutoChip: Automating HDL generation using LLM feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[15] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "RTLCoder: Fully open-source and efficient LLM-assisted RTL code generation technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 4, pp. 1448–1461, Apr. 2025, doi: 10.1109/TCAD.2024.3483089.

[16] Y. Fu et al., "GPT4AIGChip: Towards next-generation AI accelerator design automation via large language models," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9, doi: 10.1109/ICCAD57390.2023.10323953.

[17] Z. Pei, H. Zhen, M. Yuan, Y. Huang, and B. Yu, "BetterV: Controlled Verilog generation with discriminative guidance," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, vol. 235, PMLR, 2024, pp. 40145–40153.

[18] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "ChatEDA: A large language model powered autonomous agent for EDA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 3184–3197, Oct. 2024, doi: 10.1109/TCAD.2024.3383347.

[19] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[20] J. Su et al., "RoFormer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, Art. no. 127063, 2024.

[21] J. Ainslie et al., "GQA: Training generalized multi-query transformer models from multi-head checkpoints," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[22] E. Frantar et al., "GPTQ: Accurate post-training quantization for generative pre-trained transformers," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[23] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit matrix multiplication for transformers at scale," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022.

[24] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[25] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

[26] G. Gerganov et al., "llama.cpp: LLM inference in C/C++," GitHub repository. Available: https://github.com/ggml-org/llama.cpp

[27] S. Williams, "Icarus Verilog," GitHub repository. Available: https://github.com/steveicarus/iverilog

[28] C. Wolf and J. Glaser, "Yosys: A free Verilog synthesis suite," in *Proceedings of the Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[29] *IEEE Standard for Verilog Hardware Description Language*, IEEE Std 1364-2005, 2006.

[30] H. Touvron et al., "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[31] D. Shah et al., "Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, 2019.

[32] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[33] J. Blocklove, S. Thakur, B. Tan, H. Pearce, S. Garg, and R. Karri, "Automatically improving LLM-based Verilog generation using EDA tool feedback," *arXiv preprint arXiv:2411.11856*, 2024.

[34] G. Yang et al., "Large language model for Verilog code generation: Literature review and the road ahead," *ACM Computing Surveys*, 2026, doi: 10.1145/3841636.

[35] A. Ravindran, A. Patra, V. Babaey, and S. Purini, "A Critical Review and Evaluation of LLMs for RTL Generation," *IEEE Access*, vol. 14, pp. 28522–28539, 2026, doi: 10.1109/ACCESS.2026.3665894.