

DeepV: A Model-Agnostic Retrieval-Augmented Framework for Verilog Code Generation with a High-Quality Knowledge Base

ZAHIN IBNAT, PAUL E. CALZADA, RASIN MOHAMMED IHTEMAM, SUJAN KUMAR SAHA,
JINGBO ZHOU, FARIMAH FARAHMANDI, and MARK TEHRANIPOOR, University of Florida, USA

As large language models (LLMs) continue to be integrated into modern technology, there has been an increased push towards code generation applications, which also naturally extends to hardware design automation. LLM-based solutions for register transfer level (RTL) code generation for intellectual property (IP) designs have grown, especially with fine-tuned LLMs, prompt engineering, and agentic approaches becoming popular in literature. However, a gap has been exposed in these techniques, as they fail to integrate novel IPs into the model's knowledge base, subsequently resulting in poorly generated code. Additionally, as general-purpose LLMs continue to improve, fine-tuned methods on older models will not be able to compete to produce more accurate and efficient designs. Although some retrieval augmented generation (RAG) techniques exist to mitigate challenges presented in fine-tuning approaches, works tend to leverage low-quality codebases, incorporate computationally expensive fine-tuning in the frameworks, or do not use RAG directly in the RTL generation step. In this work, we introduce *DeepV*: a model-agnostic RAG framework to generate RTL designs by enhancing context through a large, high-quality dataset without any RTL-specific training. Our framework benefits the latest commercial LLM, OpenAI's GPT-5, with a near 17% increase in performance on the VerilogEval benchmark. We host DeepV for use by the community in a Hugging Face (HF) Space: <https://huggingface.co/spaces/FICS-LLM/DeepV>.

CCS Concepts: • **Hardware** → **Hardware description languages and compilation**; • **Computing methodologies** → **Natural language processing**.

Additional Key Words and Phrases: Verilog, Large Language Models, Retrieval Augmented Generation, RTL Generation

1 Introduction

Generative artificial intelligence (GenAI) is a cornerstone of modern technology, with its continued usage expanding into various industry sectors as it further enables automation compared to powerful electronic design automation (EDA) tools. Large language models (LLMs), a category of GenAI, have demonstrated impressive results in code generation with faster design output and streamlined workflows [1, 2].

As a result of these successes, hardware engineers have begun to incorporate LLMs into the hardware design flow, especially for the generation of register transfer level (RTL) code for intellectual properties (IPs) [3]. However, unlike with software code, LLMs generally struggle to generate high-quality RTL code primarily due to 1) limited high-quality RTL code used in training LLMs, 2) a lack of structural understanding, including timing and parallelism, and 3) a limited understanding of physical design considerations. Many works have attempted to improve LLM-based RTL code generation through prompt engineering [4–6], instruction fine-tuning of LLMs [7–14], or agentic approaches [15–25]. However, prompt engineering requires context or examples, despite the rules included, leading to frequent trial-and-error testing [26]. Moreover, instruction-tuned LLMs must optimally receive prompts in a specific format, and updating the approach with new data requires retraining, a computationally expensive and time-consuming process [27]. Similarly, agentic approaches can introduce significant latency and computational cost, as their iterative 'generate-and-verify' cycles require multiple calls to the LLM and external tools for a single output [28].

Authors' Contact Information: Zahin Ibnat, ibnatz16@ufl.edu; Paul E. Calzada, paul.calzada@ufl.edu; Rasin Mohammed Ihatemam, rihtemam@ufl.edu; Sujan Kumar Saha, sujansaha@ufl.edu; Jingbo Zhou, jingbozhou@ufl.edu; Farimah Farahmandi, farimah@ece.ufl.edu; Mark Tehranipoor, tehranipoor@ece.ufl.edu, University of Florida, Gainesville, Florida, USA.

Recently, a push has been made towards applying retrieval-augmented generation (RAG) approaches with fine-tuned models to enhance LLM-guided hardware design automation tasks. RTLRepoCoder [29] utilizes RAG with fine-tuning to enhance repository-level Verilog code completion. AutoVCoder [30] leverages both a knowledge and example retriever in addition to a fine-tuned model to generate RTL code. RTLFixer [31] uses RAG and ReAct prompting to interactively debug code with feedback. As models improve and fine-tuning computational costs increase, there is a need for flexible and cost-effective solutions. We are the first to study RAG’s effectiveness with a large and high-quality dataset, VerilogDB [32], without any fine-tuning, showcasing large accuracy gains that meet and rival fine-tuned RTL generator accuracy. Our key contributions are summarized as follows:

- (1) We introduce *DeepV*: a model-agnostic RTL code generation framework using RAG, where we are the first, to our knowledge, to leverage a dataset of both syntactically correct and synthesizable Verilog codes that has been preprocessed using EDA tools.
- (2) We employ a variety of pre-retrieval optimizations to improve accuracy in addition to implementing dynamic sampling to enhance retriever accuracy and efficiency.
- (3) We analyze our implementation on VerilogEval [33, 34], showcasing our approach’s capability to improve generation accuracy on diverse design problems.
- (4) Additionally, we apply *DeepV* to hierarchical testcases, including FIR filter, Sobel filter, SRNG, and UART, showcasing our tool’s capability in generating complex, multimodule IPs.

Our experiments demonstrate that RAG with VerilogDB [32] enhances the RTL generation capability of the existing models by up to 18.6% and 13% in the *pass@1* and *pass@10* metric respectively, on the VerilogEval [33, 34] benchmark. Furthermore, *DeepV* outperforms the best-performing state-of-the-art fine-tuned solutions by approximately 10%.

2 Background

2.1 Automation of Hardware Design: HLS and its Pitfalls

The challenge of automating hardware design is not new; for decades, the EDA community has attempted to raise the level of abstraction to mitigate the complexities and long development times associated with manual RTL coding. The traditional hardware design flow is a labor-intensive process and represents a significant bottleneck in the development of complex systems-on-chip (SoCs). Before the era of LLMs, the most prominent solution to this challenge was High-Level Synthesis (HLS), which automatically generates RTL code from high-level programming languages like C, C++, or SystemC [35]. The goal of HLS is to bridge the gap between design specification and implementation, which increases productivity and allows designers to explore the design space faster within aggressive time-to-market (TTM) constraints [35].

HLS tools promise to further improve productivity by allowing designers to work at a higher level of abstraction, focusing on algorithmic behavior rather than low-level implementation details. However, despite its adoption in certain domains, HLS has faced persistent challenges that have limited its universal application [36]. Designers often struggle to produce RTL that meets the stringent Power, Performance, and Area (PPA) constraints of their designs without extensive manual code restructuring and pragma insertion [36]. Additionally, the abstraction offered by HLS introduces a new verification challenge: ensuring that the automatically generated RTL is functionally equivalent to the original high-level specification. Moreover, the HLS compilation and optimization process can inadvertently introduce security vulnerabilities, such as information leakage or insecure arbitration, which are not present in the source C/C++ code [37, 38]. These vulnerabilities often arise during HLS stages like scheduling and resource binding, where the

insecure sharing of hardware resources can create side channels or other exploits that compromise security [39, 40]. This issue has led to extensive research focused on the security assessment of HLS tools and the development of security-aware synthesis flows intended to mitigate these risks [40–42].

Beyond these specific concerns, the fundamental limitations of HLS, namely the difficulty in achieving optimal PPA without extensive manual intervention and the rigid requirement for specifications to be written in a formal programming language, have persisted. These challenges highlighted the need for a new paradigm in design automation capable of interpreting high-level intent directly from natural language and using the vast knowledge encoded in existing, human-written RTL designs. The emergence of LLMs, with their powerful capabilities in natural language understanding and pattern recognition in code, provided a promising new direction to address these long-standing goals.

2.2 Foundational Models for Code Generation

The recent advancements in LLM-based RTL generation are built upon a foundation of powerful, general-purpose code intelligence models. These models are based on the Transformer architecture and are pre-trained on vast corpora of publicly available source code from diverse programming languages [43]. This large-scale pre-training gives them a fundamental understanding of programming syntax, structure, and semantics. Pioneering efforts, such as OpenAI’s Codex, the model underlying GitHub Copilot and Meta’s open-source Code Llama, showed that LLMs could achieve great performance on a wide range of code-related tasks, including completion, translation, and generation from natural language prompts [1, 44].

These foundational models provide the basis for the two primary methodologies used to create specialized solutions for Verilog generation:

- **Prompt Engineering:** Prompt engineering involves carefully designing the input prompt given to a pre-trained LLM to guide its output toward the desired result [45]. It treats the LLM as a black box and requires no changes to the model’s underlying weights. The quality of the generated code is highly dependent on how well the prompt is structured for the intended task.
- **Fine-Tuning:** Fine-tuning adapts a pre-trained model to a specific domain by continuing the training process on a smaller, curated dataset [46]. For RTL generation, this involves fine-tuning a base code model on a high-quality corpus of Verilog. This process modifies the model’s weights, embedding domain-specific knowledge directly into the model to improve its accuracy and fluency in the target language. While effective, updating all of a model’s billions of parameters during full fine-tuning is computationally intense. To address this, Parameter-Efficient Fine-Tuning (PEFT) methods have become standard practice, especially within the domain of fine-tuning for code generation [27].

2.3 LLM for RTL Code Generation

The advancements in LLMs have resulted in a significant shift in the EDA sector, providing better automation of complex hardware design tasks to combat aggressive TTM constraints [47]. By using natural language descriptions to generate RTL code, LLMs can shorten design cycles, lower the barrier to entry for hardware development, and enhance designer productivity [48]. The exploration of LLMs for hardware has rapidly evolved throughout the past few years, moving from initial prompting experiments to the development of highly specialized, domain-specific models and agentic frameworks.

Early research primarily focused on evaluating general-purpose LLMs through prompt engineering. Researchers investigated how to formulate natural language specifications to achieve more accurate and functionally correct Verilog outputs from models not explicitly trained for hardware design [49, 50]. These works enabled more complex, automated frameworks that used LLMs as reasoning engines within a larger design loop to automate design processes [15, 18]. While it was proven that prompt engineering can make an impact on the generated RTL code, the early works also showcased the limitations of relying solely on prompting: the process often required extensive trial-and-error, and the models' lack of domain-specific knowledge resulted in frequent syntactic errors.

To overcome these limitations, the focus was shifted toward domain-specific fine-tuning, enabled by the creation of high-quality Verilog datasets. Notable examples of these datasets include RTLLM [51] and OpenLLM-RTL [52], which provide benchmarks and datasets for LLM-aided RTL generation; MG-Verilog [53], which introduced a multi-grained dataset to enhance generation capabilities; CraftRTL [54], which focused on generating high-quality synthetic data; and VerilogDB, the largest and highest-quality dataset for training and evaluation [32].

As discussed, these datasets allowed for a new wave of LLM-based solutions for RTL code with specialized, fine-tuned models. Verigen [7] was an early example, fine-tuned on a curated dataset of Verilog from GitHub repositories and academic materials. RTLCoder [9] presented itself as a lightweight, open-source solution that utilized a specialized dataset to outperform larger, general-purpose models like GPT-3.5 on RTL generation tasks. Further advancements in fine-tuning have incorporated more advanced techniques. For example, BetterV [12] utilizes discriminative guidance and reward models tied to hardware performance metrics to control the generation process, while RTL++ [13] enhances a model's semantic understanding by incorporating control and dataflow graph embeddings into the training process.

Most recently, the field has advanced toward complex, agentic frameworks that contain a multi-step design and verification workflows. These systems are far more advanced than single-shot code generation; instead, they create iterative loops of generation, feedback, and refinement. AutoChip [15] was an early framework that automated HDL generation using iterative feedback from the LLM. OriGen [21] uses a self-reflection mechanism with a dual-model structure, where one model generates code and another corrects it based on compiler feedback. Other systems integrate with standard EDA tools to create verification loops. For example, some frameworks focus on self-verification and self-correction by simulating or synthesizing the generated code and feeding the results back to the LLM for debugging [17]. CoopetitiveV [22] has a multi-agent prompting strategy where different LLM agents collaborate to improve code quality. The development of these advanced systems is supported by a simultaneous effort in benchmarking. Frameworks like VerilogEval [33, 34] and its successors [55] provide standardized problems to measure progress across different models and methodologies, ensuring that the field can systematically evaluate and compare new approaches.

2.4 Retrieval Augmented Generation (RAG)

RAG is a powerful technique for enhancing the accuracy and reliability of LLMs by grounding their responses in external, verifiable knowledge [56, 57]. Instead of relying solely on the parametric knowledge learned during training, RAG dynamically augments relevant, real-time information into the user prompt, providing the LLM with applicable context to make its output. For RTL code generation, this external knowledge consists of high-quality, trusted Verilog designs and associated metadata, which serve as functional examples to guide the generation process [30].

The RAG pipeline works in two phases. First, in an offline indexing stage, a knowledge base of relevant documents, such as verified IP cores, design specifications, or code snippets from repositories like OpenCores [58] and GitHub is prepared. Each document is chunked and converted into a numerical vector representation by an embedding model. These vectors are then stored in a specialized vector database, optimized for efficient similarity search using algorithms

like FAISS [59]. Second, during the online generation phase, a user’s query is also converted into a vector. This query vector is used to search the database and retrieve the most relevant document chunks (e.g., the top-k most similar code examples). These retrieved examples are then augmented with the original user prompt and fed to the LLM, which uses this new context to generate the final RTL output.

Compared to supervised fine-tuning, RAG offers several advantages for domain-specific tasks like RTL generation:

- **Cost-Effectiveness and Adaptability:** Fine-tuning is a computationally expensive process that requires retraining the entire model to incorporate new data [46]. In contrast, a RAG system’s knowledge base can be easily updated by simply adding new documents to the vector database, allowing the system to adapt without modifying the LLM’s weights.
- **Reduced Hallucination:** By grounding the LLM in a corpus of factual, high-quality code examples, RAG mitigates the risk of hallucination, where the model might generate syntactically correct but functionally incorrect or nonsensical Verilog.
- **Traceability and Interpretability:** When a fine-tuned model generates code, its reasoning is generally unknown. A RAG system, however, can provide references to the specific documents it retrieved to generate a response. This traceability is particularly useful in high-assurance applications, as it allows engineers to verify the source of the retrieved code.
- **Dynamic Adaptation:** RAG provides context that is tailored to each specific prompt. This allows for better generalization across a wide variety of design problems, whereas a fine-tuned model’s knowledge is static and fixed at the time of training.

The efficiency of a RAG system for code generation is contingent on three factors: 1) the quality and comprehensiveness of the code corpus, 2) the precision and recall of the document retriever, and 3) the reasoning capability of the generator LLM. While some works have successfully combined RAG with fine-tuning [29], the key question we seek to answer is whether a RAG system with a very high-quality code corpus and a highly precise retriever can generate RTL code as accurately, or more accurately, than state-of-the-art fine-tuned models.

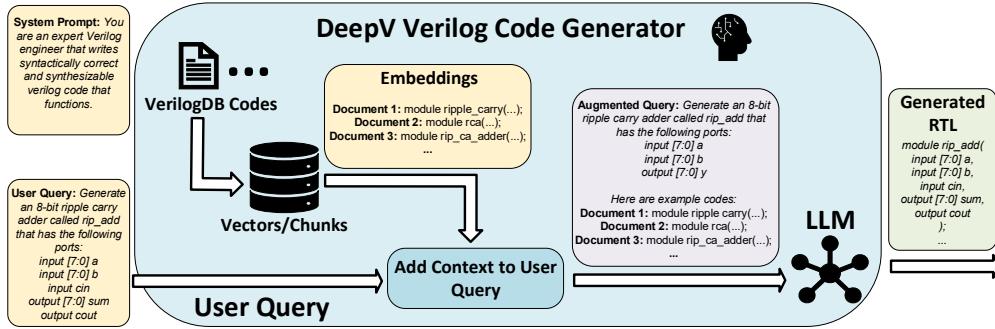


Fig. 1. DeepV Flow highlighting retrieval augmented generation of an example ripple carry adder design.

3 DeepV

To address the challenges of generating accurate Verilog, we introduce *DeepV*, a model-agnostic RAG framework designed to enhance the capabilities of LLMs for RTL design. General-purpose LLMs often struggle with RTL design, frequently hallucinating code that is syntactically invalid or not synthesizable into functional hardware, a limitation the RAG approach is designed to mitigate as discussed in Section 2.4. The primary function of *DeepV* is to aid the chosen LLM with a high-quality knowledge base of existing Verilog designs, providing relevant, in-context examples to guide the code generation process for any given user query. Because *DeepV* is model-agnostic, it does not rely on internal weights or specific API of any single LLM, only requiring the standard API for querying a given LLM with a sufficient context window size. As illustrated in Fig. 1, the *DeepV* pipeline is composed of three stages: 1) Knowledge Base Preparation, 2) Dynamic Context Retrieval, and 3) Augmented Code Generation. First, a comprehensive Verilog example corpus is prepared and vectorized. Second, these vectorized code snippets are stored in an FAISS index to enable dynamic retrieval of the most relevant examples for a user’s prompt. Finally, the retrieved examples are combined with the original query to create an augmented prompt, which is then passed to the LLM to generate the final RTL code. The methodology allows for a given LLM to be consistently provided high-quality, validated examples, allowing in-context learning to generate syntactically correct and functionally verifiable RTL codes.

A comparison of our work and notable academic solutions can be seen in Table 1. Namely, our *DeepV* follows a RAG-based approach without complex fine-tuning, opposite to that of the AutoVCoder [30], ComplexVCoder [24], and RTLRepoCoder [29]. Moreover, unlike every other work in the table, *DeepV* is model-agnostic, meaning that it does not depend on the model parameters of any singular LLM. Lastly, *DeepV*’s knowledge base undergoes post-processing with syntax and synthesis checks in addition to metadata extraction and refinement by OpenAI’s GPT. To showcase these major points, the following subsections will provide a detailed description of each component of this framework.

3.1 Verilog Example Corpus and Vectorization

The performance of a RAG system is fully dependent on the quality and comprehensiveness of its knowledge base. VerilogDB [32] was selected as the Verilog example corpus. VerilogDB provides the largest, high-quality dataset specifically curated for LLM-based RTL generation. It is composed of 20,392 Verilog modules (totaling 751 MB of code) sourced from repositories like GitHub and OpenCores, as well as academic materials, which totaled to be over 30GB of raw Verilog data. The collection of modules has a broad coverage across main RTL design classes, including combinational and sequential logic, custom accelerators, peripheral interfaces, and basic digital building blocks. Additionally, every module in the database has undergone a rigorous preprocessing framework to verify that it is both syntactically correct and synthesizable with standard EDA tools, ensuring the examples reflect industry-ready design practices. Specifically, the process includes syntax validation using the *Icarus Verilog* (*iVerilog*) compiler and logic synthesis checking via *Yosys*, making sure that the database is free of common HDL errors and unsynthesizable constructs often resulting from the accidental inclusion of SystemVerilog constructs in Verilog-2005 code. Furthermore, each module is accompanied by a natural language description, which is necessary for an accurate similarity search against a user’s query.

To create a well-detailed representation for each design, a single document is constructed for every module in the database, as seen in Fig. 2. This document is not just the raw code; it is a structured text block that begins with header information (including the module’s name, its natural language description, a detailed list of its ports, and all original code comments). Structuring the document with this descriptive metadata and natural language first allows for the maximizing of the semantic relevance captured by the embedding model’s attention window and improves

Table 1. Comparison of RTL Code Generation Methods

Method	Dataset	RAG	Fine-Tuned	Base Model(s)
Fine-Tuning Dominant Approaches				
Verigen [7]	Source: GitHub, textbook Process: Filtered	✗	✓	CodeGen-16B
RTLCoder [9]	Size: 27k; Source: Synthetic Process: Syntax checked	✗	✓	Mistral-7B, DeepSeek-6.7B
Origen [21]	Size: 220k; Source: Synthetic Process: Syntax checked	✗	✓	DeepSeek-7B
CodeV [60]	Size: 165k; Source: GitHub Process: Syntax checked	✗	✓	CodeLlama-7B+, Qwen1.5-7B
VeriCoder	Size: 125k+; Source: Synthetic Process: Functionally validated	✗	✓	Qwen, DeepSeek
CraftRTL [54]	Size: 86k+; Source: Synthetic, GitHub Process: Syntax checked	✗	✓	CodeLlama-7B, Starcoder2-15B
BetterV [12]	Size: 68k; Source: GitHub Process: Syntax checked	✗	✓	CodeLlama-7B, DeepSeek-6.7B, Qwen1.5-7B
RTL++ [13]	Size: 200k+; Source: GitHub+ Process: GPT-refined, synthesis checked	✗	✓	CodeLlama-7B
RAG and Hybrid Approaches				
AutoVCoder [30]	Source: Synthetic, GitHub Process: Scoring filtered	✓	✓	CodeLlama-7B, DeepSeek-6.7B, Qwen1.5-7B
ComplexVCoder [24]	Size: 12.5k RAG base Source: GitHub, Pyranet [61]	✓	✓	DeepSeek-V3, GPT-4o, Qwen2.5
RTLRepoCoder [29]	Source: Repository-level context Process: Cross-file parsing	✓	✓	Code Llama, StarCoder
Agentic and Feedback-Driven Approaches (No Fine-Tuning)				
AutoChip [15]	Source: N/A (Feedback-based)	✗	✗	GPT (code-davinci-002)
RTL-Assistant [17]	Source: N/A (Feedback-based)	✗	✗	GPT-4, GPT-3.5
VeriThoughts [14]	Source: N/A (Feedback-based)	✗	✗	GPT-4
VeriMind	Source: N/A (Agentic reasoning)	✗	✗	GPT-4 (or similar)
CoopetitiveV [22]	Source: N/A (Prompting strategy)	✗	✗	GPT-4, Claude 3, Gemini 1.5
DeepV	Size: 20k+; Source: GitHub, OpenCores, textbook Process: GPT-refined, Syntax + synthesis checked	✓	✗	Model Agnostic

query-to-document correlation by aligning with the natural language format of the user’s query. This header, formatted as Verilog comments, is followed by the full Verilog code of the module within the dedicated JSON for each document. This method makes it so that each document contains the complete context, both structural and descriptive, of a single hardware module. Adding this metadata required little overhead as the corpus is primarily composed of small to moderate designs where the majority of modules contained fewer than 100 lines of code, which is ideal for the RAG task as the examples are self-contained, high-quality, and computationally manageable for in-context learning.

To handle the scale of the entire 20,392-module corpus, the documents are processed in sequential, memory-efficient batches. Each structured document within a batch is then converted into a high-dimensional vector using the all-MiniLM-L6-v2 embedding model. This is a highly efficient model from the Sentence-Transformers framework, which uses knowledge distillation to create a small (6-layer) but powerful model optimized for generating semantically meaningful document embeddings [62]. The all-MiniLM-L6-v2 model outputs a 384-dimensional vector. This size is a balance between retrieval accuracy and memory footprint, allowing for low-latency inference and minimizing vectorization overhead, which prevents a latency bottleneck for the real-time RAG query. The final output of this stage is a complete vector representation of the VerilogDB corpus, where each vector captures the rich semantic meaning of an individual hardware module, making it ready for indexing and retrieval.

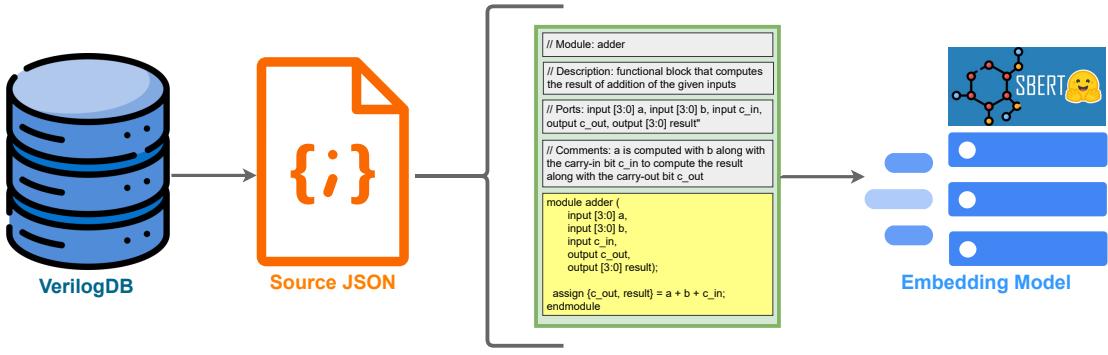


Fig. 2. Document Construction for Embedding

3.2 FAISS Storage and Dynamic Retrieval

Once the database is vectorized, the resulting high-dimensional vectors are stored in a Facebook AI Similarity Search (FAISS) index. FAISS is an open-source library specifically designed for rapid similarity search in massive, high-dimensional datasets [63]. It allows for the comparison of a user's query against all 20,392 document vectors in real-time. At initialization, the system loads this pre-built FAISS index into memory, preparing the retrieval for incoming user queries. Given the current corpus size of 20,392 modules, an exact search approach using the `IndexFlatL2` structure was selected via the LangChain FAISS wrapper, guaranteeing 100% retrieval accuracy without the speed-vs-recall trade-offs associated with approximate nearest neighbor indices like InVverted File Flat (IVFFlat) or hierarchical navigable small worlds (HNSW).

The retrieval of relevant examples is not a simple top-k lookup; it is a multi-stage dynamic retrieval process that finds the most contextually valuable documents for prompt augmentation that is unique to each query. In short, this process acts as a sophisticated filter. It is initiated when the user's natural language query is converted into a high-dimensional vector using the same embedding model that was used for the corpus. This query vector is then used to perform a similarity search against the FAISS index.

The system calculates the Euclidean Distance (L2) between the query vector and each document vector, which measures the straight-line distance between them in the embedding space; a lower distance indicates a closer match [64]. While Cosine Similarity is commonly used for directional alignment in vector spaces, L2 distance was chosen as the

underlying measure, which is mathematically equivalent to Cosine Similarity for normalized vectors, while providing a direct, un-normalized distance metric for subsequent thresholding purposes. The LangChain framework then converts these distance measurements into a normalized relevance score, where a higher value signifies greater similarity, to provide a consistent metric for the subsequent filtering stages [65]. This search retrieves an initial candidate pool of the 10 documents with the highest converted relevance scores. Following this initial retrieval, the system applies a filter on the candidate pool based on an absolute relevance score. A minimum score threshold is enforced, and any document with a relevance score below 0.55 is discarded from the pool, ensuring that only documents with a strong similarity to the user’s request are considered for the final context.

The final stage of the retrieval process is a dynamic sampling method designed to adapt the number of examples to the specific nature of the user’s query. This process can be seen in Alg. 1, where instead of using a fixed number of documents, the system analyzes the rate of decrease in relevance scores among the sorted candidates. It identifies the point of diminishing returns by calculating the drop in score between each consecutively ranked document. If the score drop between two documents is more than 1.5 times larger than the drop between the previous two, the system halts the selection process. The specific values of the 0.55 relevance threshold and the $1.5 \times$ dynamic drop factor were tuned on a validation set to optimize for both context quality and LLM token budget. This heuristic allows the context to be flexible; for queries with many highly relevant examples, more documents may be included, while for queries where relevance drops off sharply, the context will be more focused. As a final safeguard to manage the context window size and associated computational costs, the number of documents selected by this dynamic process is ultimately capped at a maximum of five examples. This multi-stage retrieval strategy ensures that the context provided to the LLM is not only highly relevant but also dynamically sized to best match the complexity of each user query.

The usage of this algorithm as well as specific val for each variable in Alg. 1 can be seen in Section 4. As a point of discussion, it can be stated that the dynamic sampling algorithm is dependent upon the FAISS system’s scoring, which is as stated, the Euclidean distance between the query vector and each document vector. While this is a consistent metric and a simple scoring method, it can prove to be unreliable in some cases. Not all prompts will be set up in a way that matches the document’s metadata for the description of the IP. While the default scoring system from FAISS has proven to be reliable in majority of cases, other scoring metrics can be evaluated in subsequent works for performance gains. In particular, future work could explore implementing the IVFFlat to scale to larger corpora beyond 100,000 examples, accepting a minor trade-off in retrieval latency for massive dataset management.

3.3 RTL Code Generation

With the FAISS index constructed with an algorithm for a dynamic sampling RAG, the final stage of the *DeepV* framework is the generation of RTL code. The backend for the framework is an LLM of the user’s choosing, as our methodology is model-agnostic. Additionally, the dynamic retriever works alongside meticulous prompt engineering that is designed to constrain the LLM’s output to adhere to strict hardware design standards. In the case of a commercial model such as GPT-4.1, the system prompt first assigns a role to the model to act as an expert Verilog engineer who adheres to strict rules for the generation task. These include, but are not limited to:

- You must produce fully implemented, accurate Verilog-2005 code.
- You must not use placeholders or incomplete modules.
- You cannot nest a module inside another module.

Algorithm 1: Dynamic Sampling for RAG Context

Require:

- $D = \{(d_1, s_1), \dots, (d_n, s_n)\}$: List of candidates sorted by relevance score ($s_1 \geq s_2 \geq \dots$).
- τ : The minimum relevance score threshold.
- k_{max} : The maximum number of documents to select (e.g., 5).
- α : The score drop-off factor (e.g., 1.5).

Result: C : The final list of context documents.

```

1: //1. Filter documents below the minimum threshold
2:  $D' \leftarrow \{(d, s) \in D \mid s \geq \tau\}$ ;
3: //2. Initialize empty context
4:  $C \leftarrow \emptyset$ ;
5: if  $|D'| \geq 1$  then
6:   //Always include the most relevant document
7:    $C \leftarrow C \cup \{d'_1\}$ 
8: if  $|D'| \geq 2$  and  $k_{max} \geq 2$  then
9:   //Include the second document to establish the initial drop
10:   $C \leftarrow C \cup \{d'_2\}$ ;
11:  //Calculate the first score drop
12:   $\Delta_{prev} \leftarrow s'_1 - s'_2$ ;
13:  for  $i \leftarrow 3$  to  $\min(|D'|, k_{max})$  do
14:    //Calculate the current score drop
15:     $\Delta_{curr} \leftarrow s'_{i-1} - s'_i$ ;
16:    if  $\Delta_{curr} > \alpha \cdot \Delta_{prev}$  then
17:      //Halt if relevance drops off sharply
18:      break;
19:    //Add the current document to the context
20:     $C \leftarrow C \cup \{d'_i\}$ ;
21:    //Update previous drop for the next iteration
22:     $\Delta_{prev} \leftarrow \Delta_{curr}$ ;
23: return  $C$ 

```

Following the system prompt, the user prompt combines the retrieved documents with the original query. The complete augmented prompt is constructed as a three-part sequence: 1) the System Role and Constraints, 2) the Contextual Code Examples (retrieved by RAG), and 3) the User's Natural Language Request. The Verilog modules selected by the dynamic retrieval process are augmented into the query as contextual examples, which can add to expected coding style and structure and the intended function of the requested module. The complete prompt applies in-context learning, where the LLM uses the provided high-quality documents to inform its generation of the new module. Then, the structured prompt is sent to the LLM via an API endpoint. After the model generates a response, a final post-processing step is applied to ensure the output is clean and directly usable. This step uses regular expressions to parse the raw text and extract only the Verilog code from the expected markdown block, isolating it from any extraneous text the model might produce. This extracted code is the final, ready-to-use RTL design produced by the framework.

4 Evaluation

To cover a wide range of designs as well as provide quantitative analysis for *DeepV*'s performance, the evaluation for the framework utilized the VerilogEval benchmark [33, 34]. The performance of this work rivals that of current LLM-based solutions for Verilog code generation. Although the benchmark provides an adequate analysis for *pass@k* metrics, it does not cover multimodule designs; therefore, we created a small benchmark of complex designs to test with *DeepV*, showcasing this work's application for complex, hierarchical designs in Section 4.2.

4.1 Quantitative Analysis on VerilogEval

4.1.1 Experiment Settings. To quantitatively evaluate our framework, we conducted a large-scale experiment using the VerilogEval benchmark suite [33, 34]. VerilogEval is the standard benchmark for RTL code generation, comprising 156 human-generated problems of varying complexity that require the generation of a functionally correct RTL module from a natural language specification. To assess performance across different model architectures and sizes, a variety of back-end large language models were evaluated. These included open-source models, such as Mistral-7B-Instruct and CodeLlama-7B-Instruct, as well as a range of proprietary models from OpenAI, e.g., GPT-5 Chat. The retrieval component of our framework utilized the all-MiniLM-L6-v2 sentence transformers model for generating embeddings.

All code generation calls to the LLM were made with consistent inference parameters to ensure fair comparison across models and configurations. The temperature was set to 0.8 to encourage a degree of creativity while maintaining structural coherence, and top-p was set to 0.95 for nucleus sampling. A maximum of 1500 new tokens was allocated for each generation to accommodate complex modules. Two configurations were also evaluated to measure the direct impact of our RAG approach: a *DeepV* configuration with the full retrieval pipeline active, and a Baseline (Zero-shot) configuration where the retrieval mechanism was disabled along with lessened prompt engineering.

The evaluation on the VerilogEval [33, 34] was performed in two sequential stages: **syntax check** and **functional verification**. During syntax validation, each module generated by a model was compiled using the *Icarus Verilog* (i.e., *iVerilog*) tool [66]. Afterwards, the module was tagged as *Compiled* or *Not Compiled*, depending on the outcome of the compilation. In the next step, syntactically correct modules were simulated with Verilator using the benchmark-provided testbenches. Next, modules were labeled as "Passed" only if the simulated results had zero mismatches with the reference solutions defined by the benchmark.

We carried out the quantitative assessment through the widely-used *pass@k* [1] metric. The metric provides the probability of a code solution passing validation under k independent generations. The calculation is based on Eq. 1.

$$\text{pass}@k = \mathbb{E} \left(1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right) \quad (1)$$

where n is the number of iterations that a certain code generation has gone through, and c is the number of codes that passed validation.

4.1.2 Analysis of RAG Improvement on LLMs. To gauge the performance of *DeepV*, we applied the framework to multiple models, both open source and proprietary. We observed the performance gains of our framework, backed by *VerilogDB* [32], compared to the baseline models to which we applied *DeepV* to. Table 2 and Table 3 summarize our findings on syntax correctness and functional accuracy of the LLM-generated modules from VerilogEval [33, 34].

With a very high-quality knowledge base provided to *DeepV* models, great improvement can be seen in the semantic understanding of Verilog code. We evaluate syntax accuracy for baseline commercial and open source LLMs and showcase the improvement seen with our approach. As can be seen in Table 2, each LLM benefited from the context provided by additional documents with regard to syntax. GPT-5 Chat was augmented the most with an increased syntax accuracy on VerilogEval of almost 24% with only one document retrieved. Also, 100% syntax accuracy for *pass@5* was achieved with only one document retrieved. GPT-4o also augmented its accuracy on the benchmark from 74.1% to 97.6% with one document retrieved. Open source models saw the largest increases on three documents of around 8% for CodeLlama 7B-Instruct and 11% for Mistral 7B-Instruct from their baselines.

DeepV's ability to enhance LLM's generation of functionally accurate RTL code is evident from Table 3. *DeepV* improves the output of GPT-5 (Chat-Latest), OpenAI's latest model, by 16% on *pass@1*, and 13% on *pass@10*. Moreover, GPT-4o and GPT-4.1 obtain 8.7% and 8.9% improvement on *pass@1*, as well as 5.8% and 5.1% on *pass@10*. Furthermore, Claude Sonnet 4 sees an increase of almost 10% and 4% on *pass@1* and *pass@10*. *DeepV* also uplifts Gemini 2.5 Flash's results by 18.5% and 7.1% on *pass@1* and *pass@10*. Additionally, smaller open-source models such as Mistral-7B-Instruct had a 15.5% improvement on *pass@1* and CodeLlama-7B-Instact results increase by 9.6% on *pass@10*.

Table 2. Syntax Correctness Comparison between Baseline Models and Corresponding *DeepV* Approach on VerilogEval

Model	Open Source	Baseline			DeepV			Improvement		
		<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>	<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>	<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>
Mistral-7B-Instruct	✓	25.3	64.0	77.6	36.1	69.6	81.4	10.8	5.6	3.8
CodeLlama-7B-Instruct	✓	29.6	66.5	82.1	37.9	77.4	89.1	8.3	10.9	7.1
GPT-4o	✗	74.1	76.7	77.6	97.9	98.7	98.7	23.8	21.9	21.2
GPT-4.1	✗	74.4	86.7	89.7	98.1	100	100	23.7	13.3	10.3
GPT-5 Chat	✗	73.7	82.1	84.0	99.4	100	100	25.6	17.9	16.0
Claude Sonnet 4	✗	82.1	90.5	92.9	99.4	100	100	17.3	9.5	7.1
Gemini 2.5 Flash	✗	69.9	88.9	94.9	91.7	98.4	99.4	21.8	9.5	4.5

Table 3. Functional Correctness Comparison Between Baseline Models and Corresponding *DeepV* Approach on VerilogEval

Model	Open Source	Baseline			DeepV			Improvement		
		<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>	<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>	<i>pass@1</i>	<i>pass@5</i>	<i>pass@10</i>
Mistral-7B-Instruct	✓	7.1	12.3	14.7	22.6	25.7	26.3	15.5	13.4	11.5
CodeLlama-7B-Instruct	✓	14.2	17.0	17.9	19.0	25.3	27.6	4.8	8.2	9.6
GPT-4o	✗	59.0	62.1	63.5	67.7	68.9	69.2	8.7	6.8	5.8
GPT-4.1	✗	61.0	70.4	74.4	69.9	76.9	79.5	8.9	6.6	5.1
GPT-5 Chat	✗	60.9	68.8	69.9	76.9	81.3	83.3	16.0	12.4	13.5
Claude Sonnet 4	✗	66.0	75.5	78.2	75.6	80.6	82.1	9.6	5.1	3.8
Gemini 2.5 Flash	✗	53.2	69.2	73.7	71.8	78.4	80.8	18.6	9.2	7.1

4.1.3 *Comparison with SOTA Techniques.* Comparing the results of our framework to baseline models reveals a significant improvement in the performance of the LLM; however, it is also important to assess how effective our technique is in comparison to other state-of-the-art techniques. Table 4 depicts this comparison, where *DeepV* outperforms the latest solutions in all metrics for the VerilogEval benchmark. *DeepV* achieved a 76.9% *pass@1* value, outscaling the

previously best scor, CraftRTL [54], by 8.9%. Moreover, *DeepV* achieves an 81.3% on *pass@5*, surpassing Veriseek [10] by 4.4%. Furthermore, our framework surpasses Veriseek [10] by 1.6%, receiving an 83.3% on *pass@10*.

Table 4. Performance Comparison of Functional Correctness on VerilogEval between Baseline Models, RTL Specific Solutions, and *DeepV*. Results are reported for *pass@1* at temperature = 0.2, and *pass@5* and *pass@10* at temperature = 0.8

Category	Model / Method [Base Model]	Open Source	pass@1	pass@5	pass@10
<i>General Purpose</i>	Mistral-7B-Instruct	✓	7.1	12.3	14.7
	CodeLLaMa-7B-Instruct	✓	14.2	17.0	17.9
<i>Commercial</i>	GPT-4o	✗	59.0	62.1	63.5
	GPT-4.1	✗	61.0	70.4	74.4
	GPT-5 Chat	✗	60.9	68.8	69.9
<i>RTL Specific</i>	RTLCoder [DeepSeek-Coder-6.7B] [9]	✓	41.6	50.1	53.4
	CodeV [Qwen-Coder] [60]	✓	59.2	65.8	69.1
	Verigen [CodeLlama-7B] [7]	✓	30.3	43.9	49.6
	Origen [DeepSeek-Coder-7B] [21]	✓	54.4	60.1	64.2
	RTL++ [CodeLlama-7B] [13]	✓	59.9	68.8	72.1
	BetterV [CodeQwen-7B] [12]	✗	46.1	53.7	58.2
	CraftRTL [StarCoder2-15B] [54]	✗	68.0	72.4	74.6
	VeriThoughts [Qwen-2.5-Instruct-14B] [14]	✓	43.7	52.2	55.1
	ReasoningV [DeepSeek-Coder-6.7B] [25]	✓	57.8	69.3	72.4
	Veriseek [DeepSeek-Coder-6.7B] [10]	✓	61.6	76.9	81.7
	ITERTL [DeepSeekV2-7k] [11]	✓	53.8	60.8	64.1
	AutoVCoder [CodeQwen-7B] [30]	✓	48.5	55.9	-
<i>DeepV</i>	DeepV[Mistral-7B-Instruct]	✗	22.6	25.7	26.3
	DeepV[CodeLLaMa-7B-Instruct]	✗	19.0	25.3	27.6
	DeepV[GPT-4o]	✗	67.7	68.9	69.2
	DeepV[GPT-4.1]	✗	69.9	76.9	79.5
	DeepV[GPT-5 Chat]	✗	76.9	81.3	83.3
	DeepV[Claude Sonnet 4]	✗	75.6	80.6	82.1
	DeepV[Gemini 2.5 flash]	✗	71.8	78.4	80.8

4.1.4 Ablation Study - RAG Configurations. To assess the additional context provided by *DeepV*'s retrieved documents, we experimented with the number of documents retrieved by our RAG approach. During the initial phase, the understanding was that an increase in the number of retrieved documents would benefit LLMs by providing further context. This understanding was supported by the smaller open-source models' outputs. As shown in Fig. 3, both Mistral-7B-Instruct and CodeLLaMa-7B-Instruct gain approximately 9% higher performance at the *pass@10* metric for generating syntactically correct code, when three documents are retrieved over one document retrieval. Moreover, Mistral-7B-instruct's ability to produce functionally accurate documents improves by 8% on *pass@1*, as evident in Fig. 3b. However, the GPT-4o results shown in Fig. 3e and Fig. 3f illustrate that the one-document retrieval performs almost equally to the three-document retrieval.

This deviation from our initial findings compelled us to dig deeper into the retrieval system of the RAG. As a result, we applied dynamic sampling on the database with our RAG approach based on Algorithm 1. We compared the dynamic retrieval system with only one-document retrieval. The results are depicted in figure 4. The figures show that additional context over one-example retrieval does not necessarily benefit these models, but it helps for consistency to have a systematic way to organize the documents chosen to be augmented to each prompt, as opposed to having a hard-coded number of documents. As stated in Section 3.2, dynamic sampling has a dependency on the scoring system used for document and query vector comparison. Perhaps, only the first document provided enough context for most prompts, while a small margin of prompts required multiple documents; therefore, the 1 document context performed the same as the dynamic sampling. It can also be noted that LLMs typically have a maximum token limit for queries. In the case that the documents retrieved are lengthy codes, the full augmented query can be cut off. This can also apply to the previous ablation study results as seen in Fig. 3.

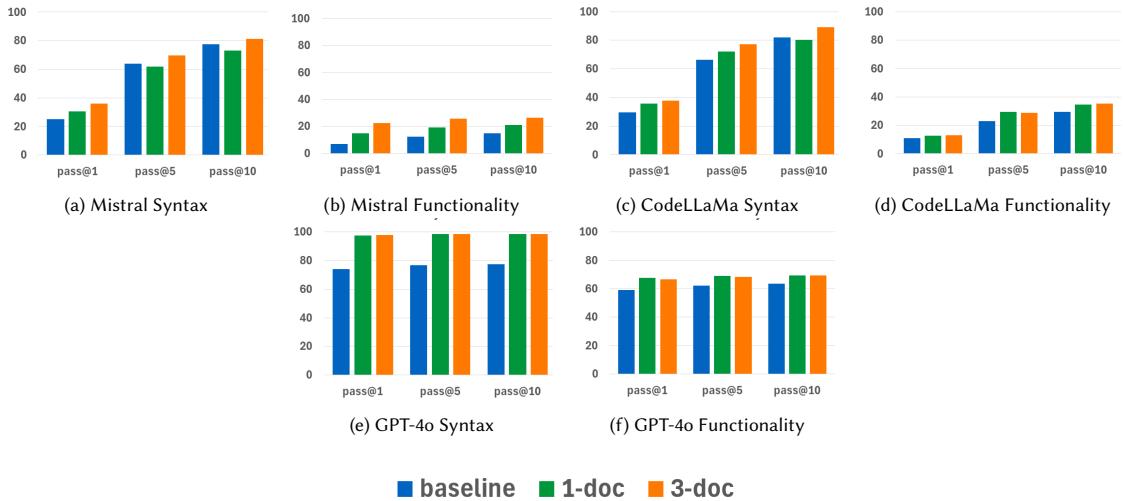
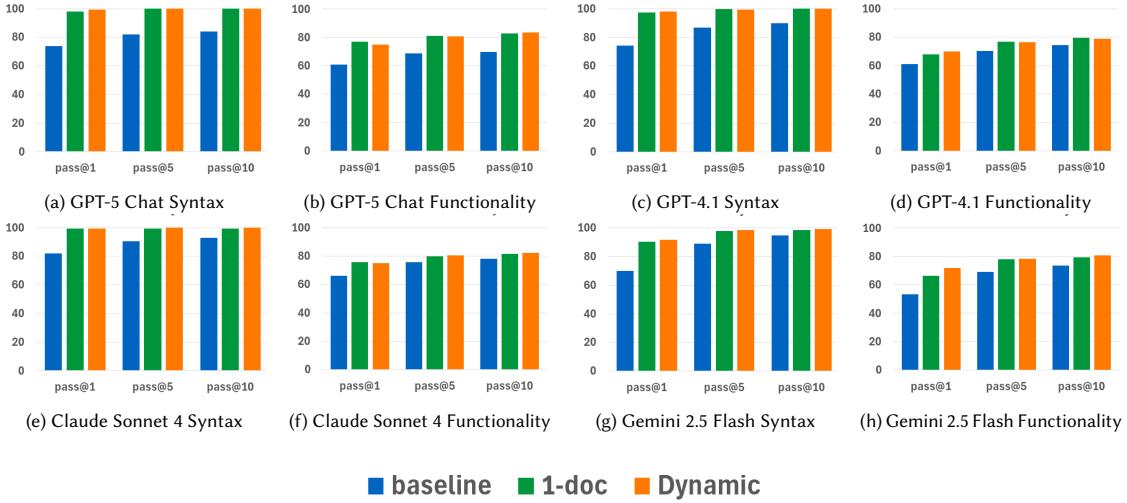


Fig. 3. Comparison of Baseline Models with 1 Document and 3 Documents Retrieval by *DeepV*. (Note: Mistral and CodeLLaMa models used are 7B Instruct)

4.2 Case Studies: Complex Hierarchical IPs

Despite the widely used benchmark, VerilogEval [33, 34], having a variety of different design types and complexities, it does not have the prompts to test an LLM’s capability to generate complex and hierarchical designs. To showcase that *DeepV* can improve RTL generation of large hierarchical designs, we test a baseline GPT-5 Chat on four different hierarchical designs, which range in complexity but are multinode. We then tested GPT-5 Chat with *DeepV* for both 1-doc and dynamic sampling to assess the improvement on these specific complex cases. To configure GPT-5 Chat, a temperature of 1, top-p of 0.95, and max tokens of 10,000 were selected.

The four designs include a finite impulse response (FIR) filter used commonly in digital signal processing (DSP) applications to remove unwanted frequencies, a Sobel filter image processing algorithm used to detect edges, a secure random number generator (SRNG) used to create a cryptographically secure sequence of random bits, and a Universal

Fig. 4. Comparison of LLM Performance between 1 Document Retrieval and Dynamic Sampling by *DeepV*

Asynchronous Receiver/Transmitter (UART) module commonly used for system-level communication. The selected case study problems are listed in Table 5, where it can be observed that the designs span different hierarchy depths, number of total modules, and application domains. We assess the application domains in order to test *DeepV*'s capability to pull domain-specific information covering different use cases. The diversity in application domain is critical, because each domain possesses a unique set of constraints and implementation nuances; The knowledge required to correctly implement the timing and state logic of the UART protocol differs greatly from the algorithmic structure needed for a FIR filter or the entropic properties of a Secure RNG. By evaluating increased functional accuracy using *DeepV* across all four varied designs, we highlight that *DeepV*'s knowledge base is not narrowly specialized but is comprehensive and versatile. To summarize, these case studies demonstrate some of *DeepV*'s key benefits:

- (1) its ability to retrieve and apply correct, domain-specific context, proving its efficacy on a spectrum of real-world hardware engineering tasks.
- (2) its ability to improve generation of complex, hierarchical IPs.
- (3) its ability to adhere to formal specifications in prompts despite variations in context from the knowledge base.

Table 5. Designs selected for Case Study

ID	Design	Application Domain	Hierarchy Depth	# Modules	Generated Avg. Line Count
D1	FIR Filter	Digital Signal Processing	Two	10	202
D2	Sobel Filter	Image Processing - Edge Detection	Two	4	171
D3	UART	Communication/Interface	Three	5	285
D4	Secure RNG	Hardware Security	Four	5	154

4.2.1 Case Study Setup and Execution. To establish an unambiguous problem definition for the LLM, we first prepared detailed prompts for each design. Each prompt has a specified port map, core functionality, and the intended module hierarchy, including the names and depth of all sub-modules. This level of detail was crucial for guiding the LLM towards

a structurally sound solution. Furthermore, design constraints were well-defined, e.g., for protocol-heavy modules like the UART, this included precise interface timing specifications and operating frequencies. To create a supporting validation framework, a functional testbench was developed for each prompt. This testbench served as the ground truth, enforcing the requirements laid out in the prompt and testing the design-under-test (DUT) for functional correctness. For each case study, we generated 10 design iterations to get a range of responses for the given prompt. Finally, each generated sample was compiled and simulated using *iVerilog* [66], with a design deemed successful only if it compiled cleanly and passed all testbench assertions. Using this framework, we evaluated several configurations. For our baseline, we used GPT-5 Chat to generate 10 designs for each case study. We then applied *DeepV* to generate an additional 10 designs per problem under two distinct retrieval settings: single-document (1-doc) and dynamic sampling. We define functional accuracy as the percentage of the 10 designs in a given set that successfully pass all simulation-based functional tests.

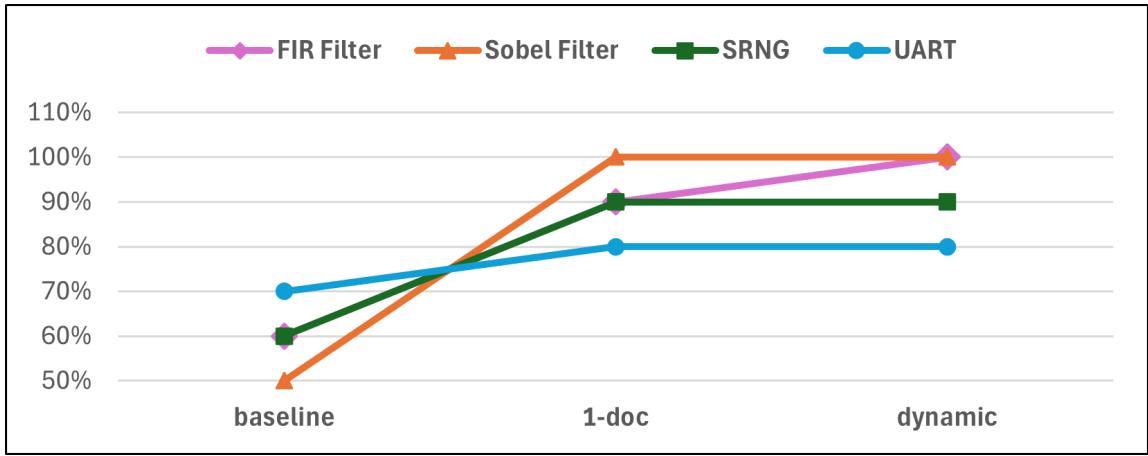


Fig. 5. Case study functional accuracy across baseline (Gpt-5-chat-latest) and *DeepV* configured for 1-doc and dynamic.

4.2.2 Experiment Results. The application of *DeepV* resulted in significant improvements in functional correctness across all case studies. The most substantial gain was observed on the Sobel filter problem, which saw a 50% accuracy increase with just 1-doc retrieval. Similarly, both the FIR filter and the SRNG experienced 30% accuracy boosts from the baseline in the 1-doc setting. Notably, the FIR filter's accuracy was further elevated to a perfect 100% when using dynamic sampling. The UART test case, while showing the smallest gain, still saw a total enhancement of 10%.

An interesting finding emerged when comparing the 1-doc and dynamic sampling results. For three of the four problems, the functional accuracy remained the same between the two settings. We hypothesize this is because the first retrieved document had a significantly higher similarity score than all subsequent candidates or other factors as mentioned in Section 4.1.4, causing the dynamic sampling heuristic to be equivalent to that of the 1-doc results. In contrast, the FIR filter clearly benefited from the dynamic approach, as the retriever found multiple highly-scored documents that provided a richer context. Ultimately, these results demonstrate that *DeepV* can substantially enhance an LLM's ability to generate functionally correct hierarchical designs, where even a single relevant document is often sufficient to see substantial accuracy gains.

5 Discussion

5.1 Accessibility and Ease-of-Use

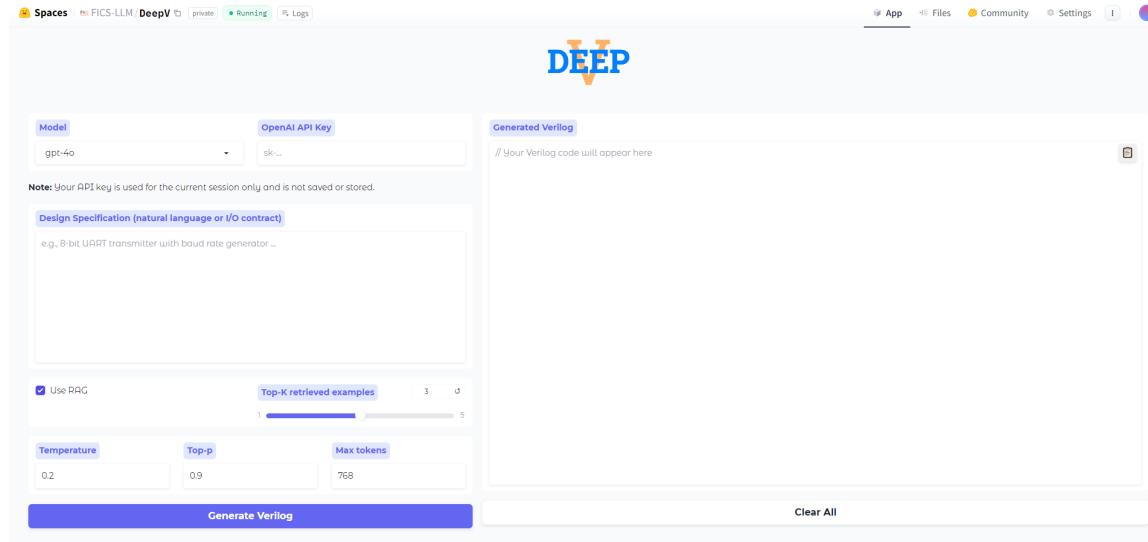


Fig. 6. DeepV Hugging Face Space Application

To allow the community to access *DeepV*, we have integrated *DeepV* into a Hugging Face (HF) Space: <https://huggingface.co/spaces/FICS-LLM/DeepV>.

Our *DeepV* HF Space provides a web-based interface requiring no installations or specialized hardware. This removes barriers to experimentation and supports reproducible research. Firstly, the application allows the user to select an OpenAI model (GPT-4o, GPT-4.1, and GPT-5 Chat) and apply their OpenAI API key. We note the inputted API key is only active for the current session and is not saved or stored due to security concerns. Secondly, the user can input a design specification in an open format, unlike with fine-tuned LLMs, which each require specific input format requirements. Thirdly, the application allows for *DeepV* configuration options like enabling RAG, number of retrieved documents, and LLM parameters, including temperature, top-p, and max tokens. As *DeepV* is hosted on HF Spaces as a Gradio application, the Gradio client Python package can be run locally to interact with the application in an automated fashion. Due to this, *DeepV* can be integrated into larger systems or agentic workflows. A tutorial API access Python script is provided in the *DeepV* HF Space repository.

5.2 Key Takeaways

The results presented in this work offer several key takeaways for the field of LLM-aided hardware design.

Key Takeaway T_1 :

A high-quality knowledge base is as critical as model architecture; RAG can elevate general-purpose LLMs to rival specialized, fine-tuned models.

Primarily, our findings demonstrate that a meticulously curated knowledge base combined with a sophisticated RAG framework can elevate the performance of general-purpose LLMs to a level competitive with, and in some cases superior to, highly specialized, fine-tuned models as seen in Table 4. This suggests that for many RTL generation tasks, the bottleneck is not the model’s inherent reasoning capability but its access to relevant, high-quality domain knowledge. *DeepV*’s success, particularly with the latest GPT-5 Chat model, shows a potential path where continuous knowledge base refinement becomes as important as advancements in model architecture itself.

Key Takeaway T_2 :

Contextual needs are query-dependent; while a single example is often powerful, dynamic, query-aware retrieval is crucial for maximizing accuracy on complex designs.

Furthermore, the comparison between 1-doc and dynamic sampling reveals nuances in how LLMs utilize context. While a single, highly relevant example provides a significant performance boost, dynamic sampling observed in several test cases suggests that more context is not always better context. The effectiveness of dynamic sampling on the FIR filter, however, proves that for certain complex problems, providing a wide range of context is critical for achieving high functional accuracy. This highlights the need for intelligent, query-aware retrieval strategies over fixed-context approaches, which adapt to the specific demands of each query.

Key Takeaway T_3 :

The effectiveness of RAG in hardware design is fundamentally tied to the quality of its knowledge base. Using syntactically correct, synthesizable code is highly important for generating reliable RTL.

A central pillar of the *DeepV* framework is its reliance on a meticulously curated knowledge base, which is a critical factor in its success. The framework utilizes *VerilogDB*, a dataset composed of over 20,000 Verilog modules sourced from reliable repositories like GitHub and OpenCores, as well as academic materials [32]. Unlike RAG systems that may pull from unvetted sources, every module in *VerilogDB* has undergone a rigorous preprocessing framework to confirm that it is both syntactically correct and synthesizable with standard EDA tools. This pre-verification makes it possible for the LLM to be grounded in industry-ready, error-free examples. Furthermore, the vectorization process creates a structured document for each module, combining the full Verilog code with crucial metadata such as natural language descriptions and detailed port lists, such that the retriever has rich semantic context for every search.

Key Takeaway T_4 :

A model-agnostic design, coupled with public deployment via a user-friendly interface, is important for driving adoption, reproducibility, and integration of new research into practical workflows.

To maximize impact and utility, *DeepV* was intentionally designed as a model-agnostic RAG framework that does not depend on a specific, fine-tuned model. This flexibility allows the framework to continuously benefit from the rapid advancements in general-purpose commercial LLMs without requiring costly retraining. To further promote adoption and reproducible research, *DeepV* has been integrated into a publicly accessible HF Space, providing a web-based interface that requires no installations or specialized hardware from the user. This accessibility lowers the barrier to

entry for experimentation and allows for seamless integration into larger systems or agentic workflows through its API. By providing an open and easy-to-use tool, we encourage community engagement and build a foundation for future RAG-centric research in hardware design.

5.3 Future Work

The success of the *DeepV* framework in generating complex, hierarchical designs opens up several avenues for future research in LLM-driven EDA. One promising direction is the exploration of more sophisticated retrieval and scoring mechanisms. While the current system effectively uses semantic similarity with the all-MiniLM-L6-v2 embeddings model, future iterations could incorporate hardware-aware metrics, such as graph-based similarity on dataflow representations or embeddings trained specifically on Verilog abstract syntax trees. This could further enhance the retriever's ability to identify the most functionally relevant examples for highly complex user queries as opposed to syntactical similarities.

Furthermore, *DeepV*'s model-agnostic nature makes it an ideal component for integration into larger, agentic workflows. By equipping an autonomous agent with *DeepV*'s high-quality retrieval capabilities, future systems could automate not only RTL generation but also subsequent tasks like self-correction, testbench generation, and synthesis optimization, creating a more holistic design automation solution. Finally, the core principles of *DeepV* could be extended beyond code generation. The framework's ability to ground LLMs in a high-quality, verifiable knowledge base like *VerilogDB* [32] could be adapted to address other critical challenges in the hardware lifecycle, such as secure linting, functional verification, and more.

6 Conclusion

The desire to propel EDA further has led hardware engineers to pursue LLM-based techniques for automated generation of RTL code. However, LLMs have not been trained on an adequate amount of hardware knowledge to have the capability in producing syntax-free, functionally accurate codes based on a given query. Moreover, fine-tuning these LLMs on hardware knowledge has limitations with the constant advancements of baseline models. In this work, we show that with an extensive knowledge base, a RAG framework can improve the performance of commercial LLMs by a large margin. Our work, *DeepV*, is model-agnostic, can be continuously updated, and provide high functional accuracy on a premier benchmark in this field, VerilogEval. As EDA automation moves forward with LLM-based frameworks, *DeepV* can act as a valuable tool not just for the accurate creation of RTL code but also for continued research within this community with our available-to-use HF space.

References

- [1] OpenAI, "Evaluating large language models trained on code," 2021, accessed: 2025-05-11. [Online]. Available: <https://openai.com/index/evaluating-large-language-models-trained-on-code/>
- [2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittweis, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.abq1158>
- [3] M. Abdollahi, S. F. Yeganli, M. A. Baharloo, and A. Baniasadi, "Hardware design and verification with large language models: A scoping review, challenges, and open issues," *Electronics*, vol. 14, no. 1, 2025. [Online]. Available: <https://www.mdpi.com/2079-9292/14/1/120>
- [4] Z. Zhao, R. Qiu, I.-C. Lin, G. L. Zhang, B. Li, and U. Schlichtmann, "Vrank: Enhancing verilog code generation from large language models via self-consistency," 2025. [Online]. Available: <https://arxiv.org/abs/2502.00028>
- [5] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, "Least-to-most prompting enables complex reasoning in large language models," 2023. [Online]. Available: <https://arxiv.org/abs/2205.10625>

- [6] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "Generating secure hardware using chatgpt resistant to cwe's," *Cryptology ePrint Archive*, 2023.
- [7] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.
- [8] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.
- [9] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.
- [10] N. Wang, B. Yao, J. Zhou, X. Wang, Z. Jiang, and N. Guan, "Large language model for verilog generation with code-structure-guided reinforcement learning," 2025. [Online]. Available: <https://arxiv.org/abs/2407.18271>
- [11] P. Wu, N. Guo, X. Xiao, W. Li, X. Ye, and D. Fan, "Itertl: An iterative framework for fine-tuning llms for rtl code generation," 2025. [Online]. Available: <https://arxiv.org/abs/2407.12022>
- [12] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterm: controlled verilog generation with discriminative guidance," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.
- [13] M. Akyash, K. Azar, and H. Kamali, "Rtl++: Graph-enhanced llm for rtl code generation," 2025. [Online]. Available: <https://arxiv.org/abs/2505.13479>
- [14] P. Yubeaton, A. Nakkab, W. Xiao, L. Collini, R. Karri, C. Hegde, and S. Garg, "Verithoughts: Enabling automated verilog code generation using reasoning and formal verification," 2025. [Online]. Available: <https://arxiv.org/abs/2505.20302>
- [15] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," 2024. [Online]. Available: <https://arxiv.org/abs/2311.04887>
- [16] J. Tang, J. Qin, K. Thorat, C. Zhu-Tian, Y. Cao, Yang, Zhao, and C. Ding, "Hivegen – hierarchical llm-based verilog generation for scalable chip design," 2024. [Online]. Available: <https://arxiv.org/abs/2412.05393>
- [17] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards llm-powered verilog rtl assistant: Self-verification and self-correction," 2024. [Online]. Available: <https://arxiv.org/abs/2406.00115>
- [18] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.
- [19] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "Autobench: Automatic testbench generation and evaluation using llms for hdl design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–10.
- [20] Z. Zhang, B. Szekely, P. Gimenes, G. Chadwick, H. McNally, J. Cheng, R. Mullins, and Y. Zhao, "Llm4dv: Using large language models for hardware test stimuli generation," 2025. [Online]. Available: <https://arxiv.org/abs/2310.04535>
- [21] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, Y. Liang, X. Zhang, D. Song, and D. Lin, *OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: <https://doi.org/10.1145/3676536.3676830>
- [22] Z. Mi, R. Zheng, H. Zhong, Y. Sun, S. Kneeland, S. Moitra, K. Kutzer, and Z. X. S. Huang, "Coopetitivev: Leveraging llm-powered coopetitive multi-agent prompting for high-quality verilog generation," 2025. [Online]. Available: <https://arxiv.org/abs/2412.11014>
- [23] D. Saha, H. A. Shaikh, S. Tarek, and F. Farahmandi, "Threatlens: Llm-guided threat modeling and test plan generation for hardware security verification," 2025. [Online]. Available: <https://arxiv.org/abs/2505.06821>
- [24] J. Zuo, J. Liu, X. Wang, Y. Liu, N. Goli, T. Xu, H. Zhang, U. R. Tida, Z. Jia, and M. Zhao, "Complexvcoder: An llm-driven framework for systematic generation of complex verilog code," 2025. [Online]. Available: <https://arxiv.org/abs/2504.20653>
- [25] H. Qin, Z. Xie, J. Li, L. Li, X. Feng, J. Liu, and W. Kang, "Reasoningv: Efficient verilog code generation with adaptive hybrid reasoning model," 2025. [Online]. Available: <https://arxiv.org/abs/2504.14560>
- [26] V. Geroimenko, *Key Challenges in Prompt Engineering*. Cham: Springer Nature Switzerland, 2025, pp. 85–102. [Online]. Available: https://doi.org/10.1007/978-3-031-86206-9_4
- [27] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-efficient fine-tuning for large models: A comprehensive survey," 2024. [Online]. Available: <https://arxiv.org/abs/2403.14608>
- [28] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges," *Information Fusion*, vol. 126, p. 103599, Feb. 2026. [Online]. Available: <http://dx.doi.org/10.1016/j.inffus.2025.103599>
- [29] P. Wu, N. Guo, J. Lv, X. Xiao, and X. Ye, "Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation," 2025. [Online]. Available: <https://arxiv.org/abs/2504.08862>
- [30] M. Gao, J. Zhao, Z. Lin, W. Ding, X. Hou, Y. Feng, C. Li, and M. Guo, "Autovcoder: A systematic framework for automated verilog code generation using llms," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 162–169.
- [31] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.
- [32] P. E. Calzada, Z. Ibnat, T. Rahman, K. Kandula, D. Lu, S. K. Saha, F. Farahmandi, and M. Tehranipoor, "Verilogdb: The largest, highest-quality dataset with a preprocessing framework for llm-based rtl generation," 2025. [Online]. Available: <https://arxiv.org/abs/2507.13369>
- [33] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," 2023. [Online]. Available: <https://arxiv.org/abs/2309.07544>

- [34] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: A year of improvements in large-language models for hardware code generation," 2025. [Online]. Available: <https://arxiv.org/abs/2408.11053>
- [35] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed. Springer Publishing Company, Incorporated, 2008.
- [36] S. Lahti, P. Sjövall, J. Vanne, and T. D. Härmäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.
- [37] N. Pundir, S. Aftabjahani, R. Cammarota, M. Tehranipoor, and F. Farahmandi, "Analyzing security vulnerabilities induced by high-level synthesis," vol. 18, no. 3, Jan. 2022. [Online]. Available: <https://doi.org/10.1145/3492345>
- [38] R. Muttaki, Z. Ibnat, and F. Farahmandi, "Secure by Construction: Addressing Security Vulnerabilities Introduced During High-level Synthesis," in *DAC '22: Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.
- [39] Z. Ibnat, H. M. Kamali, and F. Farahmandi, "Iterative Mitigation of Insecure Resource Sharing Produced by High-level Synthesis," in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2023, pp. 1–6.
- [40] S. Shi, N. Pundir, H. M. Kamali, M. Tehranipoor, and F. Farahmandi, "Sechs: Enabling security awareness in high-level synthesis," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 585–590. [Online]. Available: <https://doi.org/10.1145/3566097.3567926>
- [41] M. R. Muttaki, N. Pundir, M. Tehranipoor, and F. Farahmandi, *Security Assessment of High-Level Synthesis*. Cham: Springer International Publishing, 2021, pp. 147–170. [Online]. Available: https://doi.org/10.1007/978-3-030-64448-2_6
- [42] N. Pundir, F. Farahmandi, and M. Tehranipoor, "Secure high-level synthesis: Challenges and solutions," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 164–171.
- [43] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: <https://arxiv.org/abs/2002.08155>
- [44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: <https://arxiv.org/abs/2308.12950>
- [45] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chaddha, "A systematic survey of prompt engineering in large language models: Techniques and applications," 2025. [Online]. Available: <https://arxiv.org/abs/2402.07927>
- [46] D. Anisuzzaman, J. G. Malins, P. A. Friedman, and Z. I. Attia, "Fine-tuning large language models for specialized use cases," *Mayo Clinic Proceedings: Digital Health*, vol. 3, no. 1, p. 100184, 2025. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2949761224001147>
- [47] S. Alsaqer, S. Alajmi, I. Ahmad, and M. Alfaiakawi, "The potential of llms in hardware design," *Journal of Engineering Research*, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2307187724002177>
- [48] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "Llm for soc security: A paradigm shift," *IEEE Access*, vol. 12, pp. 155 498–155 521, 2024.
- [49] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," in *Proceedings of the 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6. [Online]. Available: <https://doi.org/10.1109/MLCAD58807.2023.10299874>
- [50] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "Chipppt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.
- [51] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.
- [52] S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie, "Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation(invited)," in *Proceedings of 2024 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2024.
- [53] Y. Zhang, Z. Yu, Y. Fu, C. Wan, and Y. C. Lin, "Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation," 2024. [Online]. Available: <https://arxiv.org/abs/2407.01910>
- [54] M. Liu, Y.-D. Tsai, W. Zhou, and H. Ren, "Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair," 2025. [Online]. Available: <https://arxiv.org/abs/2409.12993>
- [55] N. Pinckney, C. Deng, C.-T. Ho, Y.-D. Tsai, M. Liu, W. Zhou, B. Khailany, and H. Ren, "Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification," 2025. [Online]. Available: <https://arxiv.org/abs/2506.14074>
- [56] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," 2024. [Online]. Available: <https://arxiv.org/abs/2312.10997>
- [57] H. Koziolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani, "Llm-based and retrieval-augmented control code generation," ser. LLM4Code '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 22–29. [Online]. Available: <https://doi.org/10.1145/3643795.3648384>
- [58] O. Organization, "Opencores: Open-source hardware community," 1999. [Online]. Available: <https://opencores.org/>
- [59] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [60] Y. Zhao, D. Huang, C. Li, P. Jin, M. Song, Y. Xu, Z. Nan, M. Gao, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang, X. Zhang, Z. Du, Q. Guo, and X. Hu, "Codev: Empowering llms with hdl generation through multi-level summarization," 2025. [Online]. Available: <https://arxiv.org/abs/2407.10424>

- [61] B. Nadimi, G. O. Boudaib, and H. Zheng, “Pyranet: A multi-layered hierarchical dataset for verilog,” 2025. [Online]. Available: <https://arxiv.org/abs/2412.06947>
- [62] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conférence on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, 11 2019, pp. 3982–3992. [Online]. Available: <https://www.aclweb.org/anthology/D19-1410>
- [63] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvassy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, “The faiss library,” 2025. [Online]. Available: <https://arxiv.org/abs/2401.08281>
- [64] G. Qian, S. Sural, Y. Gu, and S. Pramanik, “Similarity between euclidean and cosine angle distance for nearest neighbor queries,” in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 1232–1237. [Online]. Available: <https://doi.org/10.1145/967900.968151>
- [65] H. Chase, “Langchain” GitHub repository, 2023. [Online]. Available: <https://github.com/langchain-ai/langchain>
- [66] S. Williams, “Icarus verilog documentation,” <https://steveicarus.github.io/iverilog/>, 2024.