Check for updates

EDA-Copilot: A RAG-Powered Intelligent Assistant for EDA Tools

ZHE XIAO, Hunan University, Changsha, China
XU HE, Information science and Engineering, Hunan University, Changsha, China
HAOYING WU, Wuhan University of Technology, Wuhan, China
BEI YU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
YANG GUO, National University of Defense Technology, Changsha, China

With the rise of Large Language Models (LLMs), researchers have become increasingly interested in their applications in EDA flows, particularly in specific subdomains such as serving as knowledge assistants and generating RTL code. In this study, we present a Retrieval-Augmented Generation (RAG) framework tailored to EDA task processing, named EDA-Adaptive RAG. This framework addresses the implicit semantics of EDA data and facilitates efficient knowledge acquisition through classification and enhanced retrieval, significantly enhancing LLMs ability to acquire EDA knowledge. Furthermore, we aim to integrate RAG into the design process as an EDA assistant application. Using RTL code generation as a case study, we demonstrate that the performance of RTL code generation can be enhanced through highly relevant retrievals provided by our RAG. The experimental analysis involves EDA Q&A tasks and RTL code generation evaluation. It is shown that our method outperforms the latest works in terms of both answer stability and code quality.

CCS Concepts: • **Hardware** → **Best practices for EDA;**

Additional Key Words and Phrases: Large language models, electronic design automation, RTL-to-GDSII, retrieval-augmented generation

ACM Reference Format:
Zhe Xiao, Xu He, Haoying Wu, Bei Yu, and Yang Guo. 2025. EDA-Copilot: A RAG-Powered Intelligent Assistant for EDA Tools. *ACM Trans. Des. Autom. Electron. Syst.* 30, 6, Article 99 (October 2025), 24 pages. https://doi.org/10.1145/3715326

1 Introduction

Electronic design automation (EDA) is a complex and crucial area for **integrated circuit (IC)** design. It utilizes a set of software tools within a complex design flow, especially ranging from **Register Transfer Level (RTL)** descriptions to the final chip layout format (GDSII). Mastering these tools requires significant time and expertise, as engineers must frequently write and adjust

This work is supported by the National Natural Science Foundation of China, under grant nos. 92473115 and 92373102.
Authors' Contact Information: Zhe Xiao, Hunan University, Changsha, China; e-mail: shawzer112@gmail.com; Xu He (Corresponding author), Information science and Engineering, Hunan University, Changsha, China; e-mail: dawn.hx@gmail.com; Haoying Wu, Wuhan University of Technology, Wuhan, Hubei, China; e-mail: why_dd@whut.edu.cn; Bei Yu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk; Yang Guo, National University of Defense Technology, Changsha, Hunan, China; e-mail: guoyang@nudt.edu.cn.

ACM 1084-4309/2025/10-ART99
https://doi.org/10.1145/3715326

scripts and manually interact with the tools to meet specific design requirements. This manual process is not only time-consuming and labor-intensive but also prone to configuration errors that can reduce the efficiency of design iterations. Therefore, there is a pressing need for intelligent assistance systems within EDA tools to optimize operations, reduce usage complexity, and help engineers efficiently tackle complex tasks.

With the emergence of advanced LLMs such as ChatGPT and GPT-4, researchers have shown increased interest in their potential applications in EDA flows. Leveraging generative language models, researchers have developed a variety of tools to aid engineers in EDA-related tasks. These tools target specific stages or tasks within the EDA process: (1) Knowledge assistant [14, 20, 38, 50], in which LLMs are utilized to access fragmented knowledge points, such as terminology explanation and knowledge advice. By replacing traditional search engines, they can help engineers to quickly obtain background knowledge; (2) Code Generation [21, 25, 29, 43, 51], in which LLMs assist engineers in interacting with EDA tools by generating scripts or **Hardware Description Languages (HDL)**. This speeds up the coding procedure and reduces errors, thereby improving efficiency dramatically; (3) Verification and analysis [9, 10, 45, 52], in which LLMs analyze code and error logs to identify problems and provide improvement solutions. This capability aids in debugging and optimizing designs, enhancing overall productivity.

Although these methods demonstrate significant progress in applying LLMs to the EDA domain, several challenges remain in assisting with EDA tool usage: (1) Limited functional coverage: Some tools focus on specific functionalities. For example, ChatEDA [51] specializes in script generation, while ORAssistant [14] addresses issues related to tool usage and installation, providing basic guidance to users. (2) Limited generalization: Some tools face constraints in accommodating the diversity and complexity of design workflows. For instance, OpenROAD-Assistant [38] focuses primarily on optimizing tool invocation during the physical design stage but lacks the flexibility to support other phases of the EDA design flow, i.e., RTL code generation. (3) Insufficient handling of complex data: The EDA domain involves substantial amounts of unstructured data, often a mix of code and text. Effectively supporting intricate tasks such as script invocation and annotation requires the capabilities for processing complex data.

To address these challenges, we propose an EDA-Adaptive RAG framework designed to provide comprehensive tool-assisted support. To accommodate diverse functional requirements, we constructed a large-scale EDA toolchain corpus encompassing terminology explanations, script retrieval, script annotation, and knowledge advice, as illustrated in Figure 1. This framework spans various stages of the EDA design process and integrates functionalities ranging from knowledge querying to code generation, covering typical scenarios across multiple design workflows. To effectively handle unstructured data, the framework employs a two-stage strategy comprising query classification and retrieval augmentation, thereby enhancing matching accuracy. Furthermore, recognizing that RTL code presents stricter syntactic rules and greater logical complexity, which is more challenging than other tasks, we further extend the framework for applying RAG to RTL code generation.

The major contributions of this work can be summarized as:

- Develop a comprehensive RTL-to-GDSII corpus generating and processing method for automated data extraction and cleaning. The corpus is further meticulously categorized and graded to different levels for comprehensiveness evaluation.
- Propose a low-cost EDA-Adaptive RAG without pre-training or fine-tuning. It employs a classifier and an enhanced retrieval optimizer to determine the category of the query input and narrow down the search scope, which significantly enhances the accuracy of corpus matching.

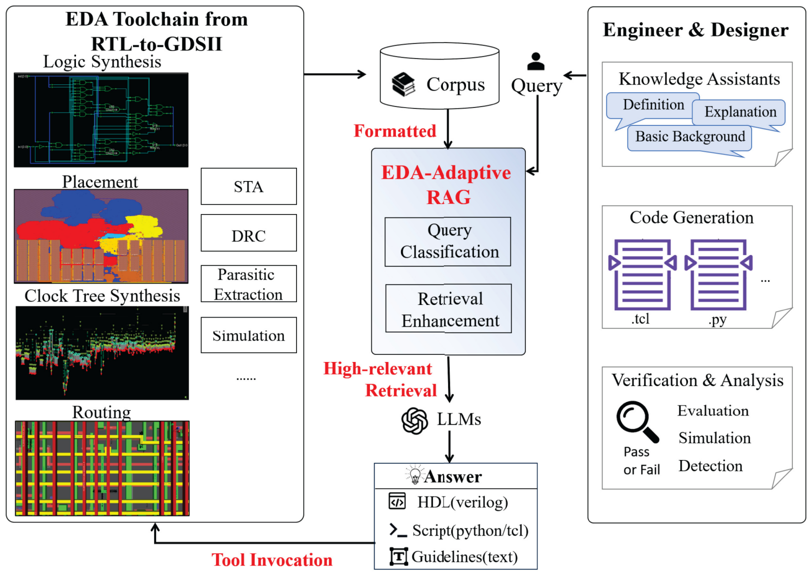

Fig. 1. The enhancement by EDA-Adaptive RAG For LLM In RTL-to-GDSII design flow.

— Propose a method to integrate EDA-Adaptive RAG into the design problem of RTL code generation. By utilizing highly relevant information retrieved from the RAG, the LLM can significantly improve the quality of the generated RTL code.
— Our experimental results demonstrate that EDA-Adaptive RAG outperforms previous works using the same LLM in handling general Q&A tasks across a variety of EDA query types and complexities. For hard-level queries, we achieve an average improvement of 7.45% to 16.11% compared to other solutions. Furthermore, when integrated into design application, such as RTL code generation, our RAG shows a 24% improvement in stability and a 36% improvement in accuracy compared to other specialized open-source RTL generation models.

The remainder of this article is organized as follows: Section 2 gives the preliminary of RAG. Section 3 presents the overview of our framework. The details of the EDA-formatted corpus and the EDA-Adaptive RAG methodology are given in Sections 4 and 5, respectively. Section 6 explores the application of RAG in RTL code generation. Experimental results are presented and discussed in Section 7, followed by the conclusion in Section 8.

2 Preliminary of RAG

LLMs are essentially general-purpose conditional probability models [26]. Extensive training on large corpora enables LLMs to acquire emergent abilities [2] by learning intricate patterns and relationships in language. However, without access to specialized data, general LLMs may struggle with tasks in specific fields.

To harness their potential in vertical domains [19, 39, 40, 46], fine-tuning [47] and RAG [16] are proved to be effective ways to help LLMs gain the necessary background knowledge. Different from fine-tuning for EDA tasks [21, 25, 29, 43, 51], RAG does not involve modifying LLM parameters. RAG is a real-time system [12] that combines information retrieval with content generation, making it ideal for knowledge-intensive tasks [54]. This approach maintains the model's ability to handle specialized tasks by not altering its original parameters during content

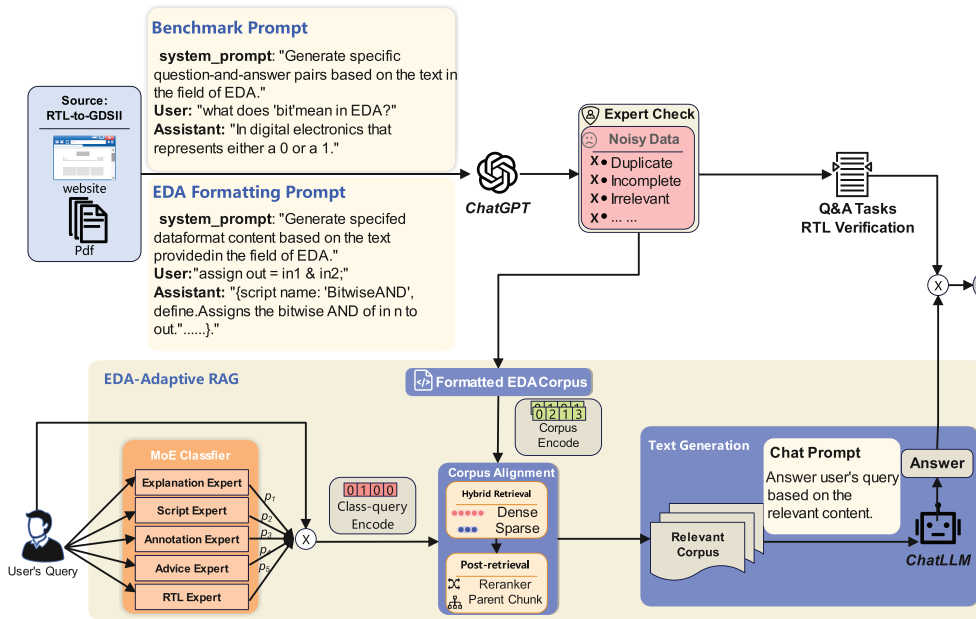

Fig. 2. The framework overview of the proposed EDA-Adaptive RAG method.

generation. Additionally, RAG allows for real-time updates to the knowledge base, providing access to the latest information at a low cost.

In the EDA domain, existing research has begun to explore the potential of RAG. For instance, ORAssistant [14] simplifies tool interaction through an intelligent question-and-answer system, while OpenROAD-Assistant [38] leverages **Retrieval-Augmented Fine-Tuning (RAFT)** techniques to enhance the accuracy of script generation and physical design Q&A tasks. Unlike ORAssistant and OpenROAD-Assistant, our approach is designed to address the diverse requirements of EDA tools comprehensively. To overcome the limitations of existing methods in task coverage and unstructured data processing, we developed a diverse corpus that includes terminology explanations, script retrieval, knowledge advice, and RTL code. We also implemented a two-stage strategy combining query classification and retrieval augmentation to improve retrieval precision and adaptability. By integrating dynamic retrieval and generation, our approach provides end-to-end support, from knowledge querying to code generation.

3 Overview

The framework overview of the proposed method is presented in Figure 2. It mainly consists of three parts: (1) Comprehensive corpus construction, (2) EDA-Adaptive RAG method in LLM, and (3) Text generation.

For the corpus construction process, we first clean the source dataset spanning all stages from RTL to GDSII. Based on the specific characteristics of EDA processes, we have categorized these data into five major categories: (i) **terminology explanation**, (ii) **script retrieval**, (iii) **script annotation**, (iv) **knowledge advice**, and (v) **RTL generation**. We then construct two prompt paradigms and feed the chunked text content into the LLM for dataset construction. The data are then stored in different databases according to their categories. This process ultimately generates an EDA corpus in a specific format, suitable for text matching within the RAG system, thereby enhancing the system's adaptability to the EDA domain.

In our EDA-Adaptive RAG method, a classifier architecture is integrated alongside retrievers and generators. We have developed a query classifier, inspired by the **Mixture-of-Experts (MoE)** approach [27], which functions similarly to a gating network to dynamically manage the identification and categorization of user queries. Each classification is responsible for managing a distinct type of handling. The system dynamically determines which expert database should be activated to assist in subsequent content retrieval. This approach can improve both the accuracy of retrieval and the generalization ability across varied EDA-related queries.

Next, we utilize mixed indexing to preliminarily filter and select content related to the query from the expert databases. This method not only includes the precision of similarity indexing but also considers the diversity brought by sparse indexing. Finally, we employ a retrieval post-processing step to further refine the selection of content from the related texts, ensuring a higher match score with the query. This method enhances the final retrieval accuracy, providing assurance of retrieving the correct textual samples.

After that, the relevant text chunks with the user query into a unified input sequence; this process resembles a few-shot learning [3] approach. It provides LLMs with a few relevant examples, enhancing its contextual understanding and enabling it to generate more accurate and contextually appropriate responses. To improve the recognizability of the input, we add context markers such as "Retrieved content" and "User query." These markers act as special tokens, clearly delineating different sections of the input. This setup provides explicit cues for the attention mechanism, guiding the model's focus more effectively. As a result, the model can dynamically adjust its generation strategy based on the provided context, ensuring that the output is coherent and appropriately tailored to the context.

Our EDA-adaptive RAG is evaluated on the general EDA Q&A tasks. We have also explored the application of RAG embedded in LLMs for more challenging tasks within the EDA domain. We selected RTL code generation as a case study and give an effective integration method of RAG for the specific task of RTL code generation.

The further details of the corpus generation and EDA-Adaptive RAG are discussed in Sections 3 and 4, respectively.

4 RTL-to-GDSII Data Sources and Formatting Construction

4.1 Corpus Resource

Total dataset consist of two main parts: The first part comes from open-source tool manuals and design process descriptions, which primarily serve to explain how to use these tools for handling our query and answer tasks. The second part is derived from open-source RTL datasets, which are mainly used to guide the task of RTL code generation.

EDA tools dataset: To ensure the legality of the dataset, our work relies entirely on open and accessible technical resources, including publicly authorized open-source EDA tools such as OpenROAD [34], Yosys [35], and KLayout [33], as well as relevant open-source cases and their associated documentation [7, 8, 30–32, 48]. Additionally, we utilize legitimate open-source datasets such as EDA Corpus [50] and ORAssistant [14]. In total, we collected 5,254 documents with a combined data volume of approximately 1.34 GB. To enhance the dataset structure and applicability, we apply regular expressions to remove irrelevant information (such as indexes and prefaces) and systematically categorize and annotate the data.

Verilog code dataset: This dataset originates from RTLCoder [21], which was generated by the commercial GPT-3.5 model rather than official RTL sources. Since our RAG approach relies on official data for accurate retrieval, any incorrect data could result in unreliable outputs from the language model. To ensure the dataset's accuracy, we manually reconstructed the open-source dataset

ORIGINAL DESIGN REQUIRMENT

Please act as a professional Verilog designer.
↵ Build a module that implements a 4-bit barrel shifter.
↵ The module should take in a 4-bit input and a 2-bit shift amount,
↵ and output the input with its bits shifted **<left or right?>** by the specified amount.
↵ Use a combination of **<multiplexers and shift registers?>** to implement the module.

FIXED DESIGN REQUIRMENT

Please act as a professional Verilog designer.
↵ Build a module that implements a 4-bit barrel shifter.
↵ The module should take in a 4-bit input and a 2-bit shift amount,
↵ and output the input with its bits shifted ***left*** by the specified amount.
↵ Use a combination of ***case statements*** to implement the module.

ORIGINAL VERILOG CODE

```
// 4-bit Barrel Shifter
// This module implements a 4-bit barrel shifter using case statements to
// determine the amount of shift based on the 2-bit shift_amt input.
// The shift is intended to be a left shift, but the implementation contains errors.
module barrel_shifter (
   input [3:0] in,
   input [1:0] shift_amt,
   output reg [3:0] out
);
   // This block handles the shifting operation based on the shift amount
   always @(*) begin
      case(shift_amt)
         2'b00: out = in;                  // No shift
         2'b01: out = {in[3], in[0:2]};  // Intended left shift by 1 bit (incorrect implementation)
         2'b10: out = {in[2:3], in[0:1]};// Intended left shift by 2 bits (incorrect implementation)
         2'b11: out = {in[1:3], in[0]};// Intended left shift by 3 bits (incorrect implementation)
      endcase
   end
endmodule
```

FIXED VERILOG CODE

```
// 4-bit Barrel Shifter
// This module correctly implements a 4-bit barrel shifter that shifts the input
// bits to the left by the amount specified in the 2-bit shift_amt input.
// The output is determined using case statements, ensuring correct left shift
// for each possible shift amount.
module barrel_shifter (
   input [3:0] in,
   input [1:0] shift_amt,
   output reg [3:0] out
);
   // This block handles the shifting operation based on the shift amount
   always @(*) begin
      case(shift_amt)
         2'b00: out = in;                  // No shift
         2'b01: out = {in[2:0], 1'b0};     // Left shift by 1 bit
         2'b10: out = {in[1:0], 2'b00};    // Left shift by 2 bits
         2'b11: out = {in[0], 3'b000};     // Left shift by 3 bits
      endcase
   end
endmodule
```

Fig. 3. An example of a 4-bit barrel shifter with corrections.

from RTLCoder. We selected 200 samples from the RTLCoder open-source dataset and manually rewrote each sample, expanding the dataset to approximately 1,000 entries. This expanded dataset was then merged with the original dataset, resulting in a training dataset of over 26,000 entries.

We manually rectify the RTL code identified in the dataset, including: (1) Syntax Reconstruction: We rearrange the sequence of sentences, adjust tenses, and refine the logical flow of the language to describe the design examples using different semantic expressions. (2) Simplification: We simplify the original the design requirements and process by removing some detailed specifics. (3) Parameter Adjustment: While maintaining the original functionality, we adjust the input and output parameters to enhance the diversity. (4) Verilog Refactoring: By modifying code structure, module names, signal names, and connectivity, we refactor the code to introduce different coding styles while preserving the original functionality. For example, Figure 3 shows that we clarified the left shift direction and identified an error where the most significant bit is not shifted properly in a 1-bit left shift.

4.2 Comprehensive Corpus Construction

The **Question Answering (Q&A)** task [42] is critical for evaluating RAG systems in the EDA-specific domain. Unlike natural language processing, EDA problems involve a wide range of specialized knowledge and higher difficulty levels. Additionally, there is a large volume of unstructured text data in this field, which makes it challenging to directly extract the characteristics of EDA problems.

To support diverse retrieval needs in the EDA field, such as terminology explanation, script retrieval, script annotation, and knowledge advice, we preprocessed the data samples into a structured format. We have identified several common retrieval scenarios, detailed as follows:

- **Terminology explanation**: explaining fundamental concepts and terminology that form the cornerstone for understanding various EDA processes from RTL to GDSII. This category

encompasses terms such as "netlist," "routing," and "synthesis." Grasping these definitions is crucial, as they constitute the language engineers use to communicate intricate design details and constraints. For instance, comprehending **Layout Versus Schematic (LVS)** is essential to ensure the physical layout of an integrated circuit accurately implements its schematic design. Moreover, a thorough understanding of these terms is necessary to navigate and apply advanced optimization techniques required in modern EDA workflows.
— **Script retrieval**: creating and manipulating scripts to automate various stages of the electronic design automation process, including testbench writing, automating simulation tasks, and synthesis scripting. Relevant domains include scripting languages such as TCL and Shell, which are instrumental in enhancing the productivity and reliability of design processes. This category assesses the ability to generate functional scripts based on specific design requirements, reflecting both basic scripting tasks and more complex scenarios involving integration of multiple EDA tools. Script retrieval not only streamlines the design process but also enables precise control over simulation parameters and the design verification flow.
— **Script annotation**: comprehending, commenting on, and improving existing EDA scripts. This includes explaining script functionality, detecting potential errors, and suggesting optimizations. Annotating scripts facilitates better collaboration among team members and ensures scripts are maintainable and adaptable to changes in design specifications. Our queries cover various aspects of script annotation, including understanding script flow, variable usage, and error handling mechanisms. Script annotation is critical for maintaining the robustness of design automation environments and ensuring scripts adhere to best practices and standards.
— **Knowledge advice**: EDA knowledge advice involves recommending tool choices, optimizations, and solutions. This category tests the ability to propose viable solutions to design challenges, such as power reduction, area optimization, and timing closure. For instance, our queries could explore how the restructuring module in OpenROAD can be employed to optimize design performance, which involves leveraging logic re-synthesis to either minimize cell area in area mode or enhance timing on critical paths in delay mode. Knowledge advices are generated based on reference datasets of process designs, problems, and relevant knowledge, offering actionable guidance and solutions. For example, ChatEDA [51] utilizes a pre-trained task planning dataset that enables the model to decompose complex tasks and apply suitable EDA tools to address them. Similarly, the EDACorpus [50] and OpenROAD-Assistant [38] include datasets related to the OpenROAD flow and design strategy, empowering the model with enhanced capability for knowledge advice.

Overall, these data share the common characteristic of being complex and requiring specialized background knowledge to understand. Because of the complexity and fragmented, interconnected nature of the information, it is challenging for the RAG framework to interpret the content directly. To address this, the information is structured by highlighting key points. Figure 4 presents the specific meanings of script-related content, which helps in more effectively finding the right information during searches.

Due to the substantial costs associated with manually crafting such instructions, the self-instruction paradigm, as described in Reference [28], has gained prominence in recent years. This methodology leverages the powerful contextual learning capabilities of GPT-3.5/4. In our corpus generation, we prompt GPT-3.5/4 with diverse instruction templates to automatically generate a series of query-answer pairs. For open-source datasets already in query-answer format, such as the EDA Corpus [50], the data are reformatted, and detailed classification information is added. For publicly authorized open-source EDA tools [7, 8, 30–35, 48], following the document preprocessing

```
SCRIPT FORMAT
{
  "script_name": "<Name of the script>",
  "definition_description": "<A brief description of the script purpose>",
  "parameters":
    {
      "parameter1": "<Description of the first input parameter and its role>",
      "parameter2": "<Description of the second input parameter and its role>",
      "parameterN": "Add more parameter as needed"
    },
  "values": "<Values corresponding to each script in the text>",
  "script_paradigm": "<TCL code paradigm for the script>",
  "examples": [{"query": "A summary of the specific example described in the text intoquestion",
            "answer": "Specific code for the example"}]
}
```

Fig. 4. Script corpus format.

approach in References [51, 53], the open resources are first split into 7,761 text chunks, with each chunk limited to a token size of 4,096. ChatGPT is then prompted to generate EDA-related questions and accurate answers based on each text chunk. Specifically, if a text chunk contains script-related information, then ChatGPT extracts and organizes the content into a structured script format, as shown in Figure 4. This process ensures that the generated content is highly consistent with the reference text chunks, effectively reducing the likelihood of erroneous information. A total of approximately eight person-days were spent validating the accuracy of the entire dataset in alignment with the referenced content, with errors or inconsistencies removed. Following this process, a corpus containing around 4,200 data samples was successfully constructed. The code for corpus generation and dataset examples are publicly available at https://github.com/Szzer1/EDACopilot.

For the RTL code dataset, since prior work has already structured it as supervised data, we only need to store the RTL design descriptions and corresponding code in our database, without requiring additional formatting. Overall, our dataset includes over 30,000 entries, covering EDA tools and Verilog code samples. These entries are sourced from open-source documentation and open-source RTL, encompassing the entire RTL-to-GDSII design process.

5 EDA-Adaptive RAG Methodology

As shown in Figure 5, given a set of natural language-described query-answer tasks $Q = (X, Y)$, where each $x_i \in X$ represents a query input and each $y_i \in Y$ is the corresponding system output, our method integrates LLM with an EDA-Adaptive RAG process, enabling the system to effectively handle the high-density knowledge-intensive tasks prevalent in EDA. Due to the diverse nature of EDA problems, for each input x_i, a category classification $C = \{c_1, c_2, c_3, c_4\}$ is applied to enhance the accuracy and stability of x_i, where c_1 is terminology explanation task, c_2 is script retrieval task, c_3 is script annotation task, and c_4 is knowledge advice task. While directly using classifier C may not be successful, we employ a prompting technique P_C to refine the natural language description of x_i. This refined description is then fed into C to ensure that the output closely aligns with EDA-specific knowledge. At last, the classified result $C(P_C(x_i))$ is then passed to an enhanced retrieval model R to align the input and its determined category with relevant text content,

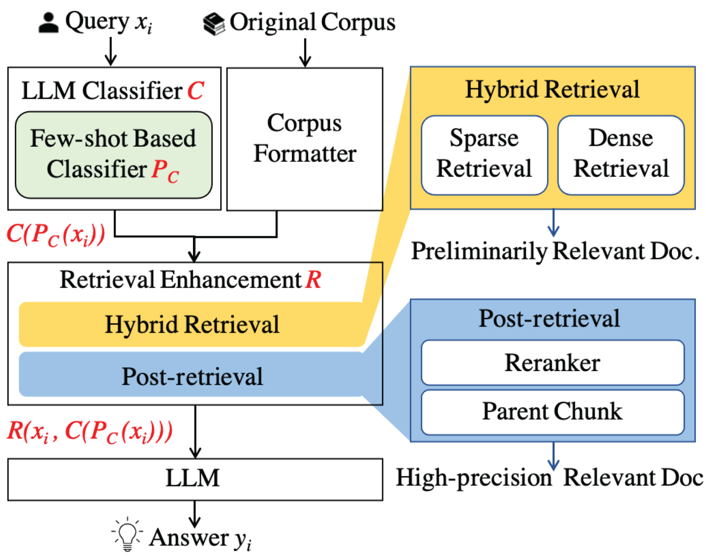

Fig. 5. EDA-adaptive RAG flow.

ultimately producing a comprehensive retrieval answer $x_i' = R(x_i, C(P_C(x_i)))$. The proposed approach of category classification and refined retrieval can greatly improve the accuracy and relevance of our query-answering system in the EDA domain, enabling it to effectively handle diverse and complex EDA-related queries. We introduce the techniques of query classification and retrieval enhancement as follows.

5.1 Query Classification

To enhance EDA task classification, we give a prompt engineering technique, which employs meticulously crafted prompts to precisely guide model output.

The prompt structure format and examples are illustrated in Figure 6. It includes three parts: (1) Task specification (lines 1–3): By explicitly detailing the expected query handling process, it ensures that the model accurately grasps the query content and aligns it with the appropriate professional category. (2) Output format specification (lines 4–5): It specifies the desired output format, instructing the LLM to output the classification results in a dictionary format. This structured output requires the model to not only generate category labels but also provide probability values for each category. These critical values can be extracted by subsequent processing steps to accurately determine the query category. (3) Context and expected response: The final section provides specific context information and the expected model response (lines 6–13). This enables the LLM to better adapt and understand specific task requirements through few-shot learning [13]. The purpose of employing few-shot examples is to enable the model to quickly adapt and optimize its performance with minimal training data. By leveraging few-shot learning techniques, the model can achieve significant performance improvements with limited data support, significantly reducing reliance on large-scale labeled datasets and enhancing learning efficiency and model adaptability.

It is found that query classification can improve answer accuracy by narrowing the search space and reducing interference from irrelevant information. However, it should be noted that query classification may not always be entirely accurate, as the query can be uncertain or subjective. If the classification is incorrect, then it may affect the accuracy of the results, though not completely.

Classification Prompt

1. Given an EDA−related question, classify it into the following categories:
2. Terminology Explanation, Script Retrieval, Script Annotation, and Knowledge Advice.
3. Give the probability of each category in a dictionary format.

4. **Format:**{'**Terminology Explanation**': xx, '**Script Retrieval** ': xx, '**Script Annotation**': xx, '
5. **Knowledge Advice**': xx, ' **RTL Generation**': xx}

6. **Example1:**
7. **Question:** 'What is logic synthesis?'
8. **Output:** {'**Terminology Explanation**': 0.90, '**Script Retrieval** ': 0.05, '**Script Annotation**': 0.03, '
9. **Knowledge Advice**': 0.02, ' **RTL Generation**': 0.01}

10. **Example2:**
11. **Question:** "How can I optimize a Verilog code for better synthesis?"
12. **Output:** {'**Terminology Explanation**': 0.10, '**Script Retrieval** ': 0.30, '**Script Annotation**': 0.10, '
13. **Knowledge Advice**': 0.50, ' **RTL Generation**': 0.02}

Fig. 6. Few-shot prompt structure.

This is because there is often overlap between different query categories. For example, a query seeking a terminology explanation might match reference text chunks related to knowledge advice, which can still provide relevant information to answer the query. Therefore, even if an error occurs, the subsequent retrieval enhancement can still identify relevant content to support the query, minimizing the impact of the error. Our experiments further discuss the effectiveness of query classification in Section 7.1.

5.2 Retrieval Enhancement

Corpus matching is a crucial component of the RAG framework. For each user query, relevant corpus data must be retrieved to enhance the ability to provide accurate and informative answers. The higher the degree of relevance between the retrieved corpus and the query, the more effective the system can mitigate hallucinatory responses from the model. Moreover, corpus matching enables the incorporation of external knowledge without the need for additional training, allowing the LLM to effectively address previously unseen knowledge content. Conversely, if the retrieved content does not align with the actual corpus required to support the answer, then the LLM may generate erroneous responses. Consequently, corpus matching plays a pivotal role in the EDA-Adaptive RAG framework, necessitating an effective retrieval algorithm to enhance matching accuracy.

As illustrated in Figure 5, our EDA-Adaptive RAG begins with preprocessing the source data, employing an LLM-enhanced approach to generate a large volume of structured data. These data preserve the domain-specific tokens of the EDA area, preventing semantic fragmentation due to context length limitations. Next, we employ a domain classifier to match the query category, promoting retrieval accuracy. Large text chunks may contain significant amounts of filler text, obscuring semantic representations and hindering retrieval performance. By leveraging smaller, more targeted chunk embedding/retrieval, we can improve retrieval accuracy. Therefore, we set the chunk size to 1,024 when segmenting text chunks. To further enhance retrieval accuracy and diversity, we employ a Retrieval-combined-with-Post-Retrieval approach. In the Retrieval stage, we must preserve the implicit mapping relationships between texts. Hence, we utilize a hybrid retrieval method that combines dense and sparse retrieval. Traditional sparse retrieval methods typically employ word frequency retrieval approaches such as BM25 [36] and TD-IDF [24]. In contrast, we adopt a sparse low-dimensional vector space representation for word distribution and a dense high-dimensional vector space representation for word distribution. Finally, we fuse the two retrieval methods, setting a hyper-parameter to limit the number of relevant text chunks retrieved to 25. For post-retrieval processing, we encounter the challenge of passing the large volume of relevant content to the LLM due to its token length limitations. To address this, we compress the relevant content using a high-precision reranker model to rerank the retrieved text chunks, ultimately selecting eight final texts. While our retrieval approach utilizes smaller text segments to improve text matching accuracy, it may result in context information loss when passed to the LLM. Therefore, we retain the parent text chunks before text slicing. After reranking, we utilize the retrieved relevant content to trace back to their parent text chunks. This ensures that more context information is preserved when passed to the LLM, promoting the accuracy of its responses.

Figure 7 illustrates our RAG process. Given a query input x_i, we obtain the corresponding query category $C(P_C(x_i))$. Both the category $C(P_C(x_i))$ and query x_i are then subjected to retrieval enhancement. Finally, the retrieval request $R(x_i, C(P_C(x_i)))$ is sent to LLM. After final text generation, the answer y_i is obtained.

6 RAG Application Exploration in RTL Code Generation

Beyond EDA Q&A tasks, we have explored the application of RAG embedded in LLMs for more complex tasks within the EDA domain. We selected RTL code generation as a case study due to its requirement for intricate logical reasoning, which presents a greater challenge compared to pure natural language-related problem. By leveraging RAG's ability to identify implicit logical relationships between code and text, we can retrieve logically coherent code that aligns with functional descriptions derived from query tasks. This approach provides LLMs with highly relevant code references, aiding in the generation of more accurate RTL code.

As shown in Figure 8, our EDA-Adaptive-RAG framework is designed to enhance the generative capabilities of LLMs by retrieving relevant RTL design cases and integrating them with test tasks to construct new prompts. This approach increases the demand for longer context lengths, as the combined input may exceed the model's capacity, leading to the truncation of the text. This truncation causes the model to focus predominantly on the most recent or end portions, potentially overlooking important information at the beginning. As a result, this can lead to incomplete or inaccurate indexing, diminishing the quality of the generated outputs. Therefore, we choose the deepseek-coder-6.7b-instruct [5], which supports a context length of up to 16K tokens, as each design case typically spans around 3,000 tokens. This is also why we choose to limit the number of retrieved relevant designs to a maximum of four, as exceeding this number could further risk input truncation and potentially compromise the quality of the generated outputs. It is important to note that our EDA-Adaptive RAG framework does not impose any restrictions on token length. Since the LLM input is standardized through the construction of natural language prompts, the input interface remains consistent across all LLMs. If handling longer tokens is required, then EAR can flexibly switch to alternative LLMs that support extended token lengths. Additionally, deepseek-coder-6.7b-instruct has not been trained on the EDA academic dataset, ensuring an unbiased evaluation in our experimental setup.

Furthermore, to minimize non-standard code in the generated output, we explicitly instruct the LLM in the prompt design to produce results in a standard Verilog format. Furthermore, to

Fig. 7. Script annotation example of query-answer process of our RAG.

improve accuracy, we implemented a post-processing mechanism based on regular expressions to precisely extract portions of the output marked as Verilog code while removing redundant or irrelevant content. If the extraction process fails, then the framework automatically initiates a new generation attempt, ensuring the final output is correct in both format and content.

7 Experimental Results

To address the wide-ranging and fragmented nature of EDA full-process data, we have found that it is very challenging to generate the desired specialized knowledge based solely on natural language descriptions provided by users. Such descriptions are often abstract, hindering effective understanding and processing. Consequently, we conducted experiments to compare the accuracy of different methods in addressing Q&A challenges across the entire EDA process. Furthermore, to validate the potential of our approach in the complex domain of EDA, particularly in RTL code generation, which requires stringent logical reasoning and precise parameter understanding. Therefore, we have compared the performance of our RAG method with other state-of-the-art RTL generation models to assess its effectiveness. We conducted our experiments on a Centos 7.9 system equipped with an A40 GPU with 45 GB of memory. Additionally, we employed the VLLM [15]

Fig. 8. RTL generation example of query-answer process of our RAG.

backend inference acceleration framework, which provided inference speeds 4–5 times faster than the Transformers [49] backend.

7.1 RAG Validation on EDA-domain Q&A Tasks

To evaluate the performance of the proposed method, we devised over 1,900 high-quality testing instructions, as shown in Figure 9, based on the difficulty classification used in ChipNemo [20]. The terms "Exp.," "Ret.," "Ann.," and "Adv." represent the four query categories: terminology explanation, script retrieval, script annotation, and knowledge advice, respectively. Furthermore, each category is divided into three levels:

Easy Level: Involves queries where the answer can be deduced from one single reference text. The information is usually specific and well-defined, such as exact numerical values or clear definitions of terms. These types of questions are often found in terminology explanations. For example, "What is a 'CPF file' in the context of low-power IC design?"

Medium Level: Represents queries are more complex and require different approaches, such as summarizing and synthesizing information from several different reference texts. Typically, they involve combining multiple sources to extract key information relevant to the user's question. For instance, "Please annotate the following script: block = ord.get_db_block() inst =

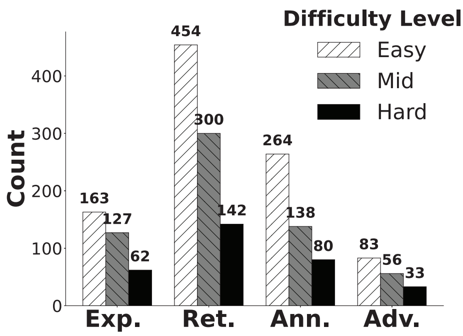

Fig. 9. The distribution of testing corpus of different categories under different levels.

Table 1. Parameter Setting of RAG

Hyperparameters	Value
Chunk Size	1,024
Overlap	50
Relevant chunks	8
Embedding model	bge-m3 [4]
Reranker model	bge-reranker-v2-m3 [4, 18]
LLM	Llama-3-8B-Instruct [44]

block.findInst("_ff_") inst.setDoNotTouch(True)". These types of questions are often found in script annotation scenarios.

Hard Level: Involves queries that often span multiple EDA processes. These questions usually provide few keyword clues and may cover a broad range of topics with more rigorous design logic, including advices, scripts, and various concepts. It is challenging to directly retrieve a relevant reference text, because these queries require extracting high-dimensional semantic information. Understanding the underlying semantics and making connections between different pieces of information is essential to grasp the hidden meaning behind the query. For example: "If the widths of the two outer shapes are 0.20 and 0.22, respectively, what is the additional width requirement for the middle shape based on the 'overrideWidths' parameter?" These types of questions are often found in script retrieval and knowledge advice scenarios.

In the text-based question-answering task, the evaluation is to measure the semantic consistency between the generated answers and the human-annotated reference answers. Referring to OpenROAD-Assistant [38] and ORAssistant [14], BERTScore [55] is applied as the primary evaluation metric. Additionally, the F1 score is utilized as the final evaluation criterion, as it effectively balances precision and recall, providing a more comprehensive assessment of the overall quality of the generated answers.

To evaluate our EDA-Adaptive RAG, our proposed EDA-Adaptive RAG methods (denoted as "`EAR`") are compared with the other RAGs, including: (1) Naive-RAG "`Base`" [17]: A traditional basis RAG framework without any specific optimization; (2) Rewrite-Retrieve-Read "`RRR`" [23]: It improves retrieval accuracy by rewriting and optimizing user queries; (3) Hypothetical Document Embedding "`HyDE`" [11]: It generates a hypothetical document from a query and encodes it into an embedding vector to find similar real documents in the corpus. (4) "`OpenROAD-Assistant`" [38]: It employs retrieval-aware fine-tuning to develop physical design chatbot for script generation and question-answering using OpenROAD; (5) "`ORAssistant`" [14]: RAG-based conversational assistant for OpenROAD. It aims to improve the user experience for the OpenROAD flow from RTL-GDSII by providing context-specific responses to common user queries, including installation, command usage, flow setup, and execution, in prose format. Other RAGs approaches are not applicable for large amounts of local data or long text problem, i.e., RAPTOR [37] and Self-RAG [1]; they are not utilized for the comparison. All RAGs are finally fed into the LLM llama3-8B model [44] for final text generation. Additionally, we compared with an expert agent in the EDA domain: (6) ChatEDA [51]: An expert LLM system for manipulating EDA tools based on the OpenRoad API. It has to be mentioned that ChatEDA is not open-sourced; we reproduce it in the experimental analysis with our dataset. To be a fair comparison with other counterparts, the Llama3-8B model

Table 2. EDA-domain Q&A Tasks Performance Comparison

Method	Level	Explain(%)	Ret.(%)	Ann.(%)	Advice(%)	Avg.(%)
Base	Easy	93.21	92.14	91.94	92.28	92.39
	Mid	89.23	85.38	89.94	84.49	87.26
	Hard	77.49	62.67	73.77	74.74	72.16
RRR	Easy	93.17	92.15	91.82	92.02	92.29
	Mid	88.40	86.62	89.82	88.01	88.21
	Hard	68.62	61.57	77.40	67.76	68.84
HyDE	Easy	93.60	91.89	91.98	**92.95**	92.60
	Mid	89.94	84.19	89.87	87.09	87.77
	Hard	76.26	62.32	74.95	74.12	71.91
ChatEDA	Easy	91.12	92.29	92.23	91.34	91.74
	Mid	**90.49**	91.24	90.14	77.30	87.29
	Hard	**81.69**	77.55	84.60	68.18	78.00
OpenROAD-Assistant	Easy	**94.16**	93.41	90.96	92.58	92.78
	Mid	89.00	88.05	88.33	83.81	87.05
	Hard	76.86	64.72	76.93	72.63	72.78
ORAssistant	Easy	89.07	91.89	91.89	91.89	91.69
	Mid	86.85	89.58	90.37	86.47	88.57
	Hard	74.13	61.02	74.57	73.88	70.90
EAR	Easy	93.55	**95.03**	**95.64**	92.14	**94.09**
	Mid	89.52	**94.80**	**93.66**	**91.60**	**92.39**
	Hard	81.65	**91.30**	**86.22**	**82.64**	**85.45**

is applied as well. Followed by ChatEDA method, the Llama3-8B model has been fine-tuned using the LoRA [6].

To eliminate the influence of other hyperparameters on the results of different RAG frameworks, those parameters are unified as the settings shown in Table 1. After the text content generation in the RAG, there is no further manual intervention on the results. The generated results are directly compared to the reference golden answers for evaluation and scoring.

Table 2 gives the RAG comparison result. The columns "Explain," "Ret.," "Ann.," and "Advice" denote the four query types of terminology explanation, script retrieval, script annotation, and knowledge advice, respectively. For Easy-level queries, all solutions achieve average scores above 89%. As the query difficulty increases, our EDA-Adaptive RAG significantly improves the accuracy of the answers. Specifically, our method outperforms others in both script retrieval and annotation across all query difficulty levels. This is attributed to our structured text representation, which preserves the completeness of script descriptions. For Hard-level queries, we achieve an average improvement of 12.67%–16.11% relative to other general RAGs. Similarly, compared to the expert agent ChatEDA, our solution outperforms it by an average of 5.10% on mid-level queries and 7.45% on hard-level queries. While ChatEDA excels in terminology explanation, it does not surpass other RAGs in knowledge advice. This is because, during its training, LoRA reduces the number of trainable parameters by using low-rank adaptation matrices. This approach may limit the flexibility in handling very specific or complex tasks, such as detailed parameter explanations. Besides, its fine-tuned model operates in a zero-shot inference mode. As a result, it may lack the relevant reference to generate accurate output, especially complex advice. For example, as shown in Figure 10, for a script generation query from user manual, ChatEDA comprehended the overall task requirements but failed to capture the specific parameter details and the detailed implementation logic. This

Query: x_i

Please generate script:
Horizontally flip the rotation of the instances whose library cell types are 'AND2_X1' with OpenROAD API

Our Answer: y_i

```
from openroad import Design
# Horizontally flip the instance orientation
def flipX(value):
    return {
        "R0": "MX", "R90": "MYR90", "R180": "MY", "R270": "MXR90",
        "MY": "R180", "MYR90": "R90", "MX": "R0", "MXR90": "R270"
    }.get(value, value)                      # Map current orientation to its horizontally flipped counterpart

design = Design("path/to/design")            # Load the design
for inst in design.getBlock().getInsts():    # Iterate through all instances
    if inst.getMaster().getName() == 'AND2_X1':   # Check if type is 'AND2_X1'
        inst.setOrient(flipX(inst.getOrient()))   # Apply flipped orientation
```

ChatEDA: y_i

```
from openroad import Tech, Design
from pathlib import Path

tech = Tech()
design = Design(tech)
design.readLibs(str(Path("path.to.your.design").parent / "libs"))
for inst in design.getInsts():             # Error: `design.getInsts()` should be `design.getBlock().getInsts()`.
    if inst.getConst() == “AND2_X1”:       # Error: `inst.getConst()` should be `inst.getMaster().getName()`.
        inst.mirrorX()                     # Error: `mirrorX()` performs mirroring, not rotation flipping.
design.writeVerilogNet("output_file.v")    # Error: `writeVerilogNet()` is not a valid method, use `writeVerilog()` instead.
```

OpenROAD-Assistant: y_i

```
# Get the design block
block = design.getBlock()           # Error: Instances are not retrieved; use block.getInsts() to access them.
# Get the technology information
tech = block.getTech()              # Error: Technology info is irrelevant to flipping instance orientations.
# Return the measurement system
return tech.getMeasureSystem()      # Error: Measurement system is unrelated to the requirement.
```

ORAssistant: y_i

```
# Import ORFS library
source ./test/convert_tcl
# Define design name and get masters
set masters [get_masters_arg -design "your_design_name"]
# Iterate instances and print their regions
foreach instance $masters { puts "Instance: $instance, Region: [get_inst_bbox -instance $instance]"}
```

Base: y_i

This script uses the RBA module to automate the horizontal flipping of rotation for instances of type 'AND2_X1' in a layout. It employs a recursive instance iterator (RBA.RecursiveInstanceIterator) to traverse all instances, filters those matching the specified cell type, retrieves their current rotation angles using inst.rotation(), computes the horizontally flipped angle with -rot_angle % 360.

RRR: y_i

You can use the klayout library to horizontally flip 'AND2_X1' instances by iterating through them, applying a flip transformation, and saving the modified layout to a new file.

HyDE: y_i

The script iterates through instances of type 'AND2_X1', retrieves their original transformations, calculates the flipped rotation, and applies the updated transformation to each instance, ensuring horizontal flipping.

Fig. 10. Script retrieval example of different solutions comparison.

Prompt

Please act as a professional Verilog designer.
↵ Create a module that implements a 4-input OR gate using Verilog. The gate should take 4 inputs and output a single bit.
↵ The inputs should be named A1, B1, C1, and D1. The output should be named Y.
↵ Power and ground connections should be named VCC and GND respectively.

```
module or4_gate (
    input A1, // Input signal A1
    input B1, // Input signal B1
    input C1, // Input signal C1
    input D1, // Input signal D1
    output Y  // Output signal Y
);
```

Correct Completion: ***BERTScore>0.9***

```
// Bitwise OR operation: combines all input
// signals bit by bit and assigns the result to output Y
    assign Y = (A1 | B1 | C1 | D1);
endmodule
```

Incorrect Completion:

```
  // Logical OR operation: assigns the result of
  // logical OR operation to output Y
    assign Y = (A1 || B1 || C1 || D1);
endmodule
```

Fig. 11. Challenges of evaluating RTL code accuracy using BERTScore.

limitation stems from the constraints of its LoRA-based training. Conversely, other RAGs systems failed to grasp the intent of the task, producing irrelevant text descriptions instead of the desired script. This occurred because they did not successfully retrieve the pertinent script information.

We also validate the effectiveness of each component in our approach. It is found that our query classification improves the average accuracy of the generated answers by 4.96%. This improvement is attributed to query classification reducing interference from irrelevant reference text chunks. Notably, since the answers for script-related tasks may not be limited to plain text style, performing classification in advance is a crucial preprocessing step for script retrieval and annotation tasks, leading to accuracy increases of 9.3% and 6.23%, respectively. In the retrieval stage, our hybrid retrieval method improves overall quality by 1.21%, with a significant 4.1% improvement in the script annotation task, compared to common dense retrieval results. This demonstrates that our hybrid retrieval enhances text diversity and hit rate by integrating multiple retrieval methods. For post-retrieval processing, our reranker contributes to an overall quality increase of 1.9%, with a more pronounced increase of 4.2% in high-difficulty tasks. This improvement is due to the reranker optimizing the ranking process, thereby enhancing the recall of high-quality reference texts.

7.2 RAG Verification in RTL Code Generation

For testing dataset of RTL code, we curate a benchmark comprising 25 diverse designs. The scales and complexities of the testing dataset refer to the “RTLLM” [22] and “RTLCoder” [21] methods.

The accuracy of RTL code cannot simply be evaluated by similarity metrics, as RTL code demands high precision in functionality and has specific definitions for inputs and outputs. As shown in Figure 11, the logical structure of RTL code is extremely rigorous, and any slight differences can lead to significantly different output results. In particular, some special symbols might be overlooked or misinterpreted when calculating similarity, yet these subtle differences in symbols can significantly impact the final function. As shown in Figure 11, using BERTScore [55] to evaluate the accuracy of RTL code may introduce errors in such cases.

Table 3. Performance Comparison on RTL Code Generation

Design	RTLCoder-DeepSeek		VeriGen		EAR-DeepSeek	
	Syn-VCS	Func	Syn-VCS	Func	Syn-VCS	Func
add_sub_split	0/5	0/5	0/5	0/5	1/5	1/5
adder_4bit	5/5	0/5	5/5	0/5	3/5	3/5
adder_8bit	0/5	0/5	0/5	0/5	0/5	0/5
arithmetic_unit_4bit	2/5	1/5	5/5	0/5	3/5	0/5
barrel_shifter_encoder	0/5	0/5	5/5	5/5	4/5	4/5
binary_counter_mod	0/5	0/5	0/5	0/5	4/5	3/5
binary_to_bcd	0/5	0/5	0/5	0/5	0/5	0/5
bor_circuit	5/5	5/5	5/5	0/5	4/5	4/5
decalper_eb_ot	4/5	0/5	5/5	0/5	2/5	2/5
fifo_64in_64out_8kb	5/5	5/5	0/5	0/5	5/5	5/5
max_value_pipelined	0/5	0/5	0/5	0/5	5/5	5/5
modbus_slave	1/5	1/5	0/5	0/5	1/5	1/5
mux32	5/5	5/5	0/5	0/5	5/5	5/5
mux4to1_priority_encoder	5/5	4/5	5/5	0/5	4/5	4/5
or4_gate	5/5	5/5	5/5	5/5	5/5	5/5
PLL40_2	4/5	4/5	5/5	5/5	4/5	4/5
reverse_byte_order	5/5	4/5	0/5	0/5	5/5	4/5
shift_register_4bit	5/5	0/5	5/5	0/5	5/5	5/5
signal_combiner	5/5	3/5	5/5	0/5	3/5	2/5
SPI_Controller	3/5	1/5	0/5	0/5	4/5	3/5
split_16bit_input	5/5	0/5	0/5	0/5	5/5	3/5
splitter	0/5	0/5	0/5	0/5	2/5	2/5
twos_complement	5/5	5/5	0/5	0/5	5/5	5/5
up_counter_4bit	5/5	5/5	5/5	5/5	4/5	4/5
axi_gpio_interface	0/5	0/5	0/5	0/5	4/5	4/5
Success rate	68%	52%	44%	16%	**92%**	**88%**

Therefore, we follow the RTLCoder [21] for the correctness evaluation of the generated RTL, specifically by applying Synopsys VCS [41]. This approach ensures that the generated Verilog code adheres to syntax rules and that the designed interfaces align with the testbench, thereby guaranteeing the successful synthesis of the circuit.

At last, testbench is also applied for simulation evaluation refer to Reference [22]. Encompassing multiple test cases, this testbench file defines input stimuli and corresponding correct output values aligned with the design description. It serves as a rigorous validation tool for design correctness.

To ensure a fair comparison, our applied LLM and other models [21, 43] are configured with similar parameters. In our experiments, we standardize the parameters for all models: *top_p = 0.95*, *temperature = 0.5*, and *max_new_tokens = 2,048*. These parameters collectively contribute to generating coherent, relevant, and concise outputs while maintaining a balance between diversity and consistency. Given that all models have comparable parameter sizes, ranging from 6B to 6.7B, the fairness of the experimental results is not compromised by disparities in model capabilities.

Table 3 presents the comparison of RTL generation solutions. For the syntax correction ratio, EAR achieved an 92% success rate, which is 24% and 48% higher than RTLCoder and VeriGen, respectively. It can be indicated that our retrieval strategy has enhanced the model's ability to produce syntactically correct RTL. As our EAR can retrieve code examples that serve as a form of few-shot learning, providing highly relevant design cases to guide the LLM in RTL generation, and significantly reducing instability in the model outputs.

In the functionality evaluation, our EAR can achieve an 88% success rate, which is 36% and 72% higher than RTLCoder and VeriGen, respectively. This improvement is attributed to the high relevance of the retrieved content, making it easier for the LLM to generate the correct functionality. For example, as shown in Figure 12, for an RTL generation, EAR outperforms these non-commercial baseline models across all metrics on both benchmarks, despite not being trained on publicly available academic datasets such as RTLCoder and VeriGen.

DESIGN REQUIRMENT PROMPT

Please act as a professional Verilog designer. Create a module that implements a 4-bit shift register with parallel load functionality. The module should have the following inputs and outputs:

Inputs:
- `clk`: clock signal
- `reset_n`: asynchronous reset signal (active low)
- `load_en`: parallel load signal (active high)
- `parallel_data`: 4-bit input data to be loaded into the shift register

Outputs:
- `shifted_data`: 4-bit output data from the shift register

The shift register should shift data on the rising edge of the clock signal. When the load_en signal is high, the input data should be loaded into the shift register on the next rising edge of the clock signal. When the reset_n signal is low, the shift register should be reset to all zeros.

Design a module that implements the 4-bit shift register with parallel load functionality and meets the requirements above. The module should be implemented within 200 lines of Verilog code.

RTL-CODER VERILOG CODE Syn: ✓ Func: ⚠

```
// 4-Bit Shift Register with Parallel Load
// This module implements a 4-bit shift register with the ability to load data from a parallel input.
// It shifts the data left on each rising edge of the clock signal, with the option to load parallel
// data based on a control signal. The module also supports an asynchronous reset to clear the register.

module shift_register_4bit (
    input clk,                // Clock signal: Synchronizes data shifting and loading
    input reset_n,            // Asynchronous reset signal (active low): Clears the register
    input load_en,            // Parallel load enable signal (active high): Enables parallel data loading
    input [3:0] parallel_data, // 4-bit parallel input data: Data to load into the shift register
    output reg [3:0] shifted_data // 4-bit output data: Current state of the shift register
);
    // Internal 4-bit shift register
    reg [3:0] shift_reg;
    // Always block for asynchronous reset, data loading, and shifting operations
    always @(posedge clk or negedge reset_n) begin
        if (~reset_n) begin
            shift_reg <= 4'b0; // Asynchronous reset: Clear shift register to zero
        end else if (load_en) begin
            shift_reg <= parallel_data; // Load parallel_data into shift register when load_en is high
        end else begin
            shift_reg <= {shift_reg[2:0], shift_reg[3]}; // Error: Circular shift
            // This line performs a circular shift, which moves the MSB to the LSB.
            // To align with typical shift register behavior, the correct line should be:
            // shift_reg <= {shift_reg[2:0], 1'b0};
            // This correction will insert a '0' into the LSB during the shift.
        end
    end
    // Assign the internal shift register's content to the output
    assign shifted_data = shift_reg;
endmodule
```

VERIGEN VERILOG CODE — Syn: ✓ Func: ⚠

```
// 4-Bit Shift Register with Parallel Load
// This module implements a 4-bit shift register with the ability to load data from a parallel input.
// It supports an asynchronous reset to clear the register and loads new data based on a control signal.
// The module operates on the rising edge of the clock signal.

module shift_register_4bit(
    input clk,              // Clock signal: Synchronizes data loading and operations
    input reset_n,          // Asynchronous reset signal (active low): Clears the register when low
    input load_en,          // Parallel load enable signal (active high): Enables loading of parallel data
    input [3:0] parallel_data, // 4-bit parallel input data: Data to load into the shift register
    output reg [3:0] shifted_data // 4-bit output data: Current state of the shift register
);

    // Always block for handling asynchronous reset and parallel data loading
    always @(posedge clk) begin
        if (!reset_n) begin
            shifted_data <= 4'b0000; // Reset the output to zero when reset_n is low
        end else if (load_en) begin
            shifted_data <= parallel_data; // Load parallel_data into shifted_data when load_en is high
        end
        // Error: Missing logic for shifting data when load_en is low and reset_n is high
        // In a complete shift register design, there should be a mechanism to shift data left or right
        // when not loading new data or resetting.
    end
endmodule
```

EAR-DEEPSEEK VERILOG CODE — Syn: ✓ Func: ✓

```
// 4-Bit Shift Register with Parallel Load
// This module implements a 4-bit shift register with the ability to load data from a parallel input.
// It supports an asynchronous reset to clear the register, parallel data loading, and data shifting.

module shift_register_4bit(
    input clk,              // Clock signal: Synchronizes data shifting and loading
    input reset_n,          // Asynchronous reset signal (active low): Clears the register when low
    input load_en,          // Parallel load enable signal (active high): Enables loading of parallel data
    input [3:0] parallel_data, // 4-bit parallel input data: Data to load into the shift register
    output reg [3:0] shifted_data // 4-bit output data: Current state of the shift register
);

    // Always block for handling asynchronous reset, parallel data loading, and shifting
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            shifted_data <= 4'b0; // Reset the output to zero when reset_n is low
        end
        else if (load_en) begin
            shifted_data <= parallel_data; // Load parallel_data into shifted_data when load_en is high
        end
        else begin
            shifted_data <= {shifted_data[2:0], 1'b0}; // Shift data left and insert 0 at the LSB
        end
    end
endmodule
```

Fig. 12. Prompt and Verilog results for 4-bit shifter from all solutions.

The above RTL code evaluations focus on the correctness of responses to one-turn queries. To eliminate logical errors, it is recommended to use a multi-turn interaction mechanism to address issues. For example, it can involve iteratively utilizing compiler error logs and retrieved human guidance from the database as feedback. This interactive debugging loop can be repeated iteratively until all errors are resolved.

7.3 Discussion

Our ultimate goal is to enable efficient interactions with EDA tools, empowering designers to perform their tasks with greater effectiveness. At present, our efforts focus on enhancing the accuracy of query responses and the correctness of functionality code, which are foundational to providing robust EDA assistance.

Given the highly customized nature of API structures across different EDA tools, it will be crucial in the future to incorporate dedicated API corpora for seamless integration with specific platforms and tools. Additionally, since practical design workflows typically involve iterative interactions with tools, supporting complex and dynamic scenarios will require the development of multi-turn query capabilities and mechanisms to retain historical interaction contexts. Therefore, it can include a context management module in the future that continuously tracks changes in design requirements and constraints throughout the design process. By associating query results with new inputs, this module will enable the framework to provide more coherent guidance and globally optimized assistance.

8 Conclusion

This work underscores the increasing prominence of LLMs in EDA tasks. We introduce EDA-Adaptive RAG for retrieval enhancement, encompassing the entire RTL-to-GDSII flow. It addresses the implicit semantics of EDA data and facilitates efficient knowledge retrieval, significantly enhancing LLMs' ability to acquire EDA knowledge. This approach enables a comprehensive evaluation of EDA Q&A tasks across multiple dimensions, including terminology explanation, knowledge advice, and script retrieval and annotation. Furthermore, to explore the generalization capability of our method in EDA design, we propose and evaluate the RTL code generation method with RAG incorporation. Our EDA-Adaptive RAG is evaluated on EDA Q&A tasks and the EDA application of RTL code generation. The final experiments show that, compared with the latest works, our method outperforms them in terms of both answer stability and code quality.

References

[1] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The 12th International Conference on Learning Representations.*

[2] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[4] Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-Embedding: Multi-Linguality, Multi-Functionality, Multi-Granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024. Association for Computational Linguistics*, 2318–2335.

[5] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *ArXiv* abs/2401.14196 (2024).

[6] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations.*

[7] Timothy R. Edwards. 2024. Magic VLSI layout tool. Retrieved from https://github.com/RTimothyEdwards/magic
[8] Timothy R. Edwards. 2024. Qflow: Full end-to-end digital synthesis flow for ASIC designs. Retrieved from https://github.com/RTimothyEdwards/qflow
[9] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, and Stefan De Gendt. 2023. A Deep Learning Framework for Verilog Autocompletion Towards Design and Verification Automation. arXiv:2304.13840.
[10] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. 2023. LLM4SecHW: Leveraging domain-specific large language model for hardware debugging. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST'23)*. 1–6.
[11] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 1762–1777.
[12] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: Retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*. Article 368, 10 pages.
[13] Yukun Huang, Yanda Chen, Zhou Yu, and Kathleen McKeown. 2022. In-context Learning Distillation: Transferring Few-shot Learning Ability of Pre-trained Language Models. *ArXiv* abs/2212.10670 (2022).
[14] Aviral Kaintura, Palaniappan R, Shui Song Luar, and Indira Iyer Almeida. 2024. ORAssistant: A Custom RAG-based Conversational Assistant for OpenROAD. *arXiv preprint arXiv:2410.03845* (2024).
[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.
[16] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktaschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*. 16 pages.
[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktaschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[18] Chaofan Li, Zheng Liu, Shitao Xiao, and Yingxia Shao. 2023. Making large language models a better foundation for dense retrieval. arXiv:cs.CL/2312.15503
[19] Yunxiang Li, Zihan Li, Kai Zhang, Ruilong Dan, Steve Jiang, and You Zhang. 2023. ChatDoctor: A medical chat model fine-tuned on a large language model Meta-AI (LLaMA) using medical domain knowledge. *Cureus* 15, 6 (2023), 40895.
[20] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. *ArXiv* abs/2311.00176 (2023).
[21] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLCoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution. In *IEEE LLM Aided Design Workshop (LAD)*. 1–5.
[22] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An open-source benchmark for design RTL generation with large language model. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE Press, 722–727.
[23] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 5303–5315.
[24] Joao Marques-Silva and Terry Glass. 1999. Combinational equivalence checking using satisfiability and recursive learning. In *Conference on Design, Automation and Test in Europe (DATE'99)*. IEEE Computer Society, 33–38.
[25] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make Every Move Count: LLM-based High-Quality RTL Code Generation Using MCTS. *arXiv preprint* arXiv:2402.03289 (2024).
[26] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Comput. Surv.* 56, 2, Article 30 (2023), 40 pages.
[27] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*.
[28] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., Article 2011, 15 pages.

[29] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. DAVE: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 27–32.
[30] Amaranth Project. 2024. Amaranth: A modern hardware definition language and toolchain based on Python. Retrieved from https://github.com/amaranth-lang/amaranth
[31] Symbiotic Project. 2024. SymbiYosys: Front-end for Yosys-based formal verification flows. Retrieved from https://github.com/YosysHQ/sby
[32] Verilator Project. 2024. Verilator: Open-source SystemVerilog simulator and lint system. Retrieved from https://github.com/verilator/verilator
[33] KLayout Project. 2024. KLayout main sources. Retrieved from https://github.com/KLayout/klayout
[34] OpenROAD Project. 2024. OpenROAD's unified application implementing an RTL-to-GDS flow. Retrieved from https://github.com/The-OpenROAD-Project/OpenROAD
[35] Yosys Project. 2024. Yosys Open SYnthesis Suite. Retrieved from https://github.com/YosysHQ/yosys
[36] Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Found. Trends® Inf. Retriev.* 3, 4 (2009), 333–389.
[37] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. 2024. RAPTOR: Recursive abstractive processing for tree-organized retrieval. In *The 12th International Conference on Learning Representations*.
[38] Utsav Sharma, Bing-Yue Wu, Sai Rahul Dhanvi Kankipati, Vidya A. Chhabria, and Austin Rovinski. 2024. OpenROAD-Assistant: An open-source large language model for physical design tasks. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. Article 22, 7 pages.
[39] Shuting Wang, Jiongnan Liu, Jiehan Cheng, Yuqi Fu, Peidong Guo, Kun Fang, Yutao Zhu, and Zhicheng Dou. 2024. DomainRAG: A Chinese Benchmark for Evaluating Domain-specific Retrieval-Augmented Generation. arXiv:2406.05654.
[40] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics* 11 (2023), 1–17.
[41] Synopsys. 2021. VCS® functional verification solution. Retrieved from https://www.synopsys.com/verification/simulation/vcs.html
[42] Yechun Tang, Xiaoxia Cheng, and Weiming Lu. 2022. Improving complex knowledge base question answering via question-to-action and question-to-question alignment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 137–147.
[43] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A large language model for Verilog code generation. *ACM Trans. Des. Autom. Electron. Syst.* 29, 3 (2024), 31.
[44] Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *ArXiv* abs/2307.09288 (2023).
[45] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically fixing RTL syntax errors with large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. 6 pages.
[46] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *61st Annual Meeting of the Association for Computational Linguistics*. 13484–13508.
[47] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.
[48] Stephen Williams. 2024. Icarus Verilog: Open-source Verilog simulation and synthesis tool. Retrieved from https://github.com/steveicarus/iverilog
[49] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45.
[50] Bing-Yue Wu, Utsav Sharma, Sai Rahul Dhanvi Kankipati, Ajay Yadav, Bintu Kappil George, Sai Ritish Guntupalli, Austin Rovinski, and Vidya A. Chhabria. 2024. EDA corpus: A large language model dataset for enhanced interaction with OpenROAD. In *1st IEEE International Workshop on LLM-Aided Design (LAD'24)*. IEEE, New York, NY.
[51] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2024. ChatEDA: A large language model powered autonomous agent for EDA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 10 (2024), 3184–3197.

[52] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining HDL debugging with large language models. *ArXiv* abs/2403.11671 (2024).

[53] Yuan Pu, Zhuolun He, Tairu Qiu, Haoyuan Wu, and Bei Yu. 2024. Customized retrieval augmented generation and benchmarking for EDA tool documentation QA. In *2024 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

[54] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997.

[55] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating text generation with BERT. In *International Conference on Learning Representations*.

Received 31 July 2024; revised 17 January 2025; accepted 17 January 2025