ChatLS: Multimodal Retrieval-Augmented Generation and Chain-of-Thought for Logic Synthesis Script Customization

Haisheng Zheng [1†] Haoyuan Wu [2] Zhuolun He [2,3†]
[1] Shanghai Artificial Intelligence Laboratory [2] The Chinese University of Hong Kong [3] ChatEDA Technology

***Abstract*—Large Language Models (LLMs) have demonstrated significant potential in automating the Electronic Design Automation (EDA) process through effective integration with EDA tools. This paper targets the customization of logic synthesis scripts, which is crucial for accommodating the unique characteristics of each design in the EDA workflow. The proposed framework, called ChatLS, integrates multimodal retrieval-augmented generation (RAG) and chain-of-thought (CoT) reasoning, enabling LLMs to collaboratively analyze design features and precisely customize synthesis scripts. Experimental results demonstrate that ChatLS has achieved superior performance in customizing synthesis scripts with a commercial logic synthesis tool.**

I. Introduction

Electronic design automation (EDA) encompasses a suite of software tools designed for the creation, analysis, and verification of electronic systems. These tools have evolved to support intricate design flows and cater to the complex requirements of advanced semiconductor manufacturing. To enhance the user experience and development efficiency, application engineers are deployed by vendors to offer tailored on-site support. However, the significant costs associated with training and manpower have led to considerations of automating support services, eliminating the need for human involvement.

Recent advancements in large language models (LLMs) have prompted the proposal of various techniques that utilize LLMs to enhance the usability of EDA tools. For instance, addressing the challenge of navigating extensive and intertwined functionalities in EDA tool documentation, several studies [1]–[3] have developed customized retrieval-augmented generation (RAG) frameworks, which improve the accuracy of LLM-generated responses, enabling the efficient retrieval of relevant information to resolve user queries. Additionally, the complexity of numerous EDA commands can overwhelm users. In response, studies [4], [5] have fine-tuned LLMs specifically for the EDA domain, facilitating the automated generation of tool call scripts based on user requirements.

While these developments represent significant advances in harnessing LLMs to assist users with EDA tools, there remains room for further enhancement. As for logic synthesis tools, a crucial aspect is the integration of user designs and target libraries with insights derived from logic synthesis tool reports. Commercial logic synthesis tools [6], [7] offer numerous optional commands aimed at improving design quality, which need to be selected based on user preferences and the analysis of the design [8], [9]. Additionally, the constraint settings included in synthesis scripts must be based on the characteristics of the design and the target library [10]. Moreover, logic synthesis is generally an iterative process, not a one-time execution. After the initial synthesis, users may employ resynthesis strategies to resolve violations within the netlist or to enhance netlist quality, with strategy choices informed not only by the design and target library but also by insights from logic synthesis tool reports [10]. Therefore, the customization of synthesis scripts, informed by a comprehensive understanding of the designs, target libraries, and the feedback from logic synthesis tool reports, is essential.

† Corresponding Authors.

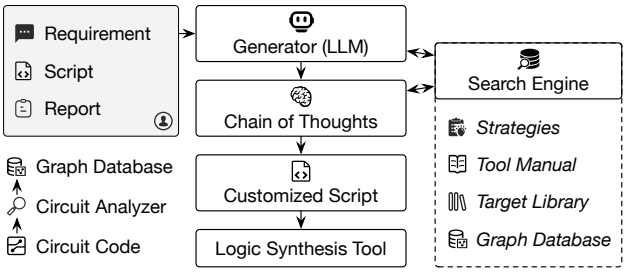


Fig. 1 An Illustration of ChatLS.

To achieve this goal, several challenges need to be addressed. The primary challenge involves a comprehensive analysis of design characteristics at both global and local levels to inform script creation. Global characteristics, such as the architecture and hierarchy of the design, guide the selection of appropriate compilation strategies [10], while local characteristics, including critical path analysis, inform constraint specifications in the scripts. Recent research [11]–[13] has demonstrated the capability of LLMs to understand register-transfer level (RTL) code snippets, suggesting potential applications in analyzing circuit design characteristics for customizing scripts. However, LLMs have significant limitations when processing long-context inputs, which affects their effectiveness in analyzing large codebases [14] and executing lengthy reasoning sequences [15].

Numerous logic synthesis algorithms [16]–[18] use graph-based representations of circuit designs to facilitate analysis and optimization. This graph-based representation allows for efficient exploration of design characteristics through a structured representation of the topology. Additionally, the graph-based representations of circuit designs have enabled numerous studies to leverage graph neural networks (GNNs) to extract meaningful characteristics from complex circuits [19]–[24]. Thus, leveraging graph-based analysis methods may help the LLMs overcome challenges in understanding complex circuit designs.

The second major challenge involves selecting synthesis commands within scripts that align with specific design characteristics. For instance, consider a scenario where a design requires timing optimization. Techniques such as retiming [25] and buffer balancing, both integrated into logic synthesis tools, can effectively reduce path delays. However, choosing the appropriate technique requires thorough consideration. Retiming is particularly effective for designs with extended critical paths that experience timing violations due to unbalanced register placement or excessively long combinational logic. Alternatively, buffer balancing is advantageous for mitigating timing issues in high-fanout nets. Therefore, a careful assessment of the strengths of each technique is essential for optimizing timing.

Techniques like retrieval-augmented generation (RAG) [26] and chain-of-thought (CoT) [27] reasoning have proven effective in assisting LLMs with complex decision-making tasks. In our context, RAG could help the LLMs retrieve relevant information and context

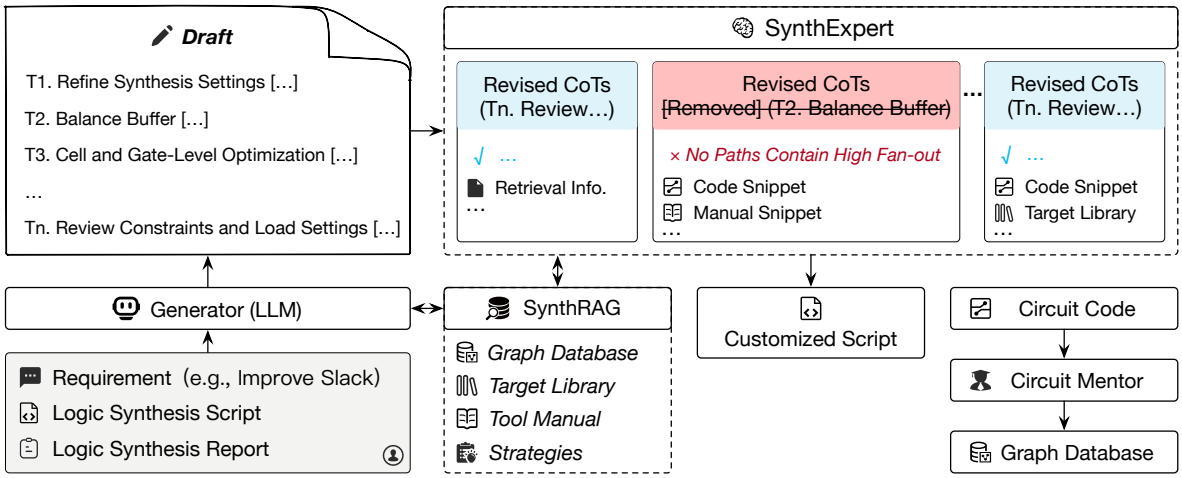


Fig. 2 The Overall Flow of ChatLS.

from circuit design snippets, target library, tool manuals, or past use cases, providing a well-informed foundation for decision-making. CoT then enables step-by-step reasoning to evaluate the trade-offs and select the most suitable solution from multiple options. It guides the LLMs through a logical progression of decisions, considering the impact of each command on design performance, ensuring that the final synthesis script is both optimal and aligned with user preferences.

In this paper, we present ChatLS, a framework for customizing logic synthesis scripts to meet specific requirements expressed in natural language. As illustrated in Fig. 1, ChatLS consists of four main components: a circuit analyzer for assisting LLM analyze and understand circuit designs, a search engine for retrieving relevant information, a CoT for decision-making during script refinement, and an LLM that interprets and generates synthesis scripts from natural language instructions.

Our major contributions are summarized as follows:

- We propose a graph-based method that enhances the ability of LLMs to analyze and understand circuit designs by transforming them into graph databases, with GNN supporting the analysis process.
- We introduce a domain-specific multimodal RAG framework that efficiently retrieves information to enhance LLMs for customizing logic synthesis scripts.
- We present a task-specific CoT mechanism, in tandem with the multimodal RAG framework, that enables iterative refinement of synthesis scripts by revising reasoning steps based on retrieved data.
- We conduct comprehensive experiments, demonstrating that ChatLS achieves superior performance in customizing logic synthesis scripts.

The remainder of this paper is structured as follows: Section II provides the preliminary background. Section III presents the problem formulation. The detailed description of the ChatLS framework is discussed in Section IV. The experimental results are presented and analyzed in Section V. Finally, conclusions are drawn in Section VI.

II. Preliminaries

Abstract Syntax Tree (AST) represents the hierarchical structure of source code, where each node corresponds to a specific construct or element in the program. It enables efficient analysis, transformation, and manipulation by abstracting the logical structure while omitting unnecessary syntactic details. ASTs are fundamental in compiler design, serving as an intermediate representation for syntax analysis, semantic validation, and optimization. They are also extensively used in static code analysis, automated reasoning, and code refactoring, effectively bridging raw code with its logical representation and enhancing software quality.

Retrieval-Augmented Generation (RAG) [26] targets the problem of enhancing natural language generation by combining the strengths of retrieval-based and generative approaches to improve both knowledge and contextual accuracy, making it effective for LLMs. RAG augments the generative process with a retriever module that dynamically retrieves relevant information $\{r_i\}_{i=1}^{k}$ from an external resource $\mathcal{R}$, such as a document corpus or knowledge base, based on an input query q. The retrieved information is then utilized by a generative model to produce a response y. In RAG, the objective is to maximize the conditional probability $P(y|q)$, which can be expressed as:

$$P(y|q) = \sum_{i=1}^{k} P(y|r_i, q)P(r_i|q), \tag{1}$$

where $P(r_i|q)$ represents the probability of retrieving a piece of information r_i given the query q, and $P(y|r_i, q)$ is the probability of generating the output y conditioned on both the query and a specific retrieved piece of information. By retrieving diverse and relevant information, RAG grounds the LLMs in external knowledge, thereby improving response quality and factual accuracy in complex tasks.

Chain of Thoughts (CoT) reasoning [27] enhances the performance of LLMs in tasks requiring complex reasoning, such as multi-step math word problems. Instead of generating the final answer directly, CoT prompts LLMs to generate intermediate steps, called thoughts, that serve as a scratch space guiding the model towards the solution. Given an input x, CoT introduces intermediate steps $s_1, s_2, \ldots, s_k$, forming a path from input to output:

$$x \rightarrow (s_1, s_2, \ldots, s_k) \rightarrow y, \tag{2}$$

this explicit reasoning framework helps LLMs break down complex tasks into manageable components, improving both accuracy and interpretability.

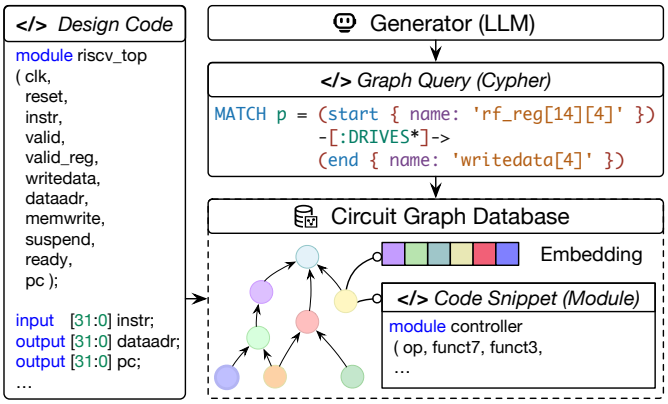


Fig. 3 Workflow Visualization of CircuitMentor.

III. The Scope of ChatLS

ChatLS is an innovative framework that provides a conversational interface, allowing users to employ natural language to customize synthesis scripts tailored to the specific requirements of each design. By interpreting both the design code and user requirements, ChatLS generates synthesis scripts that align with particular design attributes and user preferences.

Problem 1 (Objective of ChatLS)**.** *Design methodologies to enhance the capability of LLMs in customizing logic synthesis scripts to meet user requirements.*

IV. Algorithms

Fig. 2 illustrates the overall workflow of ChatLS. The process begins with the user providing the requirements, circuit design code, corresponding logic synthesis scripts, and reports generated by the logic synthesis tool. Based on this input, the following steps are carried out to generate a customized logic synthesis script for the design:

- **CircuitMentor** is a circuit design analysis tool that transforms the circuit design into a graph representation and employs GNN to extract high-level features from the circuit design.
- **SynthRAG** is a multi-modal RAG framework used to retrieve relevant information for LLMs when customizing logic synthesis scripts.
- **Generator (LLM)** drafts a customized logic synthesis script based on user requirements, incorporating the reports from the logic synthesis tool, the compilation and optimization strategies retrieved by **SynthRAG**, and high-level information matched by **SynthRAG** using embeddings extracted from the circuit design by **CircuitMentor**.
- **SynthExpert** is a CoTs mechanism that uses **SynthRAG** to fetch additional information and iteratively evaluates the suitability of the drafted logic synthesis script for the circuit design. During this process, **SynthRAG** is prompted to provide code for netlist paths, high-level information for corresponding modules, usage details for commands in the logic synthesis manual, and specifics of the target library. These elements ensure the generated logic synthesis script complies with design requirements and command specifications.

A. CircuitMentor: Graph-Based Assistant for Circuit Analysis

Circuit Representation. LLMs encounter significant challenges in comprehending complex circuit code, as discussed in Section I. Given that circuits can naturally be represented as graphs, we introduce CircuitMentor, which exploits this graph-based representation to enhance the capacity of LLMs to customize logic synthesis scripts.

As visualized in Fig. 3, the circuit code is transformed into a graph, specifically an AST. This graph structure is hierarchical, with nodes representing different levels of the circuit design. At the highest level, the graph contains nodes representing the overall design, with subnodes corresponding to individual modules. Each module node stores its associated Verilog code, enabling LLMs to analyze the structure and functionality of paths when customizing scripts. Within each module, subnodes represent the circuit components, such as gates and registers. This hierarchical organization allows for efficient traversal and multi-level analysis of the design. Customizing logic synthesis scripts often requires analyzing critical paths within the circuit. Therefore, our goal is to identify an appropriate graph database to store the circuit graph, enabling efficient path extraction and analysis. Neo4j [28] is well-suited for this purpose, as it supports Cypher [29], a query language that can be leveraged to extract paths and subgraphs from circuit graphs, facilitating efficient graph traversal and analysis.

Global Circuit Feature Extraction. Understanding the characteristics of a circuit is crucial for customizing logic synthesis scripts. Different circuit components, such as arithmetic and memory-based modules, require distinct compilation strategies. Arithmetic components focus on optimizing speed and area, while memory components prioritize access time and power efficiency. Tailoring the compilation strategy to the function of each component is essential for achieving overall circuit optimization. Nevertheless, effectively conveying these characteristics through LLMs presents significant challenges, as previously outlined in Section I.

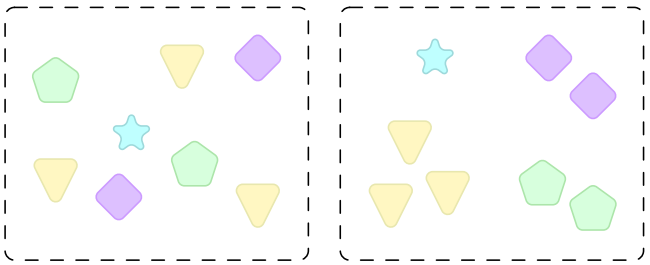
Since the circuit can be represented as a graph, GNNs can be employed to extract meaningful features from its design. We propose a hierarchical GNN based on GraphSAGE [30] to learn embeddings for the circuit. By treating each module as a subgraph, our approach captures the unique characteristics of each module. This hierarchical approach is crucial for capturing both local and global structural patterns within the circuit design. GraphSAGE aggregates features from neighboring nodes and their respective subgraphs, facilitating the extraction of hierarchical features. The key operation of GraphSAGE is defined as:

$$\boldsymbol{h}_v^{(k)} = \sigma\left(\boldsymbol{W}^{(k)} \cdot \text{Aggregator}\left(\{\boldsymbol{h}_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)\right), \quad (3)$$

where $\boldsymbol{h}_v^{(k)}$ represents the embedding of node v at the k-th iteration, $\mathcal{N}(v)$ denotes the set of neighboring nodes of v, and Aggregator refers to the aggregation function. This aggregation step effectively encodes local circuit structural features into a compact and informative node representation.

The type of circuit design also plays a crucial role in customizing logic synthesis scripts. In cases where the design consists of a single module or has been flattened for synthesis, the graph representation collapses into a single node, rendering it ineffective for GNN-based operations. To address this, global pooling strategies are employed to obtain a global embedding. The global embedding is computed as the mean of all node embeddings in the design, given by $\boldsymbol{z}global = \frac{1}{N}\sum i = 1^N \boldsymbol{h}_i$. where N is the total number of modules in the circuit. This ensures that even when the design is flattened or consists of a single module, a meaningful global feature representation for the entire design can still be derived.

Metric Learning. Since our hierarchical GNN aims to extract



(a) (b)

Fig. 4 Illustration of the Metric Learning Process. (a) Initially, the embeddings are randomly scattered across the space. (b) After training, embeddings of similar data points converge, while embeddings of dissimilar data points diverge, forming distinct clusters.

embeddings from modules in the circuit graph, we need first address the challenge that topology-based approaches may fail to identify identical components due to structural variations. Given that GNNs are inherently topology-dependent, this poses a considerable challenge.

To address this, we adopt a metric learning approach, which focuses on learning a similarity function or distance metric that ensures similar designs are close in the embedding space, while dissimilar designs are placed farther apart. Various loss functions, such as contrastive loss [31] and multi-similarity loss [32], are commonly used to optimize the embedding space. These methods encourage the model to learn an embedding space where the geometry of the embeddings aligns with the desired similarity relationships between designs. Fig. 4 illustrates the evolution of embeddings during training. Initially, the embeddings are dispersed across the space, but after training, similar designs are pulled closer together, while dissimilar ones are pushed farther apart. This enhances the discriminative power of the embeddings, making them more effective for tasks such as design retrieval and similarity comparison.

By learning these discriminative embeddings, we capture the underlying structure of designs, enabling the efficient retrieval of relevant circuit designs based on their embeddings.

B. SynthRAG: Multimodal RAG for Domain-Specific Retrieval

The RAG framework substantially improves the generative capabilities of LLMs. Building on this advancement, we present SynthRAG, a domain-specific, multimodal RAG framework designed to enhance the ability of LLMs in customizing logic synthesis scripts. To support this goal, we first establish a database that stores the relevant information. TABLE I provides a summary of the different retrieval types, their corresponding representations, and the query methods employed in our approach.

Graph Embedding-Based Retrieval. In SynthRAG, graph embeddings serve as the foundation for information retrieval, linking related circuit designs with their respective compilation and optimization strategies. These embeddings are computed using CircuitMentor, which generates vector representations of circuit designs or modules. The resulting embeddings are used to query the graph database, retrieving relevant circuit designs with their respective compilation and optimization strategies.

Given a query embedding, $\boldsymbol{z}_{query}$, the retrieval process relies on nearest neighbor search over the database of graph embeddings. The nearest neighbor search is formalized as:

$$\boldsymbol{z}_{Retrieved} = \arg\min_{\boldsymbol{z}_{db}} \left(\text{sim}(\boldsymbol{z}_{query}, \boldsymbol{z}_{db})\right), \tag{4}$$

TABLE I Summary of Query Methods.

Category	Representation	Query Method	Retrieval Content
High Level Information of Circuit Design	Graph Embedding	Join	Compile Strategy Optimization Strategy
Code of Circuit Design	Graph Structure	Direct	The code of the module where the path is located
Target Library	Graph Structure	Direct	Gate Cell Information
Logic Synthesis Tool User Manual	LLM Embedding	Direct	Command Usage Command Requirement

where $\boldsymbol{z}_{db}$ represents the embeddings of designs in the database, and sim denotes the cosine similarity function. This ensures that the most semantically similar designs to the query are retrieved, facilitating efficient information retrieval for logic synthesis.

In RAG frameworks, reranking plays a crucial role in refining retrieved results to better fit the requirements of the target domain. Existing reranking methods [33], [34], commonly used in text-based retrieval, cannot be directly applied to our graph embeddings due to the unique characteristics of circuit designs. For instance, although both an arithmetic logic unit (ALU) and a systolic array are categorized as arithmetic components, they differ considerably in terms of scale and complexity. To address this, we introduce a domain-specific reranking function that incorporates additional criteria, such as timing, area, and power consumption, to reorder the retrieved graph embeddings.

Let $\boldsymbol{z}_{retrieved} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_K\}$ represent the set of Top-K retrieved embeddings, and let $c_1, c_2, \ldots, c_K$ denote their corresponding performance metrics (e.g., timing, area, power). The reranking function, $f(\boldsymbol{z}_{retrieved}, \boldsymbol{c})$, computes a new ranking score for each design by considering both the similarity of embeddings and the additional characteristics. The reranking is computed as:

$$\text{Score}(\boldsymbol{z}_i) = \alpha \cdot \text{sim}(\boldsymbol{z}_{query}, \boldsymbol{z}_i) + \beta \cdot c_i, \tag{5}$$

where $\text{sim}(\boldsymbol{z}_{query}, \boldsymbol{z}_i)$ is the cosine similarity between the query embedding $\boldsymbol{z}_{query}$ and the retrieved embedding $\boldsymbol{z}_i$, c_i is the domain-specific characteristic for design i, and α and β are weighting parameters that balance the contribution of similarity and characteristics in the final ranking.

The reranking process selects the designs with the highest scores, prioritizing those that meet the functional requirements and resource constraints of the query. This ensures that the retrieved designs are not only semantically similar to the query but are also optimized according to specific design criteria.

Graph Structure-Based Retrieval. In SynthRAG, the graph-based retrieval system utilizes Cypher queries, which can be generated by LLMs, to extract specific design details and information about the target library from the database. Each query retrieves a single result, thereby eliminating the need for reranking. The target library is integrated into the database to avoid inefficiencies associated with directly inputting it into the LLMs for processing.

LLM Embedding-Based Retrieval. Several studies, such as [1], have adapted RAG frameworks to retrieve information from EDA user manuals, which contain hybrid information. In our context, we focus exclusively on retrieving descriptions of logic synthesis commands within the user manual. Accordingly, SynthRAG utilizes the `text-embedding-3-large` model[1] to transform the user manual into text embeddings. Subsequently, to enhance the accuracy

[1]https://platform.openai.com/docs/guides/embeddings

TABLE II Overview of Hardware Designs in the Database.

Category	Components
Processor Core	Rocket [36], Sordor [37]
Machine Learning Accelerator	NVDLA [38], Gemmini [39]
Vector Arithmetic	SIMD [40]
Signal Processing	FFT [41]
Cryptographic Arithmetic	SHA3 [37]

of the retrieved information, SynthRAG employs `GPT-4o` [35] as a reranker.

C. SynthExpert: Iterative Customization with CoT and RAG

Although RAG and CoT techniques can significantly enhance LLMs by facilitating complex decision-making tasks, their integration into ChatLS demands careful consideration.

Despite **RAG** is effective in general applications, it faces limitations in handling complex reasoning tasks requiring multi-step reasoning. These tasks are difficult to encapsulate into simple search queries, often leading to challenges in retrieving relevant information, thus limiting the applicability of RAG. Specifically, RAG approaches typically retrieve all pertinent information in a single step, disregarding the dynamic and iterative nature of information requirements throughout the synthesis script customization process. In our context, logic synthesis script customization involves multi-dimensional data, including design characteristics, user manuals, and the target library. Retrieving all relevant information at once may overwhelm the LLMs, complicating the prioritization and application of the most pertinent data at each reasoning step. Consequently, relying solely on the initial task prompt often lacks the depth needed to effectively guide subsequent reasoning and generation, leading to suboptimal retrieval performance.

Similarly, **CoT** facilitates LLMs in breaking down complex tasks into manageable steps, which is crucial for generating accurate and optimized synthesis scripts. However, without explicit guidance, LLMs may introduce errors due to incomplete domain knowledge [42] or be influenced by hallucinations [43]. In our scenario, these inaccuracies may render customized logic synthesis scripts non-executable, as LLMs could generate nonexistent or incompatible commands for the circuit design. To address these challenges, we propose SynthExpert, which synergistically integrates CoT with RAG for logic synthesis customization. As shown in Fig. 2, SynthExpert mines each reasoning step to generate the final synthesis script.

Given a task prompt I, the LLM initially generates a series of thought steps, represented as $T := \{T_i\}_{i=1}^{n}$, where T_i is the i-th thought step. In SynthExpert, these steps correspond to intermediate reasoning stages, where the synthesis script is drafted based on user inputs. Recognizing potential inaccuracies in these steps, SynthExpert utilizes RAG to revise each step sequentially.

For each thought step T_i, the LLM formulates a query Q_i by incorporating the task prompt I to retrieve information R_i pertinent to this step, thereby refining T_i in the subsequent reasoning phase. Once Q_i is formulated, the retrieval mechanism fetches the relevant information $R_i = \text{Retrieve}(Q_i)$. The retrieved information R_i is then used to revise the current thought step, producing a revised thought step$T_i^\star$:

$$T_{1:i}^\star = p_\theta(\cdot|I, T_{1:i-1}^\star, T_i, R_i) \tag{6}$$

where $p_\theta(\cdot)$ represents the probability function of the LLM generating the revised thought $T_i^\star$ based on the integrated inputs and retrieved data.

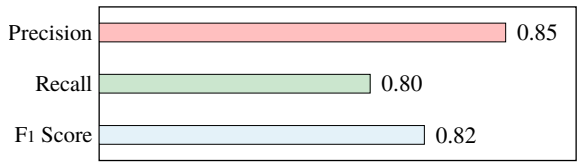


Fig. 5 Performance of SynthRAG.

This approach effectively mitigates the limitations of RAG and CoT by ensuring that each reasoning step is reinforced with the most pertinent and contextually relevant information. By revising each thought step individually, SynthExpert ensures that the LLM receives the appropriate context at every stage of the synthesis script customization process. This sequential retrieval and revision mechanism enhances the relevance and accuracy of the information used, providing richer contextual cues that align with the specialized demands of logic synthesis. Furthermore, focusing on revising individual thought steps minimizes the risk of introducing new errors into previously accurate reasoning steps, thereby preserving the integrity and reliability of the reasoning process.

As illustrated in TABLE III, integrating SynthExpert within ChatLS ensures that each aspect of synthesis script customization is informed by accurate and relevant data, thereby substantially improving the quality and reliability of the customized scripts.

V. Experimental Evaluation

ChatLS was implemented using PyTorch [49], PyTorch Geometric [50], FAISS [51], Neo4j [28], with GPT-4o (Version 2024-08-06) [35] employed as the generator. To construct the database for SynthRAG, numerous open-source designs were synthesized using the `Nangate 45nm`[2] library as the target technology, alongside the `5K_heavy_1k` wireload model, employing various optimization and compilation strategies. The corresponding logic synthesis scripts were then converted to the Design Compiler format, where they are treated as expert drafts. TABLE II presents a selection of hardware designs included in the database. Comprehensive experiments are conducted to evaluate the efficiency of ChatLS in customizing logic synthesis scripts.

A. Evaluation of SynthRAG

To evaluate SynthRAG, experiments were conducted using Chipyard [37] to generate SoCs with various configurations. These configurations provided a diverse set of circuit designs, enabling the assessment of the model's ability to retrieve relevant logic synthesis scripts and optimization strategies. The primary goal was to retrieve designs and components with high similarity to the query while considering resource usage to ensure effective customization of synthesis strategies. The F_1 score was selected for evaluation, as it balances precision and recall, making it well-suited for this task, where both false positives (irrelevant designs) and false negatives (missed relevant designs) can significantly impact performance:

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{7}$$

where Precision is $\frac{TP}{TP+FP}$ and Recall is $\frac{TP}{TP+FN}$. Here, true positive (TP) refers to correct predictions, false positive (FP) to incorrect predictions, and false negative (FN) to missed predictions. The results, summarized in Fig. 5, show that SynthRAG successfully retrieved relevant designs and modules, enabling effective customization of logic synthesis scripts.

[2]https://eda.ncsu.edu/freepdk/freepdk45

TABLE III Performance Comparison for Logic Synthesis Script Customization *(Pass@5)*.

Design	GPT-4o [35]				Claude 3.5 Sonet [44]				ChatLS (Ours)			
	Timing (ns)			Area (μm^2)	Timing (ns)			Area (μm^2)	Timing (ns)			Area (μm^2)
	WNS	CPS	TNS		WNS	CPS	TNS		WNS	CPS	TNS	
aes	-0.17	-0.17	-31.64	16408.21	-0.17	-0.17	-30.70	16126.78	0.00	0.00	0.00	15919.04
dynamic_node	0.00	0.01	0.00	16327.08	0.00	4.06	0.00	20048.95	0.00	4.85	0.00	18907.28
ethmac	-0.55	-0.55	-75.89	80502.77	-0.54	-0.54	-128.99	81993.70	-0.47	-0.47	-55.72	80349.56
jpeg	0.00	0.00	0.00	57227.24	0.00	0.00	0.00	66106.85	0.00	0.00	0.00	67290.02
risv32i	0.00	0.59	0.00	10241.00	0.00	0.60	0.00	9711.66	0.00	0.74	0.00	9329.95
swerve	0.00	0.79	0.00	143557.54	0.00	1.86	0.00	158364.70	0.00	2.05	0.00	143545.30
tinyRocket	-0.21	-0.21	-42.22	35960.81	-0.33	-0.33	-484.71	41999.01	-0.09	-0.09	-14.74	36070.66

TABLE IV Performance Baseline of Various Designs.

Design	Timing (ns)			Area (μm^2)
	WNS	CPS	TNS	
aes [45]	-0.16	-0.16	-31.64	16577.12
dynamic_node [46]	-0.08	-0.08	-0.45	21155.51
ethmac [45]	-0.54	-0.54	-76.55	80533.36
jpeg [45]	-1.17	-1.17	-439.66	107612.16
risc32i [47]	0.00	0.59	0.00	10241.00
swerv [48]	0.00	0.81	0.00	161551.64
tinyRocket [36]	-0.88	-0.88	-1057.89	44231.28

B. Evaluation of ChatLS

Timing optimization is a key focus, as it ensures sufficient slack that can be traded for improvements in area and power without compromising timing closure. Accordingly, the evaluation emphasizes ChatLS ability to customize logic synthesis scripts for optimal timing.

Benchmark and Baseline. The evaluation uses design benchmarks from OpenROAD [47], as summarized in TABLE IV. The Design Compiler was employed as the logic synthesis tool. The `Nangate 45nm` library was used as the target technology, along with the `5K_heavy_1k` wireload model. The original OpenROAD synthesis scripts were adapted to ensure compatibility with Design Compiler [6], and these adapted scripts serve as the baseline for evaluating ChatLS. This baseline provides a reference for assessing improvements or deviations in synthesis performance compared to the proposed methods.

Model Comparison. The performance of ChatLS was compared with two state-of-the-art LLMs: GPT-4o (Version 2024-08-06) [35] and Claude 3.5-Sonet (Version 2024-10-22) [44]. For this evaluation, ChatLS, GPT-4o, and Claude 3.5 each customize the logic synthesis script based on a baseline script, with only a single iteration of customization performed. To ensure a fair and consistent evaluation, all language model baselines were tested using identical prompt engineering techniques, and basic configurations, including the time period, are not allowed to change. The quality of each customized logic synthesis script was evaluated after this iteration. In cases where the design code exceeds the token limits of the models, it was partitioned into segments, each containing no more than 128,000 tokens.

Evaluation Metrics. The evaluation employs the following key metrics to assess the synthesis script quality:

- **Timing**: Measures the ability of the design to meet operational speed requirements, including worst negative slack (WNS), critical path slack (CPS), and total negative slack (TNS).
- **Area**: Assesses the total hardware resources utilized by the synthesized circuit.

These metrics offer a thorough evaluation of the quality of logic synthesis scripts.

Evaluation Results. The evaluation results for ChatLS are summarized in TABLE III. All models, including GPT-4o, Claude 3.5-Sonet, and ChatLS, improve design timing relative to the baseline. ChatLS consistently delivers the best timing performance, showing the most substantial improvements across all evaluated designs. This underscores the superior capability of ChatLS in customizing logic synthesis scripts for timing optimization. Both `ethmac` and `tinyRocket` exhibit timing violations, as only a single iteration was executed. However, logic synthesis is inherently an iterative process, not a one-time execution. Additional iterations are required to further resolve timing issues. While GPT-4o achieves the best area results in some cases, the primary objective of this evaluation is timing improvement, a goal that is not fully satisfied by GPT-4o. In contrast, ChatLS produces the highest-quality synthesis scripts, outperforming the baseline LLMs and demonstrating its effectiveness in customization.

VI. Conclusion

It is crucial to customize synthesis scripts, guided by a thorough understanding of design, target libraries, and feedback from logic synthesis tool reports. To address this need, we propose methods to enhance the ability of LLMs to customize synthesis scripts. To improve the ability of LLMs to understand circuit designs, we introduce CircuitMentor, which efficiently extracts the global characteristics of designs while assisting LLMs in understanding their local characteristics. Additionally, to enhance the ability of LLMs to customize high-quality logic synthesis scripts, we present SynthRAG, which supplies relevant information for script customization. Futhermore, to aid LLMs in selecting the appropriate commands within logic synthesis tools based on design and user specifications, we introduce SynthExpert, which works in tandem with SynthRAG. Experimental results demonstrate that the proposed framework, ChatLS, effectively and efficiently customizes synthesis scripts, aligning with user preferences and optimizing design performance. Future work could explore extending this framework to collaborate with a wider array of logic synthesis tools, such as PrimePower [52], to further advance the automation of the logic synthesis workflow.

Acknowledgment

This work is supported by Shanghai Artificial Intelligence Laboratory (JF-P23KK00071-1-DF). The authors would like to thank Professor Bei Yu at The Chinese University of Hong Kong for helpful discussions and comments.

REFERENCES

[1] Y. Pu, Z. He, T. Qiu, H. Wu, and B. Yu, "Customized Retrieval Augmented Generation and Benchmarking for EDA Tool Documentation QA," in *Proc. ICCAD*, 2024.

[2] L. Shi, M. Kazda, B. Sears, N. Shropshire, and R. Puri, "Ask-EDA: A Design Assistant Empowered by LLM, Hybrid RAG and Abbreviation De-hallucination," in *arXiv preprint*, 2024.

[3] A. Kaintura *et al.*, "ORAssistant: A Custom RAG-based Conversational Assistant for OpenROAD," in *arXiv preprint*, 2024.

[4] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "ChatEDA: A Large Language Model Powered Autonomous Agent for EDA," in *IEEE TCAD*, 2024.

[5] M. Liu *et al.*, "ChipNeMo: Domain-Adapted LLMs for Chip Design," in *arXiv preprint*, 2024.

[6] Synopsys, Inc., *Design Compiler: Concurrent Timing, Area, Power, and Test Optimization*. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[7] Cadence, Inc., *Genus Synthesis Solution: Delivering the Best Possible Productivity During RTL Design and the Highest Quality of Results (QoR) in Final Implementation*. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[8] M. M. Ziegler, H.-Y. Liu, G. Gristede, B. Owens, R. Nigaglioni, and L. P. Carloni, "A Synthesis-Parameter Tuning System for Autonomous Design-Space Exploration," in *Proc. DATE*, 2016.

[9] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys® Design Compiler™ and PrimeTime®*. 2002.

[10] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys®*. 1997.

[11] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "RTLRewriter: Methodologies for Large Models aided RTL Code Optimization," in *Proc. ICCAD*, 2024.

[12] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards LLM-Powered Verilog RTL Assistant: Self-Verification and Self-Correction," in *arXiv preprint*, 2024.

[13] F. Cui *et al.*, "OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection," in *arXiv preprint*, 2024.

[14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In *arXiv preprint*, 2024.

[15] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," in *Transactions of the Association for Computational Linguistics*, 2024.

[16] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting," in *Proc. DAC*, 2006.

[17] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Optimizing Majority-Inverter Graphs with Functional Hashing," in *Proc. DATE*, 2016.

[18] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A Novel Basis for Logic Rewriting," in *Proc. ASPDAC*, 2017.

[19] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, Y. Huang, and B. Yu, "FGNN2: A Powerful Pre-Training Framework for Learning the Logic Functionality of Circuits," in *IEEE TCAD*, 2024.

[20] Z. Shi, Z. Zheng, S. Khan, J. Zhong, M. Li, and Q. Xu, "DeepGate3: Towards Scalable Circuit Representation Learning," in *Proc. ICCAD*, 2024.

[21] H. Zheng, Z. He, F. Liu, Z. Pei, and B. Yu, "LSTP: A Logic Synthesis Timing Predictor," in *Proc. ASPDAC*, 2024.

[22] C. Xu, P. Sharma, T. Wang, and L. W. Wills, "Fast, Robust and Transferable Prediction for Hardware Logic Synthesis," in *Proc. MICRO*, 2023.

[23] N. Wu, Y. Li, C. Hao, S. Dai, C. Yu, and Y. Xie, "Gamora: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks," in *Proc. DAC*, 2023.

[24] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph Neural Network Inference for Transferable Power Estimation," in *Proc. DAC*, 2020.

[25] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," in *Proc. FOCS*, 1981.

[26] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Proc. NIPS*, 2020.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, H. Chi, Q. V. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Proc. NIPS*, 2022.

[28] J. J. Miller, "Graph Database Applications and Concepts with Neo4j," in *Proc. SAIS*, 2013.

[29] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *Proc. SIGMOD*, 2018.

[30] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning On Large Graphs," in *Proc. NIPS*, 2017.

[31] K. Sohn, "Improved Deep Metric Learning with Multi-class N-pair Loss Objective," in *Proc. NIPS*, 2016.

[32] X. Wang, X. Han, W. Huang, D. Dong, and M. R. Scott, "Multi-Similarity Loss with General Pair Weighting for Deep Metric Learning," in *Proc. CVPR*, 2019.

[33] S. Robertson, H. Zaragoza, *et al.*, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends® in Information Retrieval*, 2009.

[34] R. Nogueira, Z. Jiang, and J. Lin, "Document Ranking With a Pretrained Sequence-to-Sequence Model," in *arXiv preprint*, 2020.

[35] J. Achiam *et al.*, "GPT-4 Technical Report," *arXiv preprint*, 2023.

[36] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[37] A. Amid *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.

[38] F. Farshchi, Q. Huang, and H. Yun, "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim," in *Proc. EMC2*, 2019.

[39] H. Genc *et al.*, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *Proc. DAC*, 2021.

[40] Intel Labs, *Vector Acceleration IP core for RISC-V**. [Online]. Available: https://github.com/IntelLabs/riscv-vector.

[41] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures," in *Proc. IISWC*, 2014.

[42] A. Dubey *et al.*, "The Llama 3 Herd of Models," in *arXiv preprint*, 2023.

[43] V. Rawte, A. Sheth, and A. Das, "A Survey of Hallucination in Large Foundation Models," in *arXiv preprint*, 2023.

[44] Anthropic PBC, *Introducing Claude 3.5 Sonnet*, 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet.

[45] OpenCores, *OpenCores Hardware RTL Designs*. [Online]. Available: https://opencores.org.

[46] G. Tziantzioulis, T.-J. Chang, J. Balkind, J. Tu, F. Gao, and D. Wentzlaff, "OPDB: A Scalable and Modular Design Benchmark," in *IEEE TCAD*, 2021.

[47] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, *et al.*, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain," *Proc. GOMACTECH*, 2019.

[48] T. Marena, "RISC-V: High Performance Embedded SweRV™ Core Microarchitecture, Performance and Chips Alliance," *Western Digital Corporation*, 2019.

[49] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proc. NIPS*, 2019.

[50] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," in *arXiv preprint*, 2019.

[51] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The Faiss Library," in *arXiv preprint*, 2024.

[52] Synopsys, Inc., *PrimePower: RTL to Signoff Power Analysis*. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html.