From Errors to Solutions: LLM-Powered Command Scripting for FPGA CAD Tools

Mohamed A. Elgammal, and Vaughn Betz
Dept. of Electrical & Computer Engineering, University of Toronto, Toronto, Canada
mohamed.elgammal@mail.utoronto.ca
vaughn@ece.utoronto.ca

***Abstract*—Computer-aided design (CAD) tools provide hundreds or even thousands of options that control various optimizations throughout the design flow. While this flexibility is powerful, it requires significant experience to be familiar with those options and effectively utilize them. For example, when a design fails, in many cases errors can be resolved by adjusting the CAD tool options rather than modifying the design itself.**

In this work, we propose *VPR-LLM*, a tool that utilizes Large Language Models (LLMs) to automate error resolution in the open-source FPGA CAD tool Verilog-to-Routing (VTR) by modifying the command-line options used to run the tool. VPR-LLM parses error logs, VTR help messages, and documentation, then utilizes an LLM to generate modified command-line options that resolve the issue. VPR-LLM supports various LLM models and prompting techniques. All these models and techniques are evaluated and compared in terms of efficiency and cost. To evaluate our method, we proposed a dataset of 26 VTR run failures spanning five distinct error categories. The proposed technique successfully resolved 80% of the cases without requiring any fine-tuning to the LLM model, demonstrating the effectiveness of *VPR-LLM*. This work represents an initial step toward AI-assisted debugging in CAD flows, where LLMs can enhance productivity by automatically identifying and correcting tool configurations.

***Index Terms*—CAD, LLM, FPGA**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are widely utilized in various applications including wireless communications [1], datacenter infrastructure [2], [3], machine learning accelerators [4], [5] and many more. The key factor behind FPGAs' widespread adoption across these diverse applications is their reconfigurability. However, this same flexibility introduces significant challenges in the CAD flow, which is responsible for mapping a designer's high-level description onto the FPGA fabric. This process requires making a series of complex optimization decisions to produce an efficient implementation. Since many of these optimization tasks are NP-hard, FPGA CAD tools rely on heuristic-based approaches with numerous tunable parameters. While tool developers preconfigure these parameters to achieve good performance in general cases, users can manually adjust them through the command line, GUI, or scripting to further optimize results for specific designs.

The vast configurability of FPGA CAD tools, while highly beneficial, poses significant challenges due to the sheer number of available options and parameters. Navigating this complexity can be overwhelming for both beginners and experienced users. For instance, AMD's Vivado Design Suite provides documentation exceeding 500 pages to cover its synthesis and implementation options [6], [7]. Similarly, Intel's Quartus Prime includes extensive documentation detailing various optimization settings in more than a thousand pages [8]. While commercial tools can be difficult to master, the challenge is even greater for open-source alternatives, which often lack dedicated support infrastructure. For example, the Verilog-to-Routing (VTR) tool [9], [10] offers approximately 650 pages of documentation but does not have a customer support team [11].

Examining the VTR documentation reveals that its backend tool, VPR—which handles packing, placement, and routing—alone provides over 200 configurable options. These options govern various aspects of the flow, including which optimizations to apply, tuning parameter values, design constraints, the target FPGA device, and the tool's computational effort. While these settings play a crucial role in determining the quality of results, they can also be instrumental in resolving errors encountered during execution. Many failures arise from incorrect command-line arguments, insufficient tool effort, incompatible option combinations, or an inadequately sized target device. In such cases, adjusting the command-line options can often resolve the issue without requiring modifications to the design itself.

Large language models (LLMs) such as GPT-4 [12], Claude [13], and LLaMA [14], [15] have shown remarkable proficiency in tasks like text comprehension, context understanding, and answering questions, raising the question of whether they can be leveraged to understand the extensive and often complex documentation of FPGA EDA tools like VTR. Given the breadth and complexity of the available options, these models could potentially be used to interpret the documentation and automatically adjust tool settings to resolve errors encountered during execution.

In this paper, we introduce `VPR-LLM`, a framework that leverages a Large Language Model (LLM) to automatically diagnose and resolve errors encountered during VTR execution by modifying the command-line options used to run the tool. To improve efficiency and reduce the cost of LLM inference, we further propose `VPR-LLM-RAG`, an enhanced version that employs Retrieval-Augmented Generation (RAG) to selectively retrieve relevant sections of the documentation based on the encountered error. This targeted retrieval improves

response accuracy while minimizing computational overhead. To assess the effectiveness of our approach, we constructed a dataset of VPR failures spanning various error categories and evaluated multiple LLM models and prompting techniques.

II. Background

A. Verilog-to-Routing

VTR [9] is an open-source FPGA CAD flow that facilitates the exploration of hypothetical FPGA architectures and the development of new CAD algorithms, while also supporting the targeting of commercial FPGAs. It integrates multiple open-source tools, including Yosys [16], and Odin II [17] for logic synthesis, ABC [18] for logic optimization and technology mapping, and VPR [19] for packing, placement, and routing. VTR offers multiple CAD flow variants, allowing users to experiment with various configurations and approaches based on their specific design requirements. For example, the placement stage of VTR supports various techniques ranging from analytical placement [20], simulated annealing (SA) with range limiters [19], to reinforcement learning (RL) controlled simulated annealing [21], [22]. For each of those algorithms, there are multiple options to choose from; for example in the RL-based placement algorithm (RLPlace), the RL agent can control the move type used during SA [22] or control both the move type and the block type to be moved [23]. Furthermore, each of these settings has further tunable options that greatly affect the flow.

This flexibility allows VTR to serve as the foundation for several key frameworks, such as OpenFPGA [24], which automates the design, verification, and layout of customized FPGA architectures, and is now integral to startups like RapidSilicon and RapidFlex [25], [26]. VTR's flexibility allows it to be employed in building bespoke FPGAs implemented using standard cells [27], while also being used to program some mainstream commercial devices like the Xilinx Virtex-6 FPGA family through the VTR-to-Bitstream toolchain [28]. Additionally, VTR is instrumental in the Symbiflow project, which targets Xilinx's Artix-7 FPGAs [29]. Its robustness, adaptability, and comprehensive support for various CAD flows have made it a core component in advancing FPGA design research and applications.

B. Large Language Models

Large Language Models (LLMs) are advanced deep-learning models trained on large quantities of text to understand, generate, and manipulate human language. The versatility of LLMs, along with their ability to process and generate text with contextual understanding, has enabled their adoption in various domains, such as machine translation [30], automated code generation [31], healthcare diagnostics [32], legal document analysis [33], and financial forecasting [34].

The rapid evolution of LLMs has led to the development of increasingly large models with huge numbers of parameters. For example, OpenAI's GPT-4 has approximately 1.8 trillion parameters, significantly surpassing its predecessors in complexity and performance [35]. Similarly, Meta's Llama series has seen substantial growth; the Llama 3.1 model, released in July 2024, has more than 405 billion parameters [36]. Another example is DeepSeek, which has introduced models with large parameter counts and competitive performance, further advancing the field of LLM research [37]. These advancements in the size and capabilities of LLMs have allowed them to be deployed in more and more applications.

Most applications that leverage LLMs to address specific problems employ one of two major techniques:

1) Fine-Tuning the Model on a Specific Dataset: This approach involves adapting a pre-trained LLM to a particular task by further training it on a domain-specific dataset. One technique to do this fine tuning is Low-Rank Adaptation (LoRA) [38], which introduces trainable low-rank matrices into each Transformer layer to significantly reduce the number of trainable parameters while enabling domain-specific tuning. The main advantage of fine-tuning is that it enables the model to specialize in the target task, leading to high accuracy. However, it is computationally expensive and requires a large labeled dataset for use during the fine-tuning process, and such a dataset may not always be available.
2) Prompt Engineering: Instead of modifying the model's parameters, this method carefully designs prompts (i.e., inputs) to guide the model's responses. For example, Chain-of-Thought (CoT) prompting can enhance reasoning capabilities by prompting the model to generate intermediate reasoning steps [39]. The advantage of prompt engineering is that it requires no additional model training and can be rapidly adjusted for different tasks. However, it often requires extensive trial and error to achieve the best results.

C. Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [40] is an advanced framework that enhances the accuracy and reliability of LLMs by integrating external knowledge retrieval into the response generation process. Unlike standard LLMs that rely solely on their pre-trained parameters, RAG dynamically fetches relevant documents from an external database resulting in better accuracy and fewer hallucinations. This approach is particularly beneficial for tasks that require domain-specific knowledge where accurate and up-to-date information is crucial. RAG can be implemented through different retrieval techniques, including sparse retrieval methods like BM25 [41], dense retrieval models such as Dense Passage Retrieval (DPR) [42], and vector embedding models [43] for semantic document matching.

D. LLM for EDA

Recently, LLMs have been increasingly applied to the domain of Electronic Design Automation (EDA) and Computer-Aided Design (CAD) tools. DAVE [44] and VerilogEval [45] investigated the use of LLMs to generate hardware description language (HDL) code. Qiu et al. [46] utilized a LLM to explain synthesis errors to the users. Nvidia proposed ChipeNemo [47] that utilizes a LLM for 3 EDA applications: an engineering

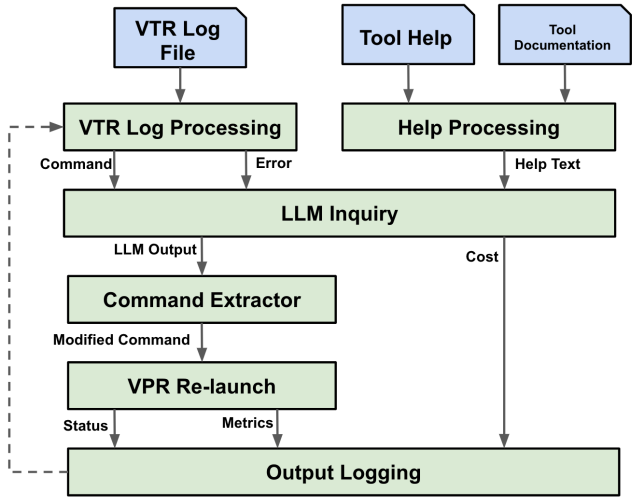


Fig. 1: VPR-LLM flow diagram.

assistant chatbot, EDA script generation, and bug summarization and analysis. Ahmad et al. [48] proposed using a LLM to repair identified security bugs hardware designs by generating replacement code. As LLMs continue to evolve and improve, they present an exciting opportunity to revolutionize the EDA and CAD landscape by improving automation, increasing the precision of feedback, and lowering the barrier to use these tools.

III. VPR-LLM

As discussed earlier, during the execution of the VTR tool suite, errors may arise due to incorrect, incomplete, or incompatible command-line options. Additionally, most optimization algorithms in VTR are heuristic-based and tuned for average-case scenarios. As a result, their default parameter values may not always yield optimal results and, in some cases, can even lead to tool failures. Choosing suboptimal parameter values—whether explicitly set in the command line or inherited as defaults—can cause the tool to fail or produce poor-quality results. These errors require manual intervention to diagnose and correct the issue. The complexity of VPR's command-line interface, combined with a vast number of configurable parameters, makes it challenging for users to quickly identify and resolve such errors. To automate this process and provide efficient debugging assistance, we introduce VPR-LLM, a system that integrates large language models (LLMs) with a specialized workflow designed to process VTR logs, extract relevant information, and generate corrected command-line options.

The proposed approach follows a structured flow, as illustrated in Figure 1. It consists of five key stages: *VTR Log Processing*, *Help Processing*, *LLM Inquiry*, *Command Extraction and Re-execution*, and *Output Logging and Iterative Refinement*. The following paragraphs provide a detailed explanation of each stage.

1) VTR Log Processing

The first stage of the flow involves extracting key information from the VTR execution logs. When an error occurs, the system parses the log file to identify three critical elements:

```
System message: You are an expert
in the VTR tool, specializing
in debugging and command-line
optimizations. You have access to
the following VTR command-line help
text: {help text}

User message: The VTR tool was
run with the following command:
{command} However, the following
error occurred: {error}

Your task is to modify the command
to resolve the error while
preserving the original file
names. Prioritize fixing settings
(e.g., effort levels, optimization
parameters) over increasing device
size (e.g., channel width, grid
size), unless absolutely necessary.
Provide only the modified command
and a brief explanation of the
changes made.
```

Fig. 2: VPR-LLM basic prompt template.

(1) the command used to invoke the VPR tool, (2) the error message produced, and (3) a small window of preceding log lines to provide additional execution context. This information helps the LLM understand not only the error message itself but also any prior warning messages or parameter settings that might have contributed to the issue.

2) Help Processing

Since resolving an error requires an understanding of valid command-line options and parameter constraints, we preprocess the VPR tool's help into a text format. The raw help text can either be directly used as input to the LLM or further segmented into small chunks for efficient retrieval of relevant sections as we will detail in Section IV.

3) LLM Inquiry

Next, the *LLM Inquiry* stage constructs a prompt using the extracted command, error message, and help text following the prompting template shown in Figure 2.The prompt is designed to guide the LLM in diagnosing the issue and suggesting modifications to the command while adhering to specific constraints:

- Prioritize correcting parameter settings (e.g., effort levels, optimization heuristics) over increasing device size (e.g., grid dimensions, channel width), unless absolutely necessary.
- Ensure that input file names and output paths remain unchanged to maintain consistency across runs and avoid introducing new errors.
- Avoid unnecessary modifications to default settings that do not directly relate to the error.

LLM Inquiry stage then initializes an LLM client from a selected provider, sends the prompt, and collects the response along with statistics on the number of input and output tokens. To facilitate broad compatibility and flexibility, the *LLM Inquiry* stage is designed in a modular way, allowing it to support any provider compatible with the OpenAI API [49], such as OpenAI [49], Groq [50], Cerebras [51], and DeepSeek [37]. Users can configure the system to select their preferred LLM provider and specify the desired model variant. The system also logs statistics such as the number of input and output tokens for each query, allowing users to track costs and optimize usage.

4) Command Extraction and Re-execution

After receiving the LLM-generated response, the system extracts the modified command from the model's output. Since LLMs may provide verbose explanations or extraneous text, the system employs regular expressions and structured parsing techniques to isolate the corrected command. The extracted command is then passed to the *VPR Re-launch* stage, where the tool is re-executed with the suggested modifications. The system monitors the execution status, captures performance metrics, and determines whether the issue has been successfully resolved.

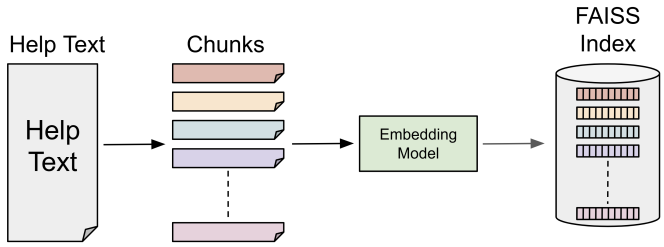
5) Output Logging and Iterative Refinement

All relevant information from *VPR-LLM* consolidated into an output log, including:

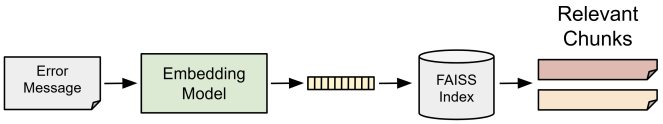
- The original command and its corresponding error message.
- The LLM-suggested modifications and the reasoning behind them.
- The modified command used for re-execution.
- The resulting VPR log output and execution status.
- Performance metrics such as runtime, memory usage, and routing congestion.

If the modified command still results in a failure, the system can optionally trigger an iterative refinement process. In this mode, the newly generated VPR log file is fed back into the system, allowing the LLM to generate further corrections based on the updated error message. This iterative approach continues until a successful execution is achieved or a predefined retry limit is reached.

The proposed flow demonstrates highly promising results, as detailed later in Section VII. However, a key limitation of this approach is the substantial number of input tokens required, primarily due to the extensive size of the tool's help and documentation. For instance, the help text alone exceeds 14,000 tokens (using OpenAI tokenizer), which significantly constrains the inclusion of additional resources, such as the full documentation or relevant research papers, within the prompt to avoid exceeding the model's context length. Moreover, the high token consumption restricts the application of advanced prompting techniques. Additionally, the high token consumption increases computational costs, particularly when using commercial LLM services. Every query incurs expenses based on input and output token counts, which can become a limiting factor for large-scale or frequent use. To address this issue, we introduce an optimized approach, `RAG-VPR-LLM` to selectively retrieve and include only the most relevant help text in each query.



(a) Indexing the tool help



(b) Retrieving relevant help

Fig. 3: `RAG-VPR-LLM` help indexing and retrieval flow

IV. RAG-VPR-LLM

To optimize the number of input tokens while preserving the effectiveness of LLM-based error resolution, we employ a *Retrieval-Augmented Generation (RAG)* approach. Instead of passing the entire help documentation as part of the LLM prompt, we dynamically extract only the most relevant sections based on the specific error message and command in question. This significantly reduces token usage while improving response quality by ensuring the model focuses only on relevant information.

Our RAG-based approach leverages semantic search techniques to retrieve the most relevant help documentation before invoking the LLM. The workflow consists of three main stages: indexing the help documentation, processing queries, and retrieving relevant parts of the help documentation.

1) Indexing the Help Documentation

As illustrated in Figure 3a, we preprocess the help text by splitting it into smaller, manageable chunks based on paragraph boundaries. This ensures that each chunk contains a coherent and self-contained piece of information. Each chunk is then transformed into a dense vector representation using a pre-trained embedding model, allowing for efficient semantic similarity comparisons. The generated embeddings, along with their corresponding text chunks, are stored in a Facebook AI Similarity Search (FAISS) [52] index, enabling fast and scalable nearest-neighbor searches.

2) Query Processing and Retrieval

When an error occurs, we construct a query by combining the error message and the original command responsible for triggering the error. The query is then encoded into an embedding using the same model used for indexing the help documentation, as shown in Figure 3b. The FAISS index is then queried to retrieve the top-k most relevant help text chunks that closely match the query's embedding. These retrieved chunks are expected to contain useful information for resolving the issue.

3) Context Construction and LLM Invocation

Rather than providing the entire help documentation to the LLM, we concatenate only the retrieved chunks and include them in the LLM prompt along with the error message and command following the structure shown in Figure 2. This approach ensures that the LLM receives the most relevant information without being overloaded with extraneous data, improving both accuracy and efficiency.

By integrating this RAG-based retrieval mechanism, our approach (`RAG-VPR-LLM`) offers several advantages in terms of efficiency, accuracy, and scalability.

First, the use of RAG significantly reduces token usage and cost. Instead of consuming thousands of unnecessary tokens by passing the entire help text, our approach ensures that only the most relevant sections are included in the LLM query. This leads to a substantial reduction in token consumption, making the solution more cost-effective and even compatible with free-tier LLM APIs in certain configurations.

Second, our method improves response quality and accuracy by providing the LLM with focused, high-quality context. Instead of overwhelming the model with excessive information, the retrieval mechanism selects only the most relevant documentation, resulting in more precise and relevant responses. This structured retrieval process also helps in reducing hallucinations, as the model is guided by only relevant information.

Another advantage of our approach is its scalability. Unlike naive solution (i.e. *VPR-LLM*) that directly feed entire help files into the LLM, RAG enables seamless integration of additional knowledge sources, including tool documentation, research papers, source code comments, and past error logs. The retrieval mechanism ensures that only the most useful information is selected, preventing performance degradation due to excessive input size.

Furthermore, the framework is designed to be modular, allowing for easy customization and experimentation. The embedding model can be replaced with a different one that offers better performance, the number of retrieved chunks can be tuned to optimize accuracy versus cost, and additional filtering mechanisms can be incorporated to refine retrieval results further.

V. Advanced Prompting techniques

The efficient retrieval technique proposed in *RAG-VPR-LLM* enables us to explore more advanced prompting techniques to enhance the accuracy and robustness of the proposed flow. These techniques enhance the LLM's ability to generate precise and well-reasoned corrections while maintaining computational efficiency. Below, we describe the key prompting techniques we tested.

A. Chain-of-Thought (CoT) Reasoning

A fundamental limitation of single-shot responses from LLMs is that they may not fully consider alternative solutions or evaluate trade-offs effectively. To address this, we implement a *Chain-of-Thought (CoT)* reasoning approach, where the error resolution process is decomposed into multiple sequential steps. First, the LLM is asked to generate a comprehensive list of potential solutions to the given error. Instead of providing a single correction outright, the system offers several alternatives. Following this, a follow-up query asks the model to rank the proposed solutions based on their effectiveness in resolving the issue. Factors such as resource efficiency and minimal modification to key design parameters are considered during ranking. Finally, the LLM selects the highest-ranked solution and verifies its correctness by cross-referencing the modified command against the extracted help documentation before execution.

We expected this structured reasoning process mitigates incorrect or suboptimal corrections, ensuring that the final recommendation is both well-justified and more likely to resolve the error effectively.

B. Evaluate N Temps: Controlling Response Variability

The temperature parameter used in LLMs controls the randomness of the responses generated. Lower temperature values (e.g., 0.2) lead to more deterministic, focused outputs, while higher values (e.g., 0.8) introduce greater variability in the responses. To harness this property, we introduce the *Evaluate N Temps* technique. In this approach, the same *Basic* prompt (similar to the one in Figure 2 and using *RAG-VPR-LLM*) is issued multiple times with varying temperature settings, producing a range of responses. These responses are then sent to the LLM again to be evaluated, and the LLM selects the most coherent and contextually appropriate solution. This method helps to balance response diversity and reliability, ensuring that edge-case errors are addressed without introducing unnecessary randomness.

C. Evaluate N Seeds: Enhancing Solution Diversity

Another source of variability in LLM-generated outputs comes from the random seed used during inference. To enhance the diversity of potential solutions, the *Evaluate N Seeds* technique executes the same query multiple times with different random seeds. This produces a variety of potential corrections, and similar to *Evaluate N Temps* technique, all these proposed corrections are sent back to LLM for evaluation and picking the optimal out of them. This approach is particularly valuable for ambiguous errors where multiple viable solutions might exist, ensuring that the final recommendation is not biased by any one response.

By incorporating these advanced prompting strategies, our flow enhances the accuracy of error resolution while maintaining computational efficiency.

VI. Failures dataset

To effectively evaluate the proposed flow, we introduce a curated dataset consisting of 26 distinct failure cases encountered during the execution of VTR. This dataset encompasses a wide variety of error scenarios, each representing a unique case that highlights a specific failure within the tool. It is important to note that modifying only the circuit or architecture file, while leaving the underlying error cause unchanged, does not create a new failure case. Each failure in our dataset

is fundamentally distinct in its root cause and manifestation, ensuring a comprehensive evaluation framework.

The failure cases in this dataset are organized into five broad categories based on their underlying cause, allowing for targeted analysis and testing of our error-handling and correction flow. These categories are as follows:

1) **Typos and Command Errors**: Failures in this category stem from incorrect syntax, misspelled commands, or missing arguments in the command-line invocation. These issues are typically caused by human errors in the formulation of command-line instructions.
   - **Example**: A user might mistype an option name, such as using `--rote_chan_width` instead of `--route_chan_width`. In this case, VPR would reject the unrecognized flag, leading to a failure. Another common mistake is omitting a required option value, such as using `--disp` instead of `--disp on`, which would result in an invalid argument error. These types of failures are straightforward to identify but often lead to frustrating debugging experiences, especially for new users.
2) **Bad Values**: This category includes errors that occur when invalid, unsupported, or out-of-range values are passed to VPR configuration options, causing the tool to behave incorrectly or even fail entirely.
   - **Example**: When using a unidirectional routing architecture, specifying an odd routing channel width (e.g., `--route_chan_width 15`) can cause VPR to fail. Similarly, another example involves using an incompatible circuit format with the `--circuit_format` flag. While the format specified might technically be valid, it may not match the actual format of the circuit file, leading to parsing errors when the architecture file is processed. These types of errors are often due to mismatches between configuration parameters and the actual design specifications.
3) **Not Enough Effort**: Failures in this category are a result of the default tool settings not providing enough computational effort to successfully complete the placement and routing tasks. In these cases, the tool may require additional resources or iterations to find a feasible solution.
   - **Example**: A common issue is that routing might fail because the default number of routing iterations is insufficient. For instance, increasing the iteration limit using the `--max_router_iterations` flag can allow VPR to make more attempts to find a valid routing solution, resolving the failure. These types of errors are often indicative of scenarios where the complexity of the design or routing constraints exceeds the tool's initial configuration settings.
4) **Small Size**: Failures in this category occur when the FPGA fabric is too small to accommodate the design's requirements, either in terms of routing resources or available logic elements. These errors are often caused by resource constraints that prevent the tool from placing and routing the design successfully.
   - **Example**: VPR provides two modes for routing: (1) searching for the minimum channel width that allows the circuit to be routed, and (2) targeting a fixed channel width. In the fixed-width mode, routing may fail if the specified routing channel width is too small to support the circuit. This can be resolved by increasing the channel width using `--route_chan_width`, which provides more routing resources. Alternatively, enabling the search for the minimum channel width may also resolve the failure by allowing VPR to automatically adjust the width to fit the design's routing requirements.
5) **File Handling**: Failures in this category are caused by missing, misformatted, or incorrectly referenced files, which can prevent proper execution of the tool. Such errors are often related to external dependencies, such as constraint files or architectural specifications, that are not correctly specified or are unavailable.
   - **Example**: A common failure occurs when an incorrect SDC (Synopsys Design Constraints) file references a non-existent clock signal. In this case, VPR cannot apply the timing constraints correctly, resulting in an error. This issue can be resolved by either removing the incorrect SDC file from the command or modifying it to reference the correct clock signal name. These types of errors highlight the importance of correct file referencing and ensuring that all necessary files are properly formatted and available.

This dataset provides a structured benchmark for evaluating the efficiency of the proposed flow. By covering a diverse set of failure types, it allows us to rigorously assess how well different error-handling mechanisms and retrieval strategies perform in diagnosing and resolving VPR failures, ensuring that the proposed flow can handle the complexities of different failure modes effectively. This dataset is available at *removed for blind review*.

VII. Results & Discussion

To evaluate the effectiveness of the proposed `VPR-LLM` flow, we conducted a series of experiments using the failure dataset introduced in Section VI. For each case, we assessed how well the proposed flow was able to generate the correct command modifications to resolve the issue. The process involved extracting the modified command generated by the LLM for each failure case, which was then automatically used to launch VPR. Following this, we monitored the execution status of VPR to determine if the failure was successfully resolved or if further adjustments were required.

All LLM queries for this evaluation were performed using the Groq [50] API, which provides a robust interface for interacting with large language models. Unless otherwise

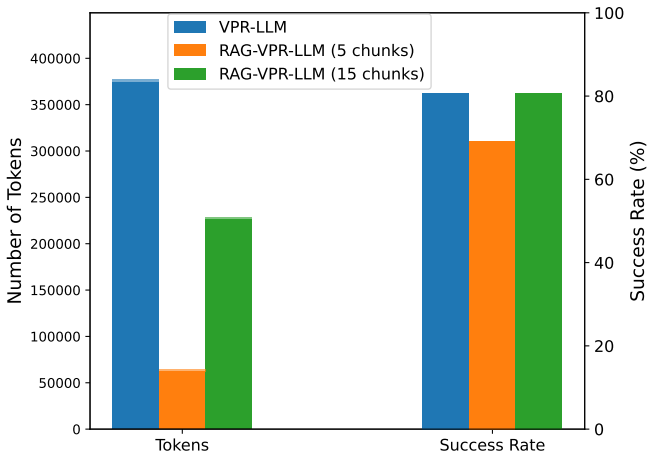


Fig. 4: `VPR-LLM` and `RAG-VPR-LLM` performance and cost comparison. For the tokens bars, the light colors represent the output tokens while the dark colors represent the input tokens.

specified, the `Llama3.3-70B` model was used for generating the command modifications. This model, referred to as `llama-3.3-70b-versatile` in the Groq API, was chosen for its strong performance across a range of tasks and its capacity to handle the complexity of the VPR-related queries. For generating the responses, we configured the LLM with a temperature setting of 0.5 and a seed value of 1, unless otherwise noted.

For the Retrieval-Augmented Generation (RAG)-based experiments, we utilized the `gte-large` embedding model from TheNLPer [53]. This model provides 1024-dimensional embeddings, which are essential for efficiently retrieving and integrating relevant information from large text corpora, such as VPR's help and error logs. By combining these embeddings with the LLM's generative capabilities, we were able to retrieve precise information and generate contextually relevant command modifications to resolve the failures.

A. *VPR-LLM Results*

As shown in Figure 4, `VPR-LLM` successfully resolved more than 80% of the test cases (21 out of 26), allowing VPR to run without errors. This demonstrates the effectiveness of leveraging LLM-based techniques to assist users in overcoming VPR-related issues. However, a subset of failure cases remained unresolved, which we analyzed in detail below:

1) **Incompatible FPGA Architecture:** One failure occurred when attempting to run VPR on a synthesized circuit targeting an FPGA architecture different from the one it was synthesized for. The netlist contained primitives absent in the target FPGA, making resolution infeasible within the scope of this work. While no command modification could resolve this issue, including it in the dataset highlights an opportunity for future enhancements where the LLM agent could access the architecture file or the entire VTR GitHub repository to propose an appropriate architecture.
2) **Incorrect Circuit Format:** In another case, a circuit was provided in `.blif` format while the command specified `--circuit_format FPGA_interchange`. We expected the LLM to resolve this error; however, upon investigation, we found that VPR did not produce a meaningful error message but instead crashed with ambiguous output. This lack of clear error reporting hindered the LLM's ability to diagnose and correct the issue. This case underscores the importance of robust error handling in EDA tools to facilitate LLM-assisted debugging.
3) **Routing Congestion with Fixed Channel Width:** A test case involved routing failures due to excessive congestion with a fixed channel width. The LLM suggested increasing the routing channel width, which was a step in the right direction. However, because we explicitly instructed the agent not to aggressively increase device resources, the adjustment was insufficient to resolve the failure. When we relaxed this constraint, the LLM successfully generated a working solution. Furthermore, allowing iterative interactions between the LLM and VPR enabled the agent to refine its suggestions over multiple attempts, ultimately resolving the issue.
4) **DSP Chain Packing Issue:** Another complex failure arose when placement failed due to insufficient space for DSP blocks, despite the presence of available DSP locations. The root cause was that the circuit created DSP chains, leading to inefficient resource utilization (e.g., a device with two DSP columns of five blocks each versus a circuit with three DSP chains of three blocks each). Since VPR only reports the total number of DSP blocks and available locations without details on chaining, the LLM did not consider increasing the device size as a potential fix. Instead, it attempted adjustments to packing options, which did not resolve the problem. However, removing the restriction against increasing device size enabled the LLM to generate a viable solution.
5) **Invalid Floorplan Constraints:** One failure stemmed from an incorrect floorplanning constraint that mapped a block outside the device's valid region. Resolving this issue would require modifying the constraint file, a task beyond the scope of our current approach. Addressing such cases may necessitate integrating an additional agent capable of analyzing input files and suggesting corrections, which is a potential avenue for future work.

These findings underscore both the strengths and limitations of `VPR-LLM`. On the one hand, the system showcases impressive performance in automatically identifying and correcting a majority of the VPR-related failures, demonstrating its potential as a valuable tool for VPR users. However, certain failure cases reveal that the current capabilities of `VPR-LLM` are not sufficient to fully resolve all errors. These unresolved issues often point to areas where the system's understanding of the underlying architecture or the specific context of the failure could be further refined. Future work will explore these directions to further enhance automated debugging for VPR users.

B. RAG-VPR-LLM Results

A notable limitation of VPR-LLM is its dependence on the complete help documentation of the tool for each query, resulting in substantial token consumption. As shown in Figure 4, processing all 26 failure cases required ∼375k input tokens, averaging approximately 14.5k tokens per individual inquiry. This heavy reliance on the full documentation leads to significant computational costs, making the approach less scalable for larger datasets or real-time applications.

RAG-VPR-LLM addresses this issue by retrieving only the top 5 most relevant data chunks, reducing total input tokens over all the dataset to approximately 62k (<2.5k tokens per inquiry). However, this optimization reduces the error resolution success rate to ∼70%. In response to this trade-off, we increased the number of retrieved data chunks to 15, effectively restoring the error resolution success rate to 80%, which matches the performance of VPR-LLM. At the same time, we maintained a total token count below 230k (averaging fewer than 8.7k tokens per test case), which represents a 40% reduction in token usage compared to VPR-LLM. This significant reduction in token consumption not only enhances efficiency but also reduces computational costs, enabling RAG-VPR-LLM to operate within the free-tier access limits of Groq's platform for the Llama3.3-70B model[1]. In terms of runtime, indexing the full tool help text takes approximately 5–6 seconds and is performed only once during the initial local execution; subsequent queries reuse the cached index. For LLM inference, using LLaMA 3.3-70B, the response time is around 7–8 seconds per query using Groq API.

These findings demonstrate that RAG-VPR-LLM effectively balances efficiency and accuracy, providing a scalable and cost-effective solution for automating debugging in VPR. By reducing token consumption while maintaining a high success rate, RAG-VPR-LLM offers a compelling alternative to VPR-LLM, making it more practical for frequent usage.

C. Advanced Prompting Techniques Results

We also investigated the impact of advanced prompting techniques, specifically *Chain of Thought (CoT)*, *Evaluate N Seeds*, and *Evaluate N Temps*, on error resolution performance. These techniques were integrated into the RAG-VPR-LLM flow, where the top 5 most relevant help chunks were retrieved for each query. The success rates of these techniques, along with the associated input and output token counts, are presented in Table I.

The results indicate that these advanced prompting techniques improve the error resolution success rates compared to the baseline RAG-VPR-LLM with the top 5 retrieved chunks—represented by the orange column in Figure 4. However, despite the improvements, none of these techniques were able to match the performance of RAG-VPR-LLM with 15 retrieved chunks—shown in the green column in Figure 4. This suggests that while the advanced prompting methods provide some accuracy benefits, they do not fully compensate for the information loss caused by the reduced number of retrieved chunks. Moreover, these techniques tended to use a similar or slightly higher token count, which indicates that while they can enhance performance, they also increase the computational cost.

[1]Groq's free-tier access and pricing details can be found at https://console.groq.com/docs/rate-limits, accessed on 10 March 2025.

TABLE I: Performance comparison of advanced prompting techniques. All the experiments use RAG-VPR-LLM with top 5 chunks of the help retrieved in each inquiry.

	Success Rate	**# Input Tokens**	**# Output Tokens**
Baseline	69.23%	62261	2886
CoT	73.08%	243290	27901
Eval N seeds	76.92%	255535	11517
Eval N Temps	69.23%	385990	17606

These findings highlight the potential of advanced prompting in improving the accuracy of RAG-VPR-LLM, but they also underscore the importance of carefully balancing accuracy improvements with the associated token costs. To maximize the performance-to-cost ratio, it is essential to apply these advanced prompting techniques strategically, considering both their impact on success rates and their effect on computational resources.

VIII. Conclusion and Future Work

This paper explores the potential of using LLMs to automatically resolve errors in complex CAD tools like VTR by modifying command-line arguments. It also utilized RAG techniques to minimize the cost of LLM deployment while enhancing its ability to access relevant information. Our work introduces a dataset of 26 diverse VTR failure cases, which serves as a benchmark for evaluating the proposed techniques.

The results demonstrate that VPR-LLM effectively resolves over 80% of the test cases, and RAG-based retrieval strategies significantly reduce token consumption without sacrificing accuracy. This enables more scalable and cost-effective deployment, even within free-tier API constraints.

In our future work we aim to expand the information available to the LLM by integrating additional sources such as research papers, source code, and full tool documentation. Furthermore, we propose a multi-agent system where distinct agents handle different aspects of error resolution: one for modifying command-line arguments (as presented in this work), another for adjusting input files when necessary, and a third for updating the tool's source code. Employing Mixture of Experts (MoE) techniques to orchestrate these agents could further improve automation and enhance the robustness of error resolution in VTR and similar CAD tools. Finally, we plan to broaden the class of errors that VPR-LLM can help resolve to include timing closure issues.

Acknowledgments

The authors would like to thank Amin Mohaghegh, Mohamed Eldafrawy, and Ayman Mohamed for their valuable insights and fruitful discussions that contributed to the development of this work. We thank NSERC for funding support.

REFERENCES

[1] J. Lorandel *et al.*, "Fast power and performance evaluation of FPGA-based wireless communication systems," *IEEE Access*, vol. 4, pp. 2005–2018, 2016.

[2] D. Chiou, "The microsoft catapult project," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2017, pp. 124–124.

[3] E. Chung *et al.*, "Serving DNNs in real time at datacenter scale with project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[4] M. Hall and V. Betz, "HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs," *arXiv preprint arXiv:2007.10451*, 2020.

[5] E. Nurvitadhi *et al.*, "Why compete when you can work together: FPGA-ASIC integration for persistent RNNs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 199–207.

[6] Xilinx Inc., *Vivado Design Suite User Guide: Synthesis (UG901)*, accessed: Feb. 16, 2025. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug901-vivado-synthesis.pdf

[7] ——, *Vivado Design Suite User Guide: Implementation (UG904)*, accessed: Feb. 16, 2025. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug904-vivado-implementation.pdf

[8] Intel Corporation, *Quartus Prime Standard Edition Handbook: Volume 1 - Design and Synthesis*, 2017, accessed: Feb. 16, 2025. [Online]. Available: https://faculty-web.msoe.edu/johnsontimoj/Common/FILES/qts-qps-5v1_design-synthesis_17.1.pdf

[9] K. E. Murray *et al.*, "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[10] M. A. Elgammal *et al.*, "VTR 9: Open-source CAD for fabric and beyond FPGA architecture exploration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, May 2025. [Online]. Available: https://doi.org/10.1145/3734798

[11] Verilog-to-Routing Developers, *Verilog-to-Routing Documentation*, 2025, accessed: Feb. 16, 2025. [Online]. Available: https://docs.verilogtorouting.org/

[12] OpenAI, "GPT-4 technical report," 2023, accessed: 2025-02-20. [Online]. Available: https://cdn.openai.com/papers/gpt-4.pdf

[13] Anthropic, *The Claude 3 Model Family: Opus, Sonnet, Haiku*, 2024, accessed: 2025-02-20. [Online]. Available: https://www-cdn.anthropic.com/

[14] Meta, *LLaMA 3.1: Large Language Model with Advanced Capabilities*, 2024, accessed: 2025-02-20. [Online]. Available: https://ai.facebook.com/llama-3-1-model-card

[15] ——, *LLaMA 3.3: Improved Large Language Model with Robust Features*, 2024, accessed: 2025-02-20. [Online]. Available: https://ai.facebook.com/llama-3-3-model-card

[16] C. Wolf, "Yosys open synthesis suite," *Search in*, 2021.

[17] P. Jamieson *et al.*, "Odin II - an open-source verilog HDL synthesis tool for CAD research," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010, pp. 149–156.

[18] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference*. Springer, 2010, pp. 24–40.

[19] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, 1997, pp. 213–222.

[20] M.-C. Kim *et al.*, "SimPL: An effective placement algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 50–60, 2011.

[21] M. A. Elgammal *et al.*, "Learn to place: FPGA placement using reinforcement learning and directed moves," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 85–93.

[22] ——, "RLPlace: Using reinforcement learning and smart perturbations to optimize FPGA placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2532–2545, 2021.

[23] F. Mahmoudi *et al.*, "Respect the difference: Reinforcement learning for heterogeneous FPGA placement," in *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023, pp. 152–160.

[24] X. Tang *et al.*, "OpenFPGA: Towards automated prototyping for versatile FPGAs."

[25] "eFPGA IP - RapidSilicon — rapidsilicon.com," https://rapidsilicon.com/efpga-ip/, [Accessed 18-Jul-2023].

[26] "Rapid Flex — Build the Future with Rapid Flex — rapid-flex.com," https://rapid-flex.com/, [Accessed 18-Jul-2023].

[27] P. Mohan *et al.*, "Top-down physical design of soft embedded FPGA fabrics," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 1–10.

[28] E. Hung *et al.*, "Escaping the academic sandbox: Realizing VPR circuits on Xilinx devices," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013, pp. 45–52.

[29] K. E. Murray *et al.*, "Symbiflow and VPR: An open-source design flow for commercial and novel FPGAs," *IEEE Micro*, vol. 40, no. 4, pp. 49–57, 2020.

[30] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," in *arXiv preprint arXiv:1609.08144*, 2016.

[31] M. Chen *et al.*, *arXiv preprint arXiv:2107.03374*, 2021.

[32] K. Singhal *et al.*, "Large language models encode clinical knowledge," *arXiv preprint arXiv:2301.08091*, 2023.

[33] I. Chalkidis *et al.*, "Legal-BERT: The muppets straight out of law school," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[34] S. Wu *et al.*, "BloombergGPT: A large language model for finance," *arXiv preprint arXiv:2303.17564*, 2023.

[35] E. Topics, "GPT model sizes: A complete breakdown," 2024, accessed: 2025-02-23. [Online]. Available: https://explodingtopics.com/blog/gpt-parameters

[36] M. AI, "Meta Llama 3: Advancing open foundation models," 2024, accessed: 2025-02-23. [Online]. Available: https://ai.meta.com/blog/meta-llama-3/

[37] D. AI, "DeepSeek models and their capabilities," 2024, accessed: 2025-02-23. [Online]. Available: https://deepseek.com/models

[38] E. J. Hu *et al.*, "LoRA: Low-rank adaptation of large language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[39] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[40] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[41] S. Robertson and H. Zaragoza, *Probabilistic Information Retrieval and Statistical Language Models*. Springer, 2009.

[42] V. Karpukhin *et al.*, "Dense passage retrieval for open-domain question answering," *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 701–710, 2020.

[43] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pp. 3980–3990, 2019.

[44] H. Pearce *et al.*, "Dave: Deriving automatically verilog from english," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 27–32.

[45] M. Liu *et al.*, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[46] S. Qiu *et al.*, "LLM-aided explanations of EDA synthesis errors," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–6.

[47] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[48] B. Ahmad *et al.*, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, 2024.

[49] OpenAI, "OpenAI API," 2024. [Online]. Available: https://platform.openai.com/docs/api-reference/

[50] G. Inc., "Groq AI and LPU technology," 2024. [Online]. Available: https://groq.com/

[51] C. Systems, “Cerebras wafer-scale AI systems,” 2024. [Online]. Available: https://www.cerebras.net/

[52] J. Johnson, M. Douze, and H. Jégou, “FAISS: A library for efficient similarity search and clustering of dense vectors,” *arXiv preprint arXiv:1702.08734*, 2017. [Online]. Available: https://arxiv.org/abs/1702.08734

[53] TheNLPer, “GTE: General text embeddings for retrieval and beyond,” *Hugging Face Model Hub*, 2023. [Online]. Available: https://huggingface.co/thenlper/gte-large