

JARVIS: A Multi-Agent Code Assistant for High-Quality EDA Script Generation

Ghasem Pasandi, Kishor Kunal, Varun Tej, Kunjal Shah, Hanfei Sun, Sumit Jain, Chunhui Li, Chenhui Deng, Teodor-Dumitru Ene, Haoxing Ren, and Sreedhar Pratty

Nvidia Corp.

Santa Clara, CA, USA

{gpasandi, kkunal, vtej, kunjals, hanfeis, sumjain, chunhuil, cdeng, tene, haoxingr, spratty}@nvidia.com

Abstract—This paper presents JARVIS, a novel multi-agent framework that leverages Large Language Models (LLMs) and domain expertise to generate high-quality scripts for specialized Electronic Design Automation (EDA) tasks. By combining a domain-specific LLM trained with synthetically generated data, a custom compiler for structural verification, rule enforcement, code fixing capabilities, and advanced retrieval mechanisms, our approach achieves significant improvements over state-of-the-art domain-specific models. Our framework addresses the challenges of data scarcity and hallucination errors in LLMs, demonstrating the potential of LLMs in specialized engineering domains. We evaluate our framework on multiple benchmarks and show that it outperforms existing models in terms of accuracy and reliability. Our work sets a new precedent for the application of LLMs in EDA and paves the way for future innovations in this field.

I. INTRODUCTION

Large Language Models (LLMs) have revolutionized software development by automating various coding tasks, streamlining repetitive processes, and enhancing developer productivity. Tools like Microsoft’s Copilot and Meta’s CodeLlama have demonstrated the potential of LLMs in generating boilerplate code, automating common patterns, and embedding best practices within generated outputs [3], [17], [21]. However, when applied to specialized fields like Very-Large-Scale Integration (VLSI) design within Electronic Design Automation (EDA), LLM performance is hindered by the scarcity of relevant training data, leading to unreliable and inaccurate outputs. These models often misinterpret and hallucinate due to a lack of contextual depth, highlighting the need for domain-specific fine-tuning.

Recent efforts have focused on enhancing the reasoning capabilities of LLM models using CoT [23] and agent-based frameworks [24] for general tasks. In the context of EDA, TOOLLLM [21] uses a fine-tuned LLaMa model and integrates a depth-first search-based decision tree (DFSdT) to bolster planning and reasoning abilities. Additionally, Retrieval-Augmented Generation (RAG) techniques have been explored in recent works [9]–[11], [13], which demonstrated the utility of retrieval-augmented control code generation in improving LLM performance for niche software engineering tasks. However, these approaches have limitations when applied to highly specialized domains like EDA, where complex dependencies and domain-specific knowledge are crucial.

To address the unique challenges of EDA script generation, we propose a novel framework that focuses on four key areas:

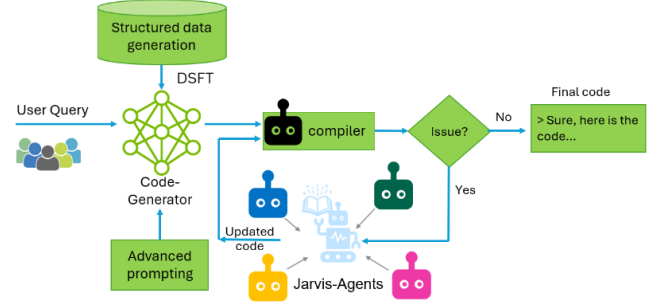


Fig. 1: Overview of the JARVIS framework, illustrating different components and the feedback loop, enabling iterative code refinement and improvement.

Domain Adapted Pre-Training (DAPT) and Domain Supervised Fine-Tuning (DSFT) for model training, multi-LLM collaboration, custom compiler development, and multi-agent feedback flow. Our approach leverages a custom compiler that converts the custom VLSI tool manual to a knowledge graph using Abstract Syntax Tree (AST) format and provides augmented feedback data to the coding agent. This is complemented by a Synthetic Data Generation (SDG) approach to further improve access to domain-specific knowledge. SDG has been shown to significantly boost LLM performance in various applications [8]. Our framework also utilizes RAG models, which effectively integrate domain-specific knowledge into the generation process by retrieving relevant contextual data from external sources.

We developed a multi-LLM framework that synergistically leverages the strengths of diverse LLMs to generate high-quality EDA scripts. Our approach includes a custom compiler that detects and diagnoses issues in EDA code, providing actionable feedback for correction and streamlining the debugging process, with a focus on mitigating hallucination errors. We also introduce a multi-episode, multi-agent framework using ReAct, which optimizes initial EDA scripts using multiple custom-developed tools and a feedback loop, demonstrating the potential of reactive and multi-agent approaches in EDA code improvement.

Our framework, called JARVIS (Just A Remarkable VLSI Intelligence System), demonstrates substantial improvements over existing models for EDA script generation. The proposed framework has broader implications, setting a precedent for the

future application of LLMs in other specialized engineering domains. We evaluated the capabilities of various LLMs, including ChatGPT with GPT4o, and LLaMa 3.1.

Fig. 1 illustrates the components of our framework. The framework consists of a code generator which uses an LLM model trained on domain specific data to generate initial codes for user queries, then through collaboration of multiple jarvis-agents, and custom tools like code compiler, the code is being refined by mitigating hallucination errors and producing high-quality EDA scripts.

The main contributions of this work are as follows:

- A novel Supervised Fine-Tuning (SFT) pipeline for LLMs, enhancing their performance in data-scarce environments and unlocking their potential for EDA tasks.
- A multi-LLM framework that collaboratively leverages diverse LLMs to generate high-quality EDA scripts.
- A custom code compiler that detects and diagnoses issues in an EDA code, providing actionable feedback for correction and streamlining the debugging process.
- A multi-agent multi-episode framework using ReAct, which optimizes initial EDA scripts using multiple custom-developed tools and a feedback loop.
- Comprehensive experimental results on multiple LLMs, validating the effectiveness of our approach on real-world VLSI/EDA tasks.

II. RELATED WORK

The capabilities of LLMs in coding related tasks such as test-driven code generation [4], code refactoring [9], [16], and generating synthetic data for training purposes [16] have been well-documented. To enhance the quality of LLM-generated code, research is being conducted in several directions. These include model improvement using various training techniques and Self-Distillation with Guidance (SDG), leveraging retrieval to enhance domain knowledge, improving reasoning abilities through prompt-tuning, and implementing corrective-feedback flows using tools or multi-LLM-based evaluation.

New LLM models targeting coding applications such as CodeLlama [1], StarCoder [20], StableCode [2] are being trained to improve LLMs code generation capabilities. SDG approaches [7], [8], [15], [16] have become popular in enhancing LLM’s capability of integrating user-provided or retrieved context for conversational tasks. RAG techniques have been explored in recent works [9]–[11], [13], which demonstrated the utility of retrieval-augmented control code generation in improving LLM performance for niche software engineering tasks. To further reduce hallucinations in code generation, approaches like AST structure-aware pretraining [6] boost the performance of coding tasks.

Recent efforts have focused on enhancing the reasoning capabilities of LLM models using Chain-of-Thought (CoT) [23], ReAct framework [4], [12] and agent-based frameworks [24] for general tasks. To improve reasoning, LLM agents can leverage external models, tools, plugins, or APIs to tackle complex problems [5]. The integration of feedback loops and agent-based frameworks has been pivotal in improving LLM performance for real-world tasks, especially in niche

domains. Works like CodeAgent [18], [25] demonstrate the effectiveness of feedback-driven mechanisms in refining LLM outputs through execution-based verification and active user interaction.

In the EDA domain, creating scripts for custom VLSI tools presents unique challenges. The complex, domain-specific requirements of EDA, combined with a lack of representative training data, limit the effectiveness of general-purpose LLMs like Copilot and CodeLlama. Recent advancements in domain-adapted models, such as ChipNeMo [14], [21], illustrate the potential of LLMs tailored to chip design tasks. ChipNeMo employs several domain adaptation techniques, including domain-adaptive tokenization, domain-adaptive pre-training (DAPT), and model alignment, to customize LLMs for chip design. TOOLLLM [21] uses a fine-tuned LLaMa model and integrates a depth-first search-based decision tree to enhance planning and reasoning abilities.

However, ChipNeMo’s performance can be further improved through additional specialized fine-tuning (SFT) using data generated by methods like SDG. TOOLLLM’s performance can also be enhanced using an agentic framework, which provides better reasoning capabilities. These innovations underscore the importance of domain-specific tasks, such as EDA, where data scarcity remains a persistent issue. Despite these efforts, significant gaps remain in applying these models to highly specialized domains like EDA. In this paper, we address some of these challenges by exploring advanced domain adaptation techniques and proposing new methodologies to enhance the performance of LLMs in the EDA domain.

III. SYNTHETIC DATA GENERATION (SDG)

DAPT and DSFT are essential components of model training. DAPT uses raw data during training and DSFT uses a custom-labeled data for fine-tuning a model. The SFT data should encompass deep knowledge of a specific domain, enabling the LLM to grasp the nuances necessary for handling specialized tasks. This requires high quality labeled data. We experimented with a small set of manually generated high quality data but we observed signs of overfitting in the trained model due to the limited domain SFT data.

Specialized SDG approaches can enhance the effectiveness of SDG by controlling the data chunking process or making small perturbations in existing codes [16]. However, due to the limited reasoning capabilities of LLMs, it is often unrealistic to expect them to generate an entire code dataset, especially when the data involves complex structures or semantics. Our observations indicate that while LLMs struggle with code generation tasks for custom tools, they excel at generating comments and summarizing tasks. Therefore, we leveraged LLMs to generate comments and questions from raw codes, thereby increasing our SFT data examples.

To overcome the data limitation and increase the tool command coverage, we developed a random code generator that utilizes the tool’s command graph as shown in an example in Fig. 2 to create syntactically correct codes using an AST. An AST is a tree representation of the abstract syntactic structure of source code written in a programming language.

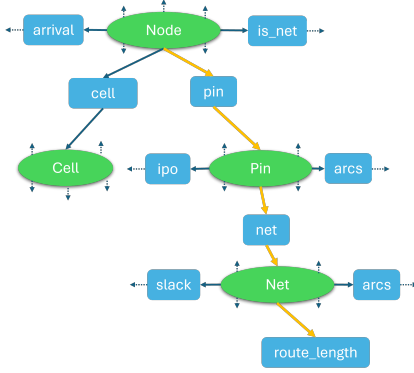


Fig. 2: Example for a tool API graph constructed from man page.

Each node of the tree denotes a construct occurring in the source code. Thus, the high level idea of using an AST is to build random code stage by stage by selecting random objects, attributes and then unparse the structure to generate human-readable code. Listing 1 shows an example function. The parsing process converts it to an AST as shown in Listing 2 where each variable, operation, function calls is identified and stored in a graph structure. This AST structure makes the code-generation flow independent of any specific programming language, allowing it to be extended to other programming languages.

```
def random_function():
    pins_obj = get_all_pins()
    return pins_obj[5]
```

Listing 1: An example function to demonstrate our random code generation process.

```
Module: entire code
FunctionDef: Function Body
Assign: pins_obj=get_all_pins()
    Name: pins_obj
    Call: get_all_pins()
Return: return pins_obj[5]
    Subscript: pins_obj[5]
        Name: pins_obj
        Constant: 5
```

Listing 2: AST for the example function shown in Listing 1.

The code generation algorithm is detailed in Algorithm 1. The process begins by randomly selecting an operation and a few start nodes (design objects). The algorithm then traverses the nearby connected nodes of the start nodes to gather various attributes of the objects. It applies the selected operation on an attribute with a valid data type matching the selected operation. The function parameters are then initialized with random values and design objects. After several iterations, the AST is converted into a code. By combining this approach with LLM-based comment and code generation, we created an additional 35,000 SFT training data points to cover most of the commands in our target EDA tool.

On these code snippets we extracted the APIs used along with their man page description and used the Nemotron 340B model [19] to add comments to the code and generate

Algorithm 1: Random Synthetic Code Generation Flow

- 1: Select Code structure by randomly choosing one operation (such as condition, math operations, iterators, OOPs operations)
 - 2: Randomly Select and Initialize Classes, Attributes, from tools API graph based on chosen operation
 - 3: Apply operation on selected class attributes and store in an AST format.
 - 4: Initialize function parameters and arguments.
 - 5: Repeat steps 2-5 for random iterations
 - 6: Convert AST to Code
 - 7: Add line by line comments using LLM
 - 8: Generate Questions based on commented code using LLM
-

questions. Listing 3 shows an example of generated code along with question and comments.

Question:

Write a code to find the largest logic delay among a set of violations.

```
# Get the set of violations
vios_obj_1 = get_violations('*')
# Initialize the largest logic delay to a 0
largest_logic_dly = 0
# Iterate over each violation in the set
for vio in vios_obj_1:
    # Compare the current value to largest delay
    if vio.logic_delay() > largest_logic_delay:
        # Update the largest delay
        largest_logic_dly = vio.logic_delay()
    # Print the largest logic delay
    print(largest_logic_dly)
```

Listing 3: Example of a randomly generated code that commented and a question is generated for it.

Once the synthetic SFT dataset is generated, we finetune the ChipNeMo model on it to enhance its understanding of complex structures in VLSI tool scripts, employing the standard autoregressive language modeling objective. We set the learning rate to $5e-6$ using a constant scheduler, apply a weight decay of 0.1, and a batch size of 128. Additionally, we limit training to one epoch to prevent overfitting. The training process utilizes 16 nodes in a compute cluster, each equipped with eight A100 GPUs boasting 80GB of memory. It completes in approximately eight hours.

IV. MULTI-AGENT FRAMEWORK

We present a novel multi-agent framework that leverages the strengths of state-of-the-art (SOTA) LLMs and domain-adapted models to generate high-quality scripts for custom VLSI tools. Our framework builds upon the ReAct framework [24], which we modified to utilize SOTA LLMs like LLaMa 3.1. To further enhance the framework’s capabilities, we developed a multi-agent multi-episode flow that integrates multiple custom tools. Our approach combines the generalized linguistic capabilities of LLMs with the specialized domain knowledge of domain-adapted models like ChipNeMo. As illustrated in Fig. 3, our multi-agent flow consists of a Top Agent, a Code Fixing Agent, a Guardrail Agent and a set of tools including Code Generator, RuleEnforce, Code Compiler, and RAG, to ensure the generation of high-quality code that

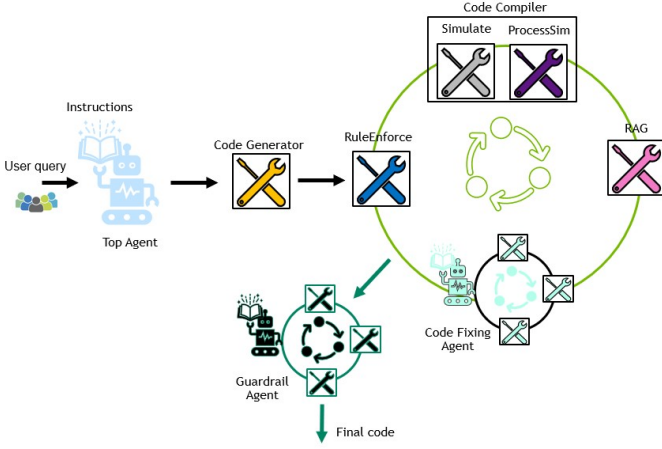


Fig. 3: Overview of the multi-agent flow, listing the various tools utilized to improve code quality: Code Generator, RuleEnforce, Code Compiler, Code Fixing Agent, RAG, and a Guardrail agent.

meets output standards. In the following, we will provide more details on each of these components. The Top Agent is a ReAct agent with instructions on how to generate high quality EDA scripts and with access to the said set of tools and ability to communicate with other agents in the loop.

1) *Code Generator*: We use ChipNeMo as our Code Generator to generate the first answer for any user query. For other tasks, if an LLM is needed, we use a SOTA LLM like LLaMa 3.1 and not ChipNeMo as we observed challenges with conversation follow-up with domain-trained models as they had limited conversation data during training. This challenge can be attributed to missing Preference Fine-Tuning such as Direct Preference Optimization [22], and we plan to improve on this. This alignment process enables the model to follow instructions better and engage in conversations effectively.

2) *RuleEnforce*: Despite the advancements of DAPT and SFT, we observed that LLMs consistently struggle with certain topics, often resulting in partial or incorrect implementations. For example, they may not know how exactly to compute power in the target EDA tool. To address this limitation, we developed RuleEnforce, a tool in the Top Agent’s disposal that leverages special rules, hints, and notes of the target EDA tool. By defining explicit rules and instructions, RuleEnforce enables the agent to accurately apply these rules to relevant code segments.

We employ two methods for rule extraction: manual and auto rule extraction. In the manual approach, we identify common patterns of errors in codes written by the flow through extensive experiments. For instance, we observed that power consumption calculations often require special handling, as illustrated in Listing 4. We wrote a set of rules manually to address these issues, such as the power computation rules shown in Listing 5. In contrast, our auto rule extraction method utilizes a ReAct agent that analyzes a set of QnA data with user queries and golden codes. The agent extracts a set of rules that can generate codes similar to the golden codes, while also consulting with our simulator to prevent simulation

errors. This process is done offline and yields thousands of rules, which are then filtered and ranked by a RAG system using the user query during the run time. The top 10 related rules are appended to our manual rules, enhancing the overall accuracy of the RuleEnforce tool.

Listing 4 demonstrates the effectiveness of RuleEnforce in correcting code errors. The upper code block contains an issue in calculating power (line 4), as the target EDA tool does not have an attribute named `leakage_power` for cell. After applying the special rules for computing power, as shown in Listing 5, the code is revised to produce the corrected output, as demonstrated in the lower code block. Section IV-3 will explain how our Code Compiler can help fix a hallucinated attribute. However, in this example, using only the Code Compiler will not resolve the issue. This is because, although the compiler can detect `leakage_power` as an invalid attribute for cell and suggest related attributes such as power, knowing valid attributes alone is not sufficient. The power computation requires an additional step, as demonstrated above.

```
total_leakage_power = 0
for cell in get_cells("*", "hierarchical"):
    if cell.is_sequential():
        leakage_power = cell.leakage_power
        total_leakage_power += leakage_power
```

```
total_leakage_power = 0
for cell in get_cells("*", "hierarchical"):
    if cell.is_sequential():
        cell.calculate_power()
        leakage_power = cell.power("is_leakage")
        total_leakage_power += leakage_power
```

Listing 4: Example for before and after applying power computation special rules.

You can compute power of a cell as below:

- + First use `Cell.calculate_power()` to compute power values.
- + Next, **access** desired power **type** using flag with the following values:
 - `power.is_leakage`
 - `power.is_dynamic`
 - `power.is_total`
 - `power.is_switching`
- + Examples:
 - `cell.calculate_power()`
 - `leakage = cell.power("is_leakage")`
 - `cell.calculate_power()`
 - `switching = cell.power("is_switching")`

Listing 5: Special rules for computing cell power.

3) *Code Compiler*: LLMs for code generation tasks are prone to mistakes such as incorporating wrong logic, inefficient code, misunderstanding the full intent of the question, and hallucinating. Addressing hallucinations and object-attribute relationship-related issues in limited data scenarios proved challenging because the model does not understand the relationship between various objects and attributes. To fix these issues, we developed an AST-based custom compiler to process LLM-generated code and provide corrective feedback to the model.

We have been focused on an internal high-usage EDA tool, which is object-oriented, interacts in Python, and has many attributes for each object. We use the tool’s man page to

generate a tool command graph similar to what is shown in Fig. 2. The green nodes represent the data-objects, and blue boxes represent the attributes available on the object. We can use these attributes to traverse across different objects, e.g., we can use ‘Pin.net()’ to get the ‘net’ object connected to a ‘pin’.

Our compiler has two main jobs: (1) simulating the code and (2) processing the simulation results and providing useful feedback. For the simulation part, it converts the given initial code into an API-graph and walks through each node. While walking through the nodes, it checks for the existence of each object and attribute in the API-graph and checks for the compatibility of this object-attribute relationship. It returns any lines of the code where there is incompatibility or usage of hallucinated APIs. This is done using the ‘Simulate’ tool in Fig. 3. Next, in case of hallucination, our compiler returns the list of valid attributes for the corresponding object, and in case of wrong object-attribute relationship, it returns the shortest path from the object to the attribute. This is done using the ‘ProcessSim’ shown in Fig. 3.

Listing 6 shows an example for our compiler. It shows a user query, an initial code for it, simulation results, and compiled data for it, followed by refined code. As seen, the shortest path provided by the compiler is used to fix the erroneous line and make it error-free.

```
User query:
Write a code to get all hold violations, if any net
in the vio has a route length greater than 2um.

for node in nodes:
    if not (node.is_net()):
        if node.route_length() > 2:
            filtered_hold_vios.append(vio)

# Simulation results:
Line No. 3: node of datatype Node has no attribute
route_length
# Valid attributes and shortest paths:
Here are a few valid attributes on Node:
Node.pin => Netlist pin object
Node.pin_name => Pin name
Node.pin_report => Report pin object
Here is how to get to route_length from Node:
Node -> pin -> Pin -> net -> Net -> route_length

for node in nodes:
    if not (node.is_net()):
        if node.pin().net().route_length() > 2:
            filtered_hold_paths.append(path)
```

Listing 6: Example showing how our compiler works and what data it provides.

4) *RAG for domain data:* To enhance our script generation framework, we incorporate RAG, leveraging an open-book strategy to retrieve in-domain knowledge from external data stores, enabling the LLM to generate more precise and contextually grounded responses. A common use of RAG is augmenting the initial query with retrieved code examples. However, we observed that providing full code snippets can lead to overfitting in cases where there is a significant gap between the user query and retrieved code. To mitigate this, we optimize RAG for API queries from tool documentation and single-line code retrieval with high accuracy.

Our implementation employs a hybrid search approach, combining BM25 for precise keyword matching with FAISS for capturing semantic relationships. To further improve accuracy, we integrate a re-ranker that merges results from both models and prioritizes the most relevant matches. This RAG framework enhances LLM performance while reducing hallucinations by providing more contextually relevant information.

5) *Code Fixing Agent:* It often happens that we have a script that is mostly correct but there are one or few issues in some lines. These issues sometimes arise because the model had good ideas of what needed to be done but couldn’t do it in a quite accurate way. For example, it was trying to get the top reference of a design, but didn’t use the right APIs or made a mistake on its correct usage. If we knew what was the purpose of an erroneous line, then a small query that specifically asks for that can fetch a small code snippet from our domain expert LLM, ChipNeMo-jarvis, and pass it to the ReAct agent to be used in next iterations for fixing the issues. The intent of a line can be understood systematically by looking up its surrounding comments and analyzing those effectively. What is described is exactly what the code fixing agent does in our framework. We implemented it as a tiny ReAct agent with access to ChipNeMo-jarvis as a tool. This way of implementation was intended to make the code fixing calls more effective and efficient.

6) *Guardrail Agent:* To ensure the quality of the generated code, we introduced a ‘guardrail agent’ that scrutinizes the output before presenting it to the user. This agent is integrated with a LLaMa 3.1 LLM and evaluates the code’s structural and functional correctness. Structurally, our compiler assesses the accuracy of API calls, assigning a score based on the percentage of correct API calls and object-attribute relationships. Functionally, the agent converts the final code’s comments into a question and compares it to the user’s original query, assigning a score accordingly. Only if the code meets both structural and functional quality checks, it is presented to the user; otherwise, a standardized message, such as ‘Sorry, I don’t have enough knowledge to write such a code for you’ is displayed, indicating that the tool is unable to provide a satisfactory response.

```
User query:
Write a code to find the violation ID of a path with
worst negative slack among vios ending in RAM

def get_worst_slack_of_viol(vpaths):
    w_slack = vpaths.sort_using_slack()[0].slack()
    for v in vpaths:
        if v.is_setup_vio():
            slack = v.slack()
            if slack < w_slack:
                worst_slack = slack
            end_pin = v.get_end_pin()
            if end_pin.of_ram():
                return w_slack
    return w_slack
# Call the function with the given vio_id
worst_slack = get_worst_slack_of_viol(
    get_all_violations())

# Get all vios
vios = get_all_violations()
# Filter vios that end at RAM
```

```

ram_vios = [v for v in vios if vio.get_end_ref().
             of_ram()]
ram_vios_v = Violations()
for vio in ram_vios:
    ram_vios_v.push_back(vio)
# Sort vios by slack and get the worst one
worst_slack_vio = ram_vios_v.sort_using_slack()[0]
worst_slack_vio_id = worst_slack_vio.id()

```

Listing 7: A complex example showcasing how our flow improves initial code by using multiple custom tools

Listing 7 shows a relatively complex example on how our flow improves the quality of the code by leveraging multiple custom agents. In this example, the user wants to find a VIO with the worst negative slack among all VIOs ending at RAM. The initial code has incorrect object-attribute relationships, ‘get_end_pin’ and ‘of_ram’ in line 9 of the upper code block, and wrong logic, returning the first VIO ending in RAM instead of the vio with the worst slack. Our flow fixes this by using multiple tools. Our compiler catches the incorrect object-attribute relationship and provides valid attributes. Our code fixing agent catches the intent of the erroneous line, accessing the end reference object of a VIO, and fetches a piece of code for that. Finally, our RuleEnforce tool realizes that we want to sort VIOs and get the worst one; it uses a provided hint to replace the for loop with a single line of code to do the same, which makes the code efficient.

The pseudo code for our multi-agent based code refinement algorithm is presented in Algorithm 2. This algorithm, as detailed above, formalizes the iterative refinement process, which involves formatting, rule enforcement, and simulation, as well as the integration of multiple agents to resolve errors and improve code quality. We implemented a multi-episode version of the discussed algorithm, where Algorithm 2 is called in each episode. This approach effectively addresses several issues present in the native ReAct implementation from Lanchain, including the tendency to trap into infinite or repeated loops where the agent becomes stuck calling the same tool without a clear termination condition. By defining each episode as a sequence of multiple rounds of tool calls and iterative refinements of the code, we obtain a summary of the overall progress at the end of each episode which is fed back in at the beginning of the next episode. This multi-episode implementation has improved the reliability and efficiency of the flow.

V. EXPERIMENTAL RESULTS

We evaluated our method on multiple benchmarks: B-easy (simple API query) with 150 questions, B-medium (2-3 stages of reasoning) with 30 questions, and B-hard (multiple reasoning stages) with 20 questions. The results are summarized in Tables I-II.

To generate the answers in our evaluation, we set the temperature to zero which allowed for more predictable and deterministic code generation, which is essential for tasks requiring high accuracy. We used pass@1 accuracy for our evaluations. B-easy was evaluated using an auto-evaluation method using string match for the desired APIs, while B-medium and B-hard required human evaluation due to the dependency of code executions on design data.

Algorithm 2: Multi-Agent based Code Refinement Algorithm

Input: Query: a coding query by user, TimeLimit, ItrLimit
Output: Refined code

- 1 Initialize Top Agent
- 2 *InitCode* = Code_Generator(*Query*)
- 3 *RefinedCode* = *InitCode*
- 4 *SimResult* = Not Clean, *TimeElapsed* = 0, *ItrCount* = 0
- 5 **while** *SimResult* != Clean and *TimeElapsed* <= *TimeLimit* and *ItrCount* <= *ItrLimit* **do**
- 6 *RefinedCode* = RuleEnforce(*RefinedCode*)
- 7 *SimResult* = Simulate(*RefinedCode*)
- 8 **if** *SimResult* != Clean **then**
- 9 *NewCodes* = {}
- 10 **for each** erroneous line in *SimResult* **do**
- 11 *NewCode* =
- 12 Code_Fixing_Agent(*RefinedCode*, erroneous line)
- 13 Append(*NewCodes*, *NewCode*)
- 14 *ValidAttrs*, *ShortPaths* =
- 15 ProcessSim(*SimResults*)
- 16 *RefinedCode* = Top_Agent(*NewCodes*, *SimResult*, *ValidAttrs*, *ShortPaths*)
- 17 *TimeElapsed* = *TimeElapsed* + *t*
- 18 *ItrCount* = *ItrCount* + 1
- 19 *Output* = Guardrail_Agent(*RefinedCode*)
- 20 **return** *Output*

As we can see, GPT4o and LLaMa 3.1 (with retrieval) achieved close to 0% accuracy on all benchmarks, indicating that off-the-shelf models do not perform well on domain-specific data. This is because these models lack the specialized domain knowledge required for code generation tasks. There are a few instances where these models managed to generate correct answers based on their general VLSI knowledge and their probabilistic nature. However, these instances are rare and do not compensate for their overall poor performance in this domain.

A. Domain training

Domain adaptive training as presented in [14] shows the importance of DAPT and DSFT for models to incorporate the domain information. The initial ChipNeMo trained model utilized existing raw data for DAPT and a manually created dataset of 150 question-and-answer pairs for DSFT. This model demonstrated a significant improvement of approximately 50% over off-the-shelf models. To address the challenge of limited user data, we generated 35,000 synthetic data points for DSFT using small code snippets, as detailed in Section III. This synthetic data notably enhanced the model’s accuracy, particularly on the easy benchmark. While there was a 7% improvement on hard benchmarks, the most substantial gain was observed on the easy benchmark, with a 21% increase. This improvement is attributed to the quality of the synthetic data, as small snippets were selected to avoid the incoherence and multiple local inherited functions that larger chunks tend to produce.

In another experiment to observe the impact of DSFT vs DAPT+DSFT, we conducted SFT training on the foundation

TABLE I: Evaluation results using different LLMs.

Model name	B-easy	B-medium	B-hard
GPT4o (with retrieval)	0%	0%	10%
LlaMa 3.1 (with retrieval)	4%	0%	7%
DAPT+DSFT with manual data (ChipNeMo)	46%	56%	36%
DAPT + DSFT with added synthetic data	67%	62%	43%
DSFT with added synthetic data	68%	36%	18%

TABLE II: Ablation studies for different agent components

Model name	B-easy	B-medium	B-hard
Full feature Multi-agent flow	92%	93%	81%
LlaMa 3.1 with multi-agent flow	53%	43%	30%
Multi-agent flow w/o RuleEnforce	89%	84%	56%
Multi-agent flow w/o RAG	73%	79%	80%
Multi-agent flow w/o code fixing agent	90%	93%	75%

model using the synthetic data. The DSFT only model provided similar gain on the easy benchmark (68% with DAPT only vs 67% with DAPT+DSFT), while the hard benchmark did not see any significant benefit. These results indicate that DSFT effectively helps the model learn domain-specific keywords. However, for complex reasoning tasks, DAPT remains crucial.

B. Multi-agent component ablation studies

We evaluated the effectiveness of the full featured multi-agent flow. The results are presented in Table II, the second row. Comparing it with Table I, the full featured multi-agent flow provides significant improvements on the quality of results across different evaluated benchmarks, as seen in Fig. 4. This demonstrates the great potential of our multi-agent flow. We evaluated different agents and their components to understand their impact on performance. In the following, we provide ablation study on main components of JARVIS.

Initial ChipNeMo calls: For the feedback loop using agentic flow to work effectively, a good starting point is essential. The initial ChipNeMo call provides this foundation. As shown in Table II, starting with an off-the-shelf model like LlaMa 3.1 results in a slight improvement in agentic-feedback flow compared to the initial zero accuracy of LlaMa 3.1 in Table I. This improvement is due to the addition of more domain knowledge. However, the overall accuracy remains significantly lower compared to starting with the domain-trained model.

RuleEnforce: Codes generated for B-hard questions were often incorrect due to numerous internal rules to cross-reference data from different sources such as library, timing-models, physical layout information. These rules need to be applied while writing complex codes. Easy and medium complexity questions have lesser dependency on these rules. Using RuleEnforce helped improve the B-hard accuracy by 25% as seen in Table II.

RAG for domain data: We employ retrieval to access data from tool documentation and existing code. While retrieval significantly improves performance on easy (19%) and medium (14%) benchmarks, its impact on hard benchmarks is minimal (1%). This highlights the need for high-quality, relevant data to reduce hallucinations. Complex reasoning remains challenging with simple documentation retrieval, as it requires understanding interactions between multiple compo-

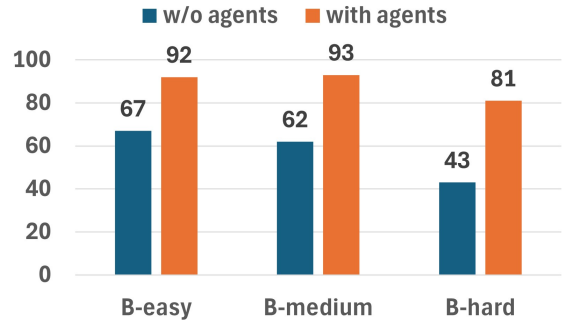


Fig. 4: Comparing best scores without multi-agent flow (4th row in Table I) with the full featured multi agent flow. As seen significant improvements across all evaluated benchmarks are achieved.

nents—especially keywords that may not be explicitly present in the user query.

Code Fixing Agent: For complex codes, it would be better to provide an example code as a feedback along with the error information. So, we used our code fixing agent to fetch example code snippets for erroneous parts of initial codes. We have seen an improvement of about 6% on B-hard by this process, while not much change in B-easy and B-medium.

VI. CONCLUSIONS

We formalize the high-quality EDA script generation task for data-scarce custom tools. To enable LLMs to tackle EDA script generation, we propose an innovative multi-LLM framework that integrates domain knowledge through specialized tools. This framework incorporates multiple custom tools such as custom code-compiler to refine initial obtained codes. Agent-based strategies are developed to optimize the use of these tools and achieve high quality final codes. To evaluate the effectiveness of our method, we construct three evaluation benchmarks, that include a large variety of common script generation scenarios. Experiments show our framework achieves a significant improvement on diverse script generation tasks, highlighting its potential for specialized custom tools.

REFERENCES

- [1] M. AI. Codellama: A large language model for code generation, 2024.
- [2] S. AI. Stablecode: A large language model for code generation, 2024.
- [3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Pondé, et al. Evaluating large language models trained on code. *ArXiv/2107.03374*, 2021.
- [4] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *arXiv/2404.10100*, 2024.
- [5] Y. Ge, W. Hua, K. Mei, j. ji, J. Tan, S. Xu, Z. Li, and Y. Zhang. Openagi: When llm meets domain experts. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, pages 5539–5568, 2023.
- [6] L. Gong, M. Elhoushi, and A. Cheung. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv/2401.03003*, 2024.
- [7] Q. Gu. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 2201–2203, 2023.
- [8] X. Guo and Y. Chen. Generative ai for synthetic data generation: Methods, challenges and the future. *arXiv/2403.04190*, 2024.
- [9] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stolica. Llm-assisted code cleaning for training accurate code generators. *arXiv/2311.14904*, 2023.
- [10] S. Jain, A. Dora, K. S. Sam, and P. Singh. Llm agents improve semantic code search. *arXiv/2408.11058*, 2024.
- [11] H. Koziolk, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani. Llm-based and retrieval-augmented control code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, page 22–29, 2024.
- [12] F. Lin, D. J. Kim, Tse-Husn, and Chen. When llm-based code generation meets the software development process. *arXiv/2403.15852*, 2024.
- [13] Y.-C. Lin, A. Kumar, N. Chang, W. Zhang, M. Zakir, R. Apte, H. He, C. Wang, and J.-S. R. Jang. Novel preprocessing technique for data embedding in engineering code generation using large language model. *arXiv/2311.16267*, 2024.
- [14] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, K. Kunal, Ismet, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv/2311.00176*, 2023.
- [15] Z. Liu, W. Ping, R. Roy, P. Xu, C. Lee, M. Shoenybi, and B. Catanzaro. Chatqa: Surpassing gpt-4 on conversational qa and rag. *arXiv/2401.10225*, 2024.
- [16] L. Long, R. Wang, R. Xiao, J. Zhao, X. Ding, G. Chen, and H. Wang. On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv/2406.15126*, 2024.
- [17] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.
- [18] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-T. Yih, S. Wang, and X. V. Lin. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pages 26106–26128, 2023.
- [19] Nvidia, :, B. Adler, N. Agarwal, A. Aithal, D. H. Anh, P. Bhattacharya, A. Brundyn, J. Casper, et al. Nemotron-4 340b technical report. *arXiv/2406.11704*, 2024.
- [20] B. Project. Starcoder: A large language model for code generation, 2024.
- [21] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y.-T. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. H. Gerstein, D. Li, Z. Liu, and M. Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv 2307.16789*, 2023.
- [22] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv/2305.18290*, 2024.
- [23] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.
- [24] S. Yao, D. Yu, J. Zhao, I. Shafraan, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- [25] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv/2401.07339*, 2024.