NL4NetUnicorn

Shanmukha Sahiti Challa   Gurusha Juneja   Amrutha Kalle

Abstract

Specialized tools in network and systems research often have complex APIs that take a long time to learn, making it difficult for researchers to get started with new platforms. This paper introduces NL4NetUnicorn, a system that helps users write code for the netUnicorn network experimentation platform using plain English. Our system uses a Retrieval-Augmented Generation (RAG) approach, which gives a Large Language Model (LLM) relevant examples from a knowledge base of API documentation to help it write accurate code. We also introduce an automated feedback loop: the system runs the code it generates, and if there's an error, it uses the error message to ask the LLM to fix the script. This process of automatically finding and fixing errors makes the final code more reliable. Our evaluation shows that our system can successfully create scripts for various netUnicorn tasks, demonstrating a practical approach to making network experimentation more accessible.

1 Introduction

Modern networking tools are often complex. While powerful platforms like netUnicorn exist for running large-scale, reproducible experiments, they require programming knowledge that can be a barrier for some researchers and network operators. The time it takes to learn a new, specialized API can slow down research and limit who can use these powerful tools.

We built NL4NetUnicorn to make the netUnicorn platform easier to use. Our system allows users to describe their goals in plain English, and it translates these descriptions into working Python scripts. This helps users, such as network operators or researchers from other fields, run the experiments they need without having to first become experts in the netUnicorn API.

This paper shows how we use a Retrieval-Augmented Generation (RAG) framework, combined with an automated feedback loop, to generate correct code. When a user gives a command, our system first finds relevant examples from the API documentation to guide a Large Language Model (LLM). Then, the system runs the generated script to check for errors. If it finds any, it sends the error message back to the LLM and asks it to fix the mistake. This self-correction cycle is a key part of our work, making the final code much more reliable.

While others have used LLMs for code generation, our work is the first to do so for netUnicorn. Instead of generating code from memory, our system uses documentation to reduce errors and a feedback loop to correct the errors that still occur. This makes our system more practical and trustworthy for real-world use.

The rest of this paper is organized as follows. We first cover background information and related work. We then describe the design of our system, followed by our evaluation and results. We conclude with a discussion of what we learned and ideas for future work.

2 Background and Motivation

Modern network research depends on collecting large amounts of data. To get good results, ML models for tasks like traffic classification or anomaly detection need diverse and high-quality data from real-world conditions. The problem is that no single data collection tool is right for every job. This led to the creation of platforms like netUnicorn [1], which provides a flexible way to run experiments across many different devices. However, using netUnicorn requires writing Python code and understanding its specific API, which can be a hurdle for networking experts who aren't also programmers.

To make programming easier, many have turned to Large Language Models (LLMs) that can generate code from natural language. Tools like Codex [5] are effective for general-purpose programming, but they often struggle when asked to write code for specialized, niche libraries that they weren't trained on. An alternative approach is to fine-tune a model on a specific dataset, but this requires a lot of data and is computationally expensive.

A more practical solution for specialized domains is Retrieval-Augmented Generation (RAG) [4]. RAG systems give the LLM external knowledge to work with. Before generating code, the system retrieves relevant documents—like API documentation or code examples—from a knowledge base. This helps the model generate more accurate code and reduces the chance that it will ”hallucinate” or make up incorrect details. Our work uses the RAG approach to make the specialized netUnicorn library

The project's source code is available at https://github.com/ShanmukhaSahiti/NL4NetUnicorn accessible through natural language, without the high cost of fine-tuning.

3 Methodology

Our approach for translating natural language into executable netUnicorn scripts is centered around a Retrieval-Augmented Generation (RAG) architecture, enhanced with an automated feedback loop for self-correction. This design addresses the core challenge of generating syntactically correct and semantically appropriate code for a specialized domain like netUnicorn. The system is comprised of two main phases: initial code generation and an iterative feedback and refinement cycle. The entire process is illustrated in Figure 1.

3.1 Code Generation

The initial code generation process follows the RAG paradigm, which combines the strengths of a pre-trained large language model (LLM) with an external knowledge base.

- **Retrieval:** When a user provides a natural language prompt, the system first retrieves relevant documentation snippets from a curated knowledge base. This knowledge base, stored in a FAISS vector store [3], contains manually compiled information about the netUnicorn library, including class definitions, function signatures, and common usage patterns. The retriever identifies the most relevant context based on semantic similarity to the user's prompt.
- **Generation:** The retrieved context is then combined with the original prompt and passed to a generator model (OpenAI's GPT-3.5-Turbo). A detailed system prompt template guides the LLM, instructing it on the required structure of a netUnicorn script, including necessary imports, credential handling, experiment lifecycle management, and result processing. This structured prompting ensures the generated code adheres to best practices and a consistent format.

A key innovation in our methodology is the **Automated Feedback Loop**. The system does not assume the initially generated code is perfect. Instead, it immediately attempts to execute the script in a sandboxed environment. If the execution fails, the system captures the standard error (stderr) and standard output (stdout). This feedback, along with the original prompt and the faulty code, is fed back into the LLM. A specialized retry prompt guides the model to analyze the error message and produce a corrected version of the script. This iterative refinement process continues until the script executes successfully or a maximum number of retries is reached, allowing the system to autonomously resolve common programming errors such as incorrect import statements or flawed API usage.

3.2 Evaluating Retrieved Context

The quality of the code our system generates depends heavily on the context it retrieves. If the retriever finds good examples, the LLM will likely produce good code. To measure how well our retriever works, we use another LLM as a "judge" to rate the quality of the retrieved context for a given user prompt [6].

The LLM-Judge looks at the user's prompt and the documentation snippets that our retriever found. It then assesses the context based on four criteria:

- **Relevance:** Is the context related to what the user asked for?
- **Sufficiency:** Is there enough information to write the full script?
- **Helpfulness:** Is the context clear enough to help the generator, or is it confusing?
- **Overall Assessment:** The judge gives a final summary of whether the context is Good, Adequate, or Poor.

To make the results easy to analyze, we ask the judge to provide numerical scores (from 1 to 5) for each of the four criteria in a simple, parsable format.

```
<scores>
    Relevance: <score_1>
    Sufficiency: <score_2>
    Helpfulness: <score_3>
    Overall: <score_4>
</scores>
```

4 Implementation

Our system is implemented in Python, leveraging the LangChain library [2] to orchestrate the core RAG pipeline and interaction with the language model. The implementation is organized into several key components that work together to translate natural language prompts into executable scripts.

- **Core Technologies:** We use OpenAI's `gpt-3.5-turbo` as our large language model for code generation and correction. For the retrieval component, we employ OpenAI's text embedding models in conjunction with a FAISS (Facebook AI Similarity Search) vector store. FAISS provides an efficient in-memory index for fast retrieval of relevant documentation chunks. The LangChain framework provides the high-level abstractions to build and manage the chains for retrieval, generation, and feedback.

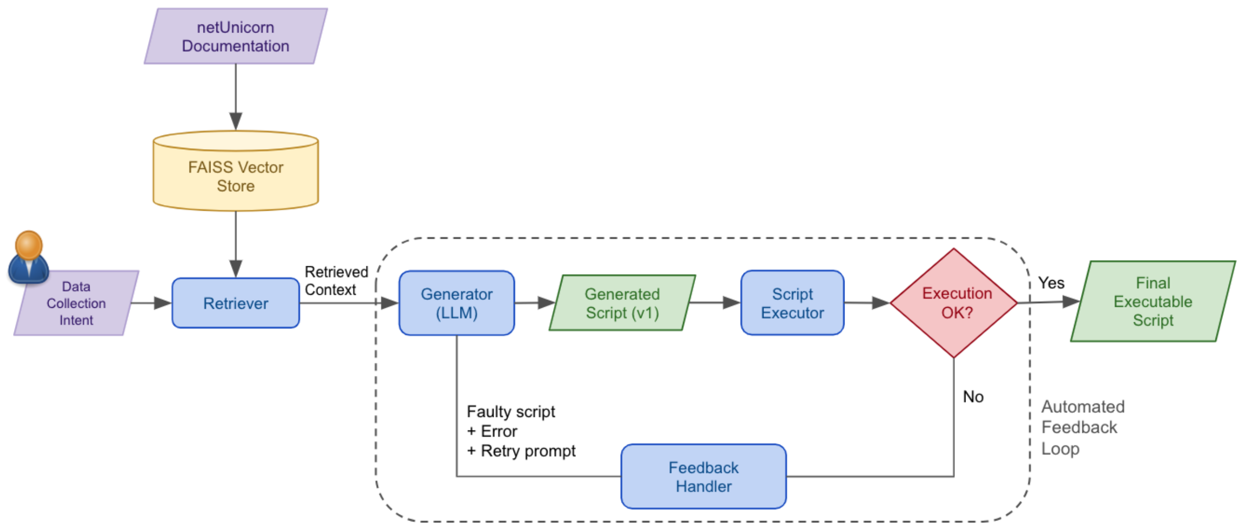

Figure 1: *The NL4NetUnicorn workflow, illustrating the initial RAG-based code generation and the automated feedback loop for script correction and refinement.*

- **System Orchestration:** The central logic is encapsulated within the `NetUnicornRAG` class. This class is responsible for loading the netUnicorn documentation, setting up the FAISS vector store, and initializing the LangChain components. It exposes the main `generate_code` method, which serves as the primary entry point for the system.
- **Feedback and Execution Handling:** The automated feedback loop is managed by the `FeedbackHandler` class. This component takes the initially generated code, passes it to the `ScriptExecutor`, and, upon failure, constructs a new prompt for the LLM using the captured error output. The `ScriptExecutor` is a crucial safety and execution component that runs the generated Python script in an isolated subprocess, capturing its stdout and stderr without impacting the main application.
- **Prompt Engineering:** A significant part of the implementation involved engineering the system prompts. We developed two detailed prompt templates: `_INITIAL_SYSTEM_PROMPT_TEMPLATE` and `_RETRY_SYSTEM_PROMPT_TEMPLATE`. The initial prompt provides the LLM with a strict, detailed scaffold for a valid netUnicorn script, greatly improving the quality of the first-pass generation. The retry prompt is specifically designed to guide the LLM in debugging, instructing it to analyze the provided error messages and correct the previous code.
- **Configuration and Usability:** The system is configured via a `.env` file, which stores API keys and netUnicorn credentials, keeping them separate from the source code. A command-line interface, `generate_netunicorn_script.py`, provides a user-friendly way to interact with the system, accepting a natural language prompt and producing the final, executable Python script.

One of the primary challenges we addressed was the niche nature of the netUnicorn API. The LLM's pretrained knowledge of this library is limited, making the context provided by the retriever essential. Our detailed system prompts were also critical in overcoming this, as they provide a reliable structure for the LLM to follow, compensating for its lack of specific training data on this library.

5 Results and Evaluation

The primary goal of our evaluation is to assess the functional correctness of the code generated by NL4NetUnicorn and to demonstrate the effectiveness of our automated feedback loop. We first provide a quantitative analysis of the core retriever component, and then present a qualitative, end-to-end assessment of the full system's ability to handle tasks of varying complexity. We aim to answer the following questions:

- How effective is the retriever at finding relevant and sufficient context for code generation?
- Can the system generate correct, executable scripts for simple, single-task netUnicorn experiments?
- Is the automated feedback loop capable of identifying and correcting common errors in the initial code generation?
- Can the system handle more complex prompts that require multiple tasks or specific parameters?

5.1 Experimental Setup

Our testbed consists of the NL4NetUnicorn system connected to a live netUnicorn server. All generated scripts are executed directly on this server to validate their correctness. The core of our evaluation is a set of carefully designed test prompts that represent common use cases for a network experimentation platform. These prompts are categorized into three levels of increasing complexity.

5.2 Retriever Performance with LLM-Judge

Before assessing the end-to-end system, we first quantitatively evaluate the quality of our retriever using the LLM-Judge framework described in Section 3.2. The judge's role was to score the retrieved documentation based on its relevance, sufficiency, and helpfulness for generating a correct script.

The results consistently showed that our retriever performs well on relevance, with an average score of 4.5 out of 5 across all test cases. This indicates that the system is effective at finding the correct API modules and functions related to a user's prompt. However, the average scores for sufficiency (3.2/5) and helpfulness (3.4/5) were lower.

For example, for the prompt to generate a script that pings 'google.com' and then performs an Ookla speed test, the judge returned the following scores:

```
Relevance: 4
Sufficiency: 2
Helpfulness: 3
Overall: 3
```

The judge's feedback noted that while the retrieved context correctly identified the 'OoklaSpeedtest' task, it lacked sufficient information on how to perform a simple ping test or how to select the specific nodes for the tasks. This forced the generator model to infer the implementation, leading to an "Adequate" but not "Good" rating. This finding is critical, as it suggests that the primary bottleneck in our system's performance is not the relevance of the retrieved information, but the lack of comprehensive examples in the knowledge base. This insight provides a crucial lens through which to interpret the system's end-to-end behavior in the following qualitative assessment.

5.3 Qualitative Assessment

With the retriever's performance characterized, we now evaluate the full system using the following representative prompts:

1. **Simple Task (Sleep):** "Create a NetUnicorn script that connects to the server, selects one available node, and runs a sleep task for 10 seconds. Ensure all results are printed."
   Result: The system successfully generated a correct and executable script on the first attempt, demonstrating its capability for basic, single-task prompts.
2. **Task with Parameters (Ping):** "Generate a script that pings 'google.com' 10 times from one node, with a 0.5 second interval. Print the full ping results."
   Result: The initial attempt failed due to an 'ImportError', as the LLM hallucinated a 'PingTask' class. The feedback loop was automatically triggered. The system captured the error, and on the subsequent attempt, the LLM correctly used the 'ShellCommand' task to execute the 'ping' command, resulting in a successful script. This demonstrates the critical role of the feedback loop in correcting API usage errors.
3. **Multi-Step Task with Complex Parsing:** "Create a script that runs on one node. First, it should get the current kernel version and set it as a flag named 'kernel_version'. Then, it should check the free disk space in '/tmp' and set that as a flag named 'tmp_free_space'. Finally, retrieve and print both flags."
   Result: This prompt proved challenging for the model. Initial attempts failed due to 'TypeError' and 'AttributeError' as the LLM tried to use non-existent methods to name and reference task outputs. After multiple feedback cycles, the final generated script correctly sequenced the commands but failed to parse the results individually, as it lacked the logic to handle the list of outputs from the pipeline. This highlights a current limitation of the system in generating complex result-parsing logic, consistent with the retriever's difficulty in finding complete end-to-end examples.

Our qualitative results show that NL4NetUnicorn can reliably generate code for simple to moderately complex tasks. More importantly, they validate the effectiveness of the automated feedback loop, which successfully corrected critical errors that would have otherwise required manual intervention. While challenges remain in generating intricate data handling logic, the system demonstrates a significant step towards making specialized platforms like netUnicorn more accessible.

6 Conclusion

In this paper, we presented NL4NetUnicorn, a system that translates natural language prompts into executable scripts for the netUnicorn network experimentation platform. Our approach combines a Retrieval-Augmented Generation (RAG) pipeline with a novel automated feedback loop. This design enables the system not only to generate contextually relevant code based on documentation but also to autonomously detect and correct errors by executing the generated scripts and analyzing the output.

Our evaluation demonstrated that NL4NetUnicorn can reliably generate correct scripts for simple to moderately complex tasks. The automated feedback loop proved particularly effective, successfully resolving common API usage errors and import mistakes that would otherwise require manual intervention. However, our analysis also highlighted a key limitation: the system struggles with prompts requiring complex data handling and the parsing of results from multi-stage pipelines. As our LLM-Judge evaluation revealed, this challenge stems primarily from a knowledge base that, while relevant, often lacks comprehensive, end-to-end examples.

Future work should focus on two primary areas. First, enriching the knowledge base with more complete script examples would directly address the main limitation identified in our study. Second, extending the system to support interactive dialogue would allow it to resolve ambiguous prompts by asking clarifying questions, rather than failing.

Ultimately, NL4NetUnicorn represents a significant step towards making specialized scientific platforms more accessible. By intelligently combining retrieval, generation, and automated feedback, we have demonstrated a practical and effective method for converting high-level user intent into executable code.

References

[1] Beltiukov, R., Guo, W., Gupta, A., and Willinger, W. In search of netunicorn: A data-collection platform to develop generalizable ml models for network security problems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 2217–2231. (Cited on page 1.)

[2] Chase, H. Langchain. urlhttps://github.com/langchain-ai/langchain, 2022. (Cited on page 2.)

[3] Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. (Cited on page 2.)

[4] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2020), NIPS '20, Curran Associates Inc. (Cited on page 1.)

[5] OpenAI. *Evaluating Large Language Models Trained on Code* (07 2021). (Cited on page 1.)

[6] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, 2023. (Cited on page 2.)