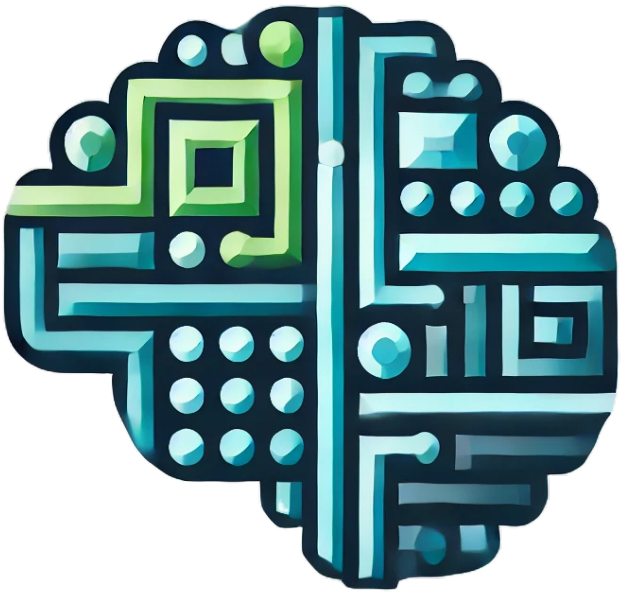
Generating Terraform Configuration Files with Large Language Models

Cybersecurity M



Studenti:

Nicole Giulianelli

Andrea Berti

Vincenzo Salvemini

Docente:

Prof. Michele Colajanni

Anno Accademico 2024/2025

Abstract

This study explores the integration of Large Language Models (LLMs) into the management of infrastructure using Terraform, with a specific focus on enhancing cybersecurity. The primary objective is to investigate how LLMs can automate the generation, analysis, and optimization of infrastructure configurations, thereby minimizing human errors and addressing vulnerabilities associated with misconfigurations.

LLMs demonstrate remarkable capabilities in translating natural language descriptions into complete configuration files and in identifying vulnerabilities within existing setups. Beyond detection, these models can suggest and automatically implement additional security measures to fortify overall infrastructure resilience.

This feature is particularly advantageous for engineers who, while proficient in Terraform and major cloud platforms, may lack advanced expertise in network security. The system offers proactive and intelligent support, combining automation and adaptability to address evolving threats effectively.

The report also examines the technical challenges associated with applying LLMs to Terraform code generation, characterized by its block-structured syntax and the scarcity of well-annotated training data. To overcome these hurdles, the study proposes fine-tuning models using specialized datasets.

In addition to theoretical exploration, this study presents a practical implementation framework that integrates LLMs into the Terraform workflow. The framework automates key stages such as configuration generation, validation, and iterative security enhancements, showcasing how these tools can be deployed in real-world scenarios.

Future developments will focus on enhancing the contextual awareness of these models and fine-tuning them with datasets featuring real-world examples of deception configurations, such as honeypots and decoy resources. Additionally, the implementation of API endpoints will replace the graphical user interface, ensuring greater scalability, flexibility, and automation in the deployment of configurations.

Contents

Chapter 1

Terraform

1.1 Introduction

Terraform is a leading Infrastructure-as-Code (IaC) tool designed to provision and manage resources across various platforms, including public clouds like AWS, Google Cloud Platform (GCP), as well as on-premises environments. It utilizes declarative, human-readable configuration files written in HashiCorp Configuration Language (HCL). Terraform's ability to manage resources such as compute, storage, and networking components makes it a versatile choice for modern infrastructure automation.

```
terraform {
  required_providers { # Specifies the required provider for the configuration (AWS)
    aws = {
      source = "hashicorp/aws" # Source of the AWS provider (HashiCorp's official registry).
    }
  }
}
provider "aws" {
  profile = "default" # Configures the AWS provider to use the default profile.
  region = "us-east-1" # Configures the AWS provider region to 'us-east-1'.
}
resource "aws_instance" "ec2demo" {
  ami = "ami-0be2609ba883822ec" # Sets Amazon Machine Image to use for the instance launch.
  instance_type = "var.instance_type" # Sets instance type using the 'instance_type' variable.
}
variable "instance_type" {
  default = "t2.micro" # Defines variable with a default value of 't2.micro'.
  description = "EC2 Instance Type" # A description of the variable.
  type = string # Specifies the type of the variable as a string.

}
```

1.2 Execution Workflow

Given a configuration file, Terraform follows these steps:

1. **Initialization**: Prepares the working directory by downloading provider plugins.
2. **Plan Creation**: Execution plan generated to outline the Terraform infrastructure changes.
3. **Apply Changes**: Implements the execution plan by creating, updating, or destroying resources.

1.3 Security in Terraform Configurations

Terraform configurations play a critical role in managing and provisioning infrastructure. However, ensuring their security is challenging due to the complexity and scale of modern infrastructures. Common security risks include: **misconfigurations**, **human errors**, and **lack of standardization**.
Automation tools that enhance security, such as those powered by LLMs, help mitigate these risks by enforcing consistency, **identifying potential security issues** by performing an analysis of the Terraform configuration, and **automatically generating a new secure configuration** by incorporating industry best practices and policies.

1.4 Challenges for LLMs in Generating Terraform Code

While LLMs excel in generating Python functions, adapting them to Terraform configuration files presents unique challenges:

- **Complexity of Block Structure**: Unlike Python functions, Terraform configurations consist of multiple discrete blocks, each serving a distinct purpose. The lack of a clear endpoint in these files makes automated generation non-trivial;
- **Data Availability and Context Understanding**: Python has abundant training data. In contrast, public Terraform files often lack comprehensive comments. Instead, functionality is inferred from the `description` argument within blocks or the naming conventions of variables and resources.

1.5 Conclusion

Integrating LLMs to assist with configuration generation and analysis requires addressing unique challenges. Nonetheless, with proper adaptation, these models can play a pivotal role in automating and securing infrastructure management.

Chapter 2

Large Language Models (LLMs)

Large Language Models (LLMs) represent a significant advancement in natural language processing (NLP) by utilizing deep learning techniques to predict, generate, and understand text. These models are trained on massive datasets, enabling them to comprehend and generate human-like text across a wide range of tasks. Their impact extends beyond NLP to areas such as software development, research, and education.

2.1 Core Attributes of LLMs

LLMs derive their capabilities from the foundational principles of the transformer architecture, pretraining on diverse datasets, and fine-tuning for specific applications. Key attributes of LLMs include:

- **Transformer Architecture**: LLMs are typically built on the transformer architecture, which uses a self-attention mechanism to analyze relationships between words in a sequence. This architecture excels at capturing long-range dependencies, making it suitable for tasks requiring contextual understanding.
- **Scalability and Parallelization**: Unlike Recurrent Neural Networks (RNNs), which process input sequentially, transformers allow for parallel computation, significantly reducing training time and enabling scalability to billions of parameters.
- **LLMs training phases**:
  - **Pretraining**: The model is trained on a general corpus, often including text from books, websites, and repositories. This phase teaches the model language structure, syntax, and semantics. For code-focused LLMs, the corpus includes structured datasets such as GitHub repositories.
  - **Fine-tuning**: After pretraining, the model is further refined on domain-specific data or tasks to enhance its performance in specialized applications.
- **In-context Learning**: LLMs exhibit the ability to learn tasks directly from input context, requiring minimal or no additional fine-tuning.

- **Generative Capabilities**: Decoder-based LLMs, particularly autoregressive models, excel in text generation by predicting the next token based on prior tokens. These capabilities extend to code generation, where models like CodeLLaMa leverage structured code data to produce syntactically correct and functionally relevant outputs.

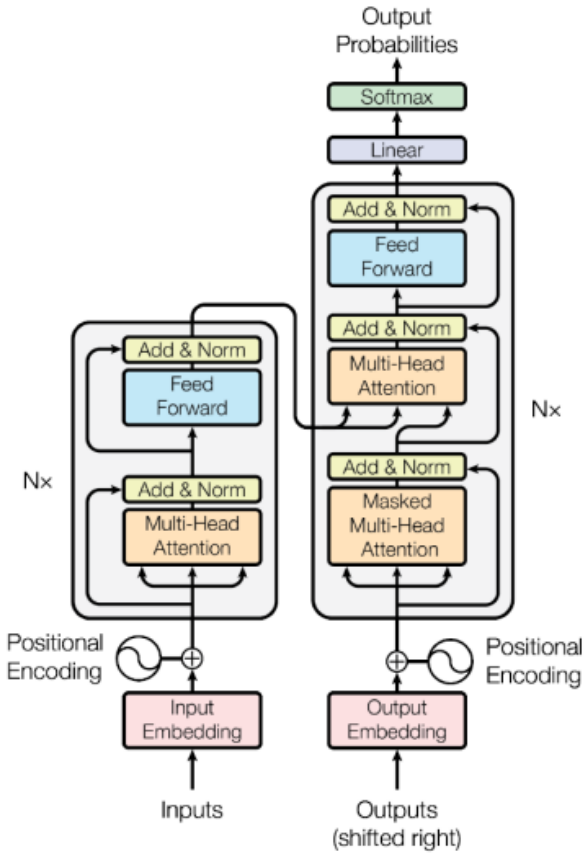


Figure 2.1: Trasformer Architecture

2.2 LLaMa

The **Large Language Model Meta AI (LLaMa)** represents a family of open-access models designed to democratize LLM research and reduce barriers associated with proprietary systems. LLaMa emphasizes transparency and accessibility, providing researchers with tools to explore and advance LLMs. LLaMa models are characterized by:

- **Open-source Availability**: LLaMa provides open access to models of varying sizes, enabling research without the restrictions of commercial APIs.
- **Scalability**: With parameter counts ranging from millions to billions, LLaMa scales to accommodate diverse computational resources.
- **Adaptability**: The models are pre-trained on a wide range of data, making them versatile for downstream tasks such as summarization, translation, and sentiment analysis.

The release of LLAMA aims to foster innovation by allowing the academic community to experiment with large-scale LLMs while prioritizing ethical considerations, including minimizing carbon footprint and promoting responsible AI development.

2.3 CodeLLaMa

CodeLLaMa, a derivative of LLAMA, focuses specifically on code generation and understanding. It leverages LLAMA's robust transformer architecture and adapts it for programming tasks. Key features include:

- **Pretraining on Code**: CodeLLaMa is trained on a comprehensive dataset of programming languages, including Python, Java, and C++, sourced from repositories like GitHub. This enables the model to learn syntax, structure, and best practices in software development.
- **Fine-tuning for Code Tasks**: Beyond pretraining, CodeLLAMA is fine-tuned for specific use cases such as code completion, bug fixing, and documentation generation.
- **Enhanced Context Handling**: CodeLLaMa excels in managing long context windows, allowing it to work effectively with extensive codebases and providing accurate suggestions based on project-wide context.
- **Applications in Software Engineering**: Developers use CodeLLaMa for automating repetitive tasks, generating boilerplate code, and even solving complex programming challenges. Its capabilities align closely with the demands of modern software engineering, enhancing productivity and reducing error rates.

CodeLLaMa exemplifies the potential of domain-specialized LLMs, combining general NLP advancements with targeted pretraining and fine-tuning to address the specific needs of coding and software development.

Chapter 3

Tokenization

Tokenization is the process of breaking down text or code into smaller, manageable units, or tokens, that can be efficiently processed by large language models (LLMs). Tokenization is a critical step in natural language and code processing, as it converts human-readable input into numerical representations.

3.1 Challenges in Tokenization

Different tokenization strategies can impact the efficiency and accuracy of a language model. The selection of a tokenizer must balance representation fidelity, vocabulary size, and computational efficiency. A smaller vocabulary reduces computational overhead but may require more tokens to represent a given input, potentially impacting performance. Conversely, a larger vocabulary provides more precise representation at the cost of increased computational demands.

3.2 Tokenization Strategies

Tokenizers fall into several categories based on their granularity and method of segmentation:

- **Character-Level Tokenization**: Character-level tokenizers assign a unique ID to each character. This approach can represent any text but often requires a large number of tokens, making it computationally expensive. For instance, a text containing 180 characters would need 180 tokens. This granularity ensures complete text representation but at the cost of efficiency.
- **Word-Level Tokenization**: Word-level tokenizers split text into words and assign each word a unique ID. This reduces the number of tokens required for text representation, as each word corresponds to a single token. However, words not included in the tokenizer's vocabulary cannot be processed directly, leading to potential limitations. Word-level tokenizers require large vocabularies—typically in the range of 50,000 tokens—which demand significant computational resources during training and inference.
- **Subword Tokenization**: Subword tokenization strikes a balance between character-level and word-level approaches by breaking text into smaller units based on frequency. Byte Pair Encoding

(BPE) is a common subword tokenization method. It merges the most frequent sequences of characters into tokens, allowing the representation of both common words and less frequent or out-of-vocabulary sequences through combinations of smaller units. A byte-level variant of BPE enables the encoding of all Unicode characters while maintaining a fixed base vocabulary size, typically as small as 256 tokens. This flexibility is beneficial for handling diverse inputs, including rare characters, without expanding the vocabulary size significantly.

Figure 3.1: Tokenization example

3.3 Advanced Approaches

Some tokenization methods extend beyond traditional character or subword-based models to optimize performance for specific domains.

3.3.1 CodeLLaMa

CodeLLaMa, a specialized LLM for programming tasks, employs tokenization strategies specifically tailored for code. Unlike general-purpose tokenizers, CodeLLaMa treats key elements of programming languages, such as indentation, brackets, and keywords, as distinct tokens. This granularity improves the model's understanding of syntax and structure, enabling accurate processing of a wide variety of programming languages. Additionally, CodeLLaMa balances vocabulary size and computational efficiency, ensuring scalability for large codebases while maintaining high performance.

Chapter 4

Implementation

The primary objective of the project is to create an automated pipeline for managing and deploying Terraform configurations, integrating deception elements generated through advanced language models such as Codellama. This approach enables dynamic and iterative infrastructure management, optimizing time and reducing manual errors. The solution includes advanced features for file reading, interaction with LLM models, and automated management of the Terraform lifecycle, with a simple and intuitive user interface developed in Streamlit.

4.1 Structure and Operations

4.1.1 User Interaction

The user interface is implemented using Streamlit, allowing users to:

- Input custom prompts to request specific changes to Terraform files.
- Select a LLaMa model for processing (e.g., `llama3.2` or `codellama`).
- Specify whether processing should be limited to files with the `.tf` extension to optimize performance.

4.1.2 File Management

- **Configuration reading:** A function reads all files in the specified directory, focusing on Terraform files (`.tf`) if indicated. The content of the files is used to construct a context to pass to the LLaMa model.
- **Configuration update:** Once the updated configuration is generated by the model, it is saved in a new file to replace the original configuration file.

4.1.3 Integration with LLaMa

The LLaMa model is started locally via a dedicated server:

```
@st.cache_resource
def start_ollama_server():
    try:
        subprocess.Popen(
            ["ollama", "serve"], stdout=subprocess.PIPE, stderr=subprocess.PIPE
        )  # Avvia il server Ollama in background
    except Exception as e:
        print(f"Errore durante l'avvio del server Ollama: {e}")
```

- **Prompt generation:** The content of existing Terraform files is concatenated with a user-provided prompt to create a contextualized input.

```
default_prompt = """Il codice del file 'main.tf' deve essere riportato in modo completo
e preciso, aggiungendo nuovi dispositivi di deception o modificando quelli esistenti
in modo coerente con l'infrastruttura Terraform.
La risposta deve contenere solo il nuovo codice modificato del file 'main.tf'."""
```

- **Model processing:** The LLaMa model processes the prompt and returns updated configurations in textual format.

```
def query_ollama(model, prompt):
    try:
        response = requests.post(
            "http://localhost:11434/api/generate",
            json = { "model": model, "prompt": prompt, "stream": False }
        )
        return response.json()["response"]
    except requests.RequestException as e:
        print(f"Errore durante la connessione a Ollama: {e}")
        raise e
```

4.1.4 Terraform Deploy Pipeline

The script automates all essential deploy Terraform phases:

```
def execute_terraform_deploy(terraform_dir):
    if not os.path.exists(terraform_dir):
        print(f"Errore: la directory Terraform {terraform_dir} non esiste.")
        return
    try:
        subprocess.run([os.path.join(terraform_dir, "terraform"), "init"],
        cwd=terraform_dir, check=True)
```

```
        ....
        #Operazione "validate"
        #Operazione "plan"
        #Operazione "apply"
        ....
    except Exception as e:
        print(f"Errore durante il deploy della configurazione Terraform: {e}")
        raise e
```

- **Initialization:** Configures the Terraform environment for the project.
- **Validation:** Verifies the correctness of the Terraform configuration.
- **Planning:** Creates a detailed plan of infrastructure changes.
- **Approval and Deployment:** Automatically applies changes to the infrastructure.

4.1.5 Asynchronous Automation

Using an asynchronous function based on `asyncio`, the pipeline can run periodically with random intervals, ensuring continuous configuration updates and fast deployment.

```
async def periodic_deploy(prompt=default_prompt):
    while True:
        try:
            with st.spinner("Generazione del codice Terraform in corso..."):
                # Directory contenente i file Terraform
                terraform_directory = os.path.join(BASE_DIR, "terraform_architecture")

                # Costruzione prompt completo...

                # Invia il prompt al modello Codellama
                response = query_ollama(model_choice, full_prompt)
                print("\nRESPONSE:\n", response)
                # Parsing della risposta
                response_parsed = parse_string(response)
                print("\nRESPONSE PARSED:\n", response_parsed)
                # Salva nuovo file Terraform generato
                new_main_file = os.path.join(BASE_DIR, "new_main.tf")
                save_to_file(response_parsed, new_main_file)

                # Salva vecchia configurazione Terraform in caso deploy fallisca
                old_main_file = os.path.join(BASE_DIR, "old_main.tf")
                backup_terraform_file(os.path.join(terraform_directory, "main.tf"), old_main_file)
```

```
23
24                  # Update file Terraform
25                  update_terraform_file(new_main_file,
26                  os.path.join(terraform_directory, "main.tf"))
27              except Exception as e:
28                  st.error(f"Errore durante la generazione del codice terraform: {e}")
29                  return
30
31              try:
32                  with st.spinner("Deploy in corso..."):
33                      # Esegui la pipeline Terraform
34                      execute_terraform_deploy(terraform_directory)
35                      #se il deploy fallisce viene ripristinata la configurazione precedente
36
37              interval = random.randint(300, 900)  # Range casuale tra 5 e 15 minuti
38              print(f"Attendo {interval} secondi prima del prossimo aggiornamento.")
39              await asyncio.sleep(interval)  # Pausa asincrona
```

4.2 Interaction Between Components

- **Streamlit and User Input:** Collects user prompts and controls main settings.
- **Configuration Reading:** Generates the prompt for the model by analyzing the files in the specified directory.
- **LLaMa Server:** Processes the request and returns an updated configuration based on specified requirements.
- **Configuration Update:** Updated files are saved and synchronized with the existing architecture.
- **Terraform Deploy Pipeline:** Validates, plans, and applies configurations, completing the automation cycle.

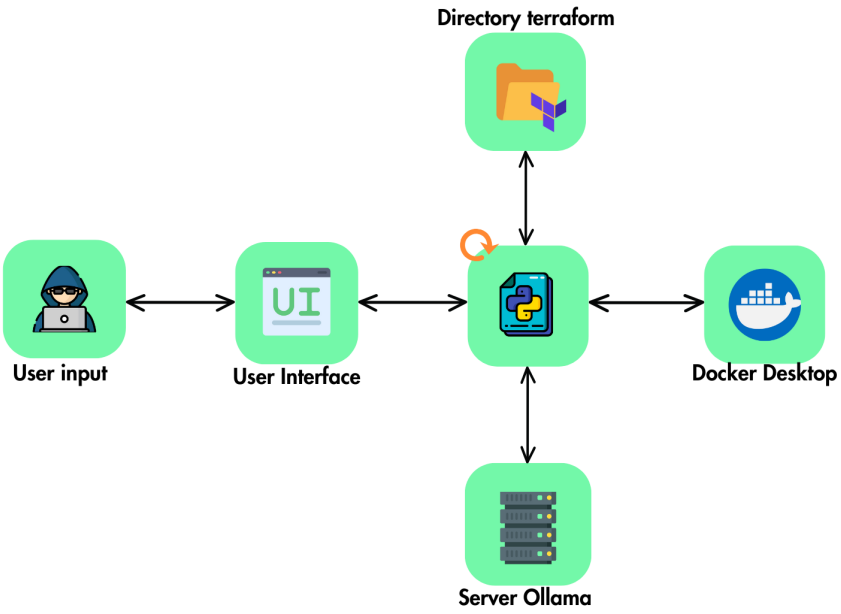


Figure 4.1: Interaction Between Components

4.3 LLMs and Benefits

The creation of an automated pipeline that periodically generates and updates Terraform configurations while modifying deception elements offers numerous security benefits.

- **Periodic updates:** Deception elements, such as honeypots or decoy servers, are regularly updated, making it difficult for attackers to identify and map such resources over time.
- **Configuration evolution:** Each update can include new configuration and deployment patterns reflecting the latest threats, enhancing the effectiveness of the countermeasure.
- **Uncertainty for attackers:** Frequent configuration changes create uncertainty for attackers' reconnaissance tools, such as network scanners or resource enumerators.
- **Scalability:** The pipeline can easily be adapted to generate and manage more complex configurations.

4.4 Final considerations and Future Developments

During the current phase of the project, several tests were conducted using different models and parameters, including StarCoder2, CodeLLAMA, and CodeGemma. The results of tests on models with 3B or less parameters indicate significant challenges in handling complex requests or scenarios requiring deep problem understanding. Additionally, these models are prone to producing code with syntactic or logical errors at a higher frequency. Useful results can only be achieved through targeted fine-tuning.

Models with 7B parameters, on the other hand, demonstrated rapid generation capabilities, albeit with limited precision. They can address moderately complex problems with reasonable understanding and produce fewer errors compared to smaller models. However, their overall knowledge and consistency remain constrained. Among these, CodeLLAMA exhibited the best performance under comparable conditions. Models with 13B or more parameters offer advanced knowledge, covering a broader range of programming languages and advanced techniques. These models deliver high-quality results even without fine-tuning, but their use requires significantly higher computational resources and processing time.
The primary reason for the prevalent generation of static honeypots as deception components lies in the limitations of the 7B-parameter model. Being less extensively trained than larger models like GPT-4, the 7B model tends to produce simpler components due to limited exposure to sophisticated examples. To overcome this limitation and generate more advanced deception elements, fine-tuning with domain-specific datasets would be necessary.

Additionally, mechanisms will be implemented to enhance the model's contextual awareness, allowing it to better understand the purpose of the configurations, such as protecting sensitive resources or simulating critical infrastructure.

Finally, the graphical user interface will be replaced with a set of API endpoints, ensuring greater scalability, flexibility, and automation in the management and deployment of configurations.

4.5 GitHub Repository

https://github.com/bertiandrea/TerraformAI