Sapienza
Università di Roma

Automation of Network Infrastructure Configuration through NLP and Software Orchestration

Sapienza University of Rome
Master's Degree in Engineering in Computer Science

Stefano Rucci
ID number 1794004

Advisor
Prof. Ioannis Chatzigiannakis

Academic Year 2024/2025

Automation of Network Infrastructure Configuration through NLP and Software Orchestration
Master's Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: rucci.1794004@studenti.uniroma1.it

Abstract

The process of setting up a network infrastructure is challenging and time consuming because it requires a full knowledge of vendor-specific commands, which often vary between vendors.

This thesis introduces an innovative approach based on *Natural Language Processing* (NLP) *and automation* to simplify and standardize network settings.

The proposed system, built on Amazon Lex V2, provides to users a chat-bot interface, allowing them to specify their configuration requirements in natural language. Based on user input, the system automatically generates a configuration file adapted to the target device. Configurations are then deployed by Ansible without requiring human involvement.

This method decreases the effort required by system developers, removing the need for vendor-specific command knowledge and improving scalability for integrating more devices and services. The outcomes show a notable increase in flexibility and efficiency, opening the door for further network administration automation powered by AI.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

Basic network operations often require a technician with specialized expertise in multiple vendor ecosystems: this dependence on vendor-specific configurations introduces *inefficiencies*, forcing network administrators to spend significant time and effort on *repetitive* and low-level tasks. This does not allow them to focus on strategic activities like network architecture, security enforcement, and performance optimization. Declarative automation simplifies configuration management by abstracting vendor-specific details, allowing engineers to concentrate on high-level intent rather than low-level syntax.

In order to ensure the reliable operation of IT infrastructures, *network configuration* is a crucial step. However, manually configuring network devices has several disadvantages, such as the requirement for in-depth knowledge of vendor-specific command syntax, which makes the process time-consuming and prone to human error.

Towards a Smarter Approach

The growing demand for automation and scalability in IT infrastructures strongly encourages adopting solutions that minimize manual intervention and speed up configuration.

The system proposed in this thesis:

- *Interprets* user natural language commands;
- *Dynamically generates* the appropriate configuration settings;
- *Automatically applies* them to network devices of any vendor.

1.2 Problem Statement and Objectives

Problem Statement

Traditional network configuration methods are based on vendor-specific command-line interfaces (CLIs) and engineers need to learn all the syntax rules for each device. Ansible, Python, and CLI template scripting are currently used for automation, but they still require a significant amount of technical expertise.

Considering limitations, these strategies remain not accessible to non-experienced users.

Objectives

This thesis aims to develop an automated network configuration system that:

- *Eliminates vendor dependency* – Translates user requests from natural language into vendor-specific configurations.
- *Reduces the need for deep technical expertise* – Allows network engineers to configure devices using simple intent-based commands.
- *Minimizes human errors* – Reduces human mistake by automating the configuration procedure.
- *Improves scalability* – Enables easy expansion to support new network devices and services.
- *Enhances operational efficiency* – Reduces the time required for configuring network devices.

This study suggests a system that can automatically deploy configuration files across heterogeneous network settings and dynamically generate them based on user input by utilizing automation with Ansible and Natural Language Processing (NLP) with Amazon Lex V2.

1.3 Personal Contribution

This thesis is based on a research and development (R&D) initiative that was started by the IT consultancy company *Capgemini*. The goal of the project was to create a tool that used *Natural Language Processing* (NLP) to automatically configure network devices. I was involved in this project when Capgemini was looking for an artificial intelligence specialist to help with the system's development. I was asked to join the team because of my AI experience, and the project became the basis for my master's thesis.

Here is a summary of my main contributions:

- *Chatbot interface development* – I spent a lot of time creating and deploying the chatbot that interprets user input. At first, I considered building the chatbot from scratch, using the SpaCy NLP toolkit, but my thesis advisor recommended AWS services because they offered a more reliable and scalable solution, eliminating the need to develop the chatbot interface manually.

- *Generation of the inventory file* – One of the main difficulties was the generation of the `.ini` inventory file dynamically based on user input. In order to work, Ansible needs this inventory file where all informations required to configure the network device are inserted. I created a system to convert user commands into an Ansible-processable structured inventory file.

- *Testing and evaluation* – I carried out comprehensive testing with a range of use cases to guarantee the system's scalability and dependability. I also created a test script to verify the consistency of the configurations between the user's request and the configuration performed.

I designed the system to be modular and scalable, allowing for easy extension of:

- *New services* – Currently, the system supports SSH connections, port enable/disable, VLAN configuration, DHCP, and SNMP. Adding new services is straightforward due to the modular architecture.

- *New vendor devices* – The system was designed to support multiple network vendors by configuring Ansible dynamically according to each brand's requirements. Currently, the switches brands *ExtremeXOS* [4] (on which most of the tests were done) and *Cisco* are supported.

This work proves how a combination of NLP and automation tools can reduce the complexity of network configuration, enabling IT professionals to focus on more strategic tasks rather than manual device setup.

1.4 Thesis Structure

This thesis is structured into eight chapters, each addressing a key aspect of the research, design, implementation, and evaluation of the proposed system.

- **Chapter 2 - Background**: Provides an overview of the key technologies and frameworks used in this project, including AI, Natural Language Processing (NLP), AWS services, Ansible, and GNS3.

- **Chapter 3 - System Requirements**: Defines the *functional* and *non-functional* requirements of the system, including scalability, modularity, and automation constraints.
- **Chapter 4 - System Design**: Describes the overall architecture of the proposed solution, detailing how different components interact and how the system was designed to be vendor-independent.
- **Chapter 5 - Implementation**: Explains in detail the development of the chatbot, the generation of the inventory file, the automation process using Ansible and the consistency test script.
- **Chapter 6 - Use Cases and Results**: Presents different test scenarios, analyzing the performance and scalability of the system in various network topologies.
- **Chapter 7 - Limitations and Future Works**: Evaluates the results achieved, discusses the system's limitations, and suggests potential future improvements.
- **Chapter 8 - Conclusions**: Summarizes the results of the thesis and highlights the overall contributions of the work.

This structure follows a logical progression, from defining the problem and selecting appropriate technologies, to designing and implementing the solution, and finally evaluating its effectiveness through practical use cases. The goal is to provide a comprehensive understanding of the system's capabilities and potential impact on network automation.

Chapter 2

Background

2.1 Introduction to Key Technologies

The implementation of an automated network configuration system requires the integration of multiple technologies, each focusing on a specific aspect of the process.

Amazon Web Services (AWS)

It is used to develop a chatbot capable to recognize users natural language requests and convert them into structured configuration commands. AWS provides a set of cloud services, including Amazon Lex for NLP, along with backend tools for data storage, authentication, and workflow automation. It is also responsible for generating the inventory `.ini` file and storing it within the S3 bucket by means of a Lambda [15].

Ansible

Network automation is a crucial component in eliminating manual configuration tasks and reducing human errors. Ansible is an open-source tool designed for infrastructure automation, particularly suited for managing network devices from different vendors. In this project, this technology is responsible for applying configurations generated by the chatbot, ensuring consistent deployment of network settings across different devices [5, 16].

GNS3

Testing and validating an automated network configuration system in a real-world scenario can be expensive and complex due to the need for physical devices. To overcome this challenge, GNS3 is used as a network simulation platform. GNS3 allows the creation of virtual network environments, enabling realistic testing of different topologies and configurations. This approach guarantees that the system is validated before being deployed in production environments. [9]

These technologies form the basis of the proposed system, allowing for a simplified, innovative, scalable, and vendor-independent approach to network configuration. The following sections will provide a more in-depth discussion of each technology.

2.2 Natural Language Processing and AI

Artificial Intelligence (AI) is a multidisciplinary field of computer science that focuses on the creation of systems capable of performing tasks typically requiring human intelligence, such as *learning*, *reasoning*, and *problem-solving* [3]. The concept of AI dates back to the 1950s with the foundational work of Alan Turing, who proposed the idea of machines capable of exhibiting intelligent behavior [13]. Over the decades, AI has evolved significantly, transitioning from rule-based expert systems to modern machine learning and deep learning models.

Machine Learning (ML), a subset of AI, involves training models on large datasets to recognize patterns and make predictions without explicit programming [3]. ML techniques can be broadly classified into three categories: *supervised learning*, where models learn from labeled data; *unsupervised learning*, where models identify patterns in unlabeled data; and *reinforcement learning*, where models learn through trial and error. Deep Learning (DL), an advanced branch of ML, utilizes neural networks with multiple layers to extract high-level features from raw data, enabling applications such as image recognition and natural language understanding [8].

Natural Language Processing (NLP), a key domain of AI, enables computers to *process* and *understand* human language [2]. It combines computational linguistics, machine learning, and deep learning techniques to analyze and generate text. Fundamental NLP techniques include tokenization, lemmatization, part-of-speech tagging, named entity recognition, and syntactic parsing. According to Jurafsky & Martin (2023), modern NLP models rely on deep contextual embeddings that allow for more accurate and dynamic language understanding, significantly improving applications such as chatbots and automated assistants. [7]

Modern NLP systems have been revolutionized by the introduction of deep learning models such as Word2Vec, GloVe, and transformer-based architectures like BERT and GPT. One of the key advancements in NLP is the introduction of contextual word representations, such as BERT, which significantly improved natural language understanding by leveraging bidirectional transformers. [1] These models leverage large-scale datasets to improve language understanding and generation, enabling applications such as machine translation, sentiment analysis, and conversational AI [2].

In the context of this project, NLP is used to facilitate automated network configuration by allowing users to specify their intents in natural language.

The next section describes the AWS services used to implement NLP functionalities within this system.

2.3 AWS Services for NLP and Data Management

Amazon Web Services (AWS) is one of the leading *cloud computing platforms*, providing scalable and flexible services. Over time, it has expanded to include everything from virtual machines to advanced AI and NLP capabilities [15].

AWS offers several key advantages that makes it a dominant player in the cloud computing and AI landscape:

- *Scalability*: AWS provides automatic scaling capabilities that allow applications to handle varying workloads efficiently. Services like Elastic Load Balancing (ELB) and Auto Scaling ensure that applications remain responsive under different loads [15].
- *Security and Compliance*: AWS follows rigorous security standards, offering encryption, identity and access management (IAM), and compliance with various industry regulations, making it a trusted platform for enterprise applications [15].
- *Integration with AI and Machine Learning*: AWS offers pre-built AI services, such as Amazon Lex for NLP, Amazon Comprehend for text analysis, and Amazon SageMaker for building and deploying machine learning models [15].
- *Cost Efficiency*: AWS follows a pay-as-you-go pricing model, allowing businesses to pay only for the resources they use, reducing upfront infrastructure costs [15].
- *Global Infrastructure*: AWS operates in multiple regions and availability zones, ensuring redundancy and minimal latency for global applications [15].

Amazon Lex

Amazon Lex is a service for building conversational interfaces using speech and text. It leverages deep learning-based automatic speech recognition (ASR) and natural language understanding (NLU) to create advanced chatbot applications [15].

This service allows developers to design conversational agents that can process user requests and generate meaningful responses. Moreover, Amazon Lex enables users to submit input requests through both *text* and *voice commands*, enhancing flexibility and accessibility in automation systems [7]. The use of Transformer architectures, introduced by Vaswani et al. (2017), has significantly improved sequence processing tasks, enabling chatbots like Amazon Lex to understand and generate human-like responses with greater contextual accuracy. [14]

AWS Lambda

AWS Lambda is a serverless computing service that automatically runs code in response to events that triggered it. It eliminates the need for managing infrastructure while ensuring scalability [15]. Lambda is widely used for event-driven architectures, enabling seamless integration between AWS services, for example between Amazon Lex and Bucket S3.

Amazon DynamoDB and S3

For data management, AWS offers DynamoDB, a NoSQL database service that provides low-latency and high-performance data storage, and Amazon S3, a scalable object storage service. These services are crucial for storing structured and unstructured data, making them essential components of modern cloud architectures [15].

AWS provides a comprehensive ecosystem for NLP applications, allowing businesses and developers to build intelligent, scalable, and efficient solutions.

2.4 Ansible for Network Configuration

Ansible is an open-source automation tool designed to simplify the configuration, management, and deployment of IT infrastructures [5]. Unlike traditional configuration management tools, Ansible is agentless, meaning that it does not require any software to be installed on the target systems, reducing complexity and overhead [10]. Its structured approach to network automation, based on predefined execution tasks, allows for a more efficient and error-free configuration process, reducing manual intervention and improving operational consistency [16]. This characteristic makes Ansible highly efficient for network automation, allowing it to interact with various network devices using protocols such as SSH, API, and Netconf.

2.4.1 Key Features of Ansible

Ansible provides several advantages over traditional automation tools, making it ideal for network configuration:

- *Agentless Architecture*: Ansible uses SSH or WinRM to connect to remote systems, eliminating the need for additional software installations [5].
- *Idempotency*: Ensures that operations only make changes when necessary, preventing unintended modifications and ensuring consistency [10].
- *Modularity*: Uses modules and playbooks to define infrastructure as code, making it highly extensible [5].
- *Cross-platform Support*: Can manage Linux, Windows, network devices, cloud services, and more [10].
- *Scalability*: Designed to manage thousands of nodes efficiently with simple *YAML*-based configuration [5].

2.4.2 Ansible Playbooks

Playbooks are one of the core components of Ansible. They are written in *YAML* and define a set of tasks that need to be executed on a group of hosts [5]. These structured files serve as reusable blueprints for network configuration, ensuring uniformity across multiple devices while minimizing configuration errors and inconsistencies. Playbooks facilitate automation by enabling network engineers to apply and maintain configurations in a scalable and version-controlled manner [16].

Each playbook consists of:

- *Hosts*: Specifies the target machines or network devices.
- *Tasks*: A list of automation steps, executed sequentially.
- *Modules*: Pre-built or custom Ansible modules that execute specific commands.
- *Variables*: Dynamic values used to customize configurations.
- *Handlers*: Special tasks triggered by changes in configuration.

An example of a simple Ansible playbook for configuring a network switch:

```
- name: Configure Network Switch
  hosts: switches
  gather_facts: no
  tasks:
    - name: Configure VLAN
      ios_config:
```

```
        lines:
          - vlan 10
          - name Management
    - name: Save Configuration
      ios_command:
        commands:
          - write memory
```

This playbook configures a VLAN on a Cisco switch using the `ios_config` module and saves the configuration.

2.4.3 Use of Ansible in Network Automation

Ansible is widely used in network automation to manage *switches, routers, firewalls*, and other network devices [10]. Its vendor-agnostic approach abstracts low-level CLI commands into structured playbooks, allowing for a standardized and scalable automation process. By leveraging Ansible's declarative nature, engineers can efficiently implement complex network configurations while reducing reliance on vendor-specific knowledge [16]. With support for multiple network vendors, it enables consistent and repeatable automation across heterogeneous environments.

Some of the most commonly used Ansible modules for network automation include:

- *ios_config*: Used for managing Cisco devices.
- *nxos_config*: Used for managing Cisco Nexus switches.
- *juniper_config*: Used for managing Juniper network devices.
- *fortios_config*: Used for managing Fortinet firewall configurations.
- *ansible.netcommon.net_ping*: Generic module for testing network connectivity.

Recent studies demonstrate how NLP can be used to interpret high-level user commands and automatically generate structured network configurations, bridging the gap between human intent and automated execution. [6]

2.5 GNS3 for Network Simulation

GNS3 (Graphical Network Simulator-3) is an open-source network simulation platform that allows users to *design, configure*, and *test* complex network environments without requiring physical hardware [9]. It provides an intuitive graphical interface that enables network engineers and system administrators to create virtual network topologies using real network device images and emulated environments.

One of the key advantages of GNS3 is that it eliminates the need for expensive physical network hardware: instead of purchasing physical routers and switches, engineers can leverage virtual instances of real network operating systems, such as Cisco IOS, ExtremeXOS, Juniper JunOS, and others. These virtual devices can run on GNS3 using official vendor-provided images with valid licenses, ensuring an accurate and realistic simulation environment [11].

2.5.1 Key Features of GNS3

GNS3 offers several advantages for network simulation and testing:

- *Graphical Interface*: Provides an easy-to-use drag-and-drop environment for building network topologies.
- *Real Hardware Emulation*: Supports running actual Cisco IOS, Juniper JunOS, and other network operating systems.
- *Integration with Virtual Machines*: Compatible with VirtualBox, VMware, and Docker, allowing for advanced network testing.
- *Packet Capture*: Allows real-time traffic analysis using tools like Wireshark.
- *Multi-vendor Support*: Enables testing of cross-platform network configurations.
- *Official Licenses*: Supports the use of vendor-provided images with authentic licenses, ensuring that the network behavior closely matches real-world environments.

2.5.2 Use Cases in Network Simulation

GNS3 is widely used in network research, certification training, and enterprise network testing [11]. Some key applications include:

- *Network Certification Training*: Used by candidates preparing for CCNA, CCNP, and other networking certifications.
- *Testing New Configurations*: Allows engineers to validate network changes before deployment.
- *Troubleshooting Scenarios*: Simulates network failures and performance bottlenecks.

- *Security Testing*: Helps in evaluating firewall rules, intrusion detection systems, and security policies.
- *Automation Testing*: Validates the effectiveness of network automation scripts before deploying them in production.

Chapter 3

System Requirements

System requirements define the expected functionality and operational characteristics of the software. Their correct classification and documentation ensures compliance with the project objectives and user needs.

This chapter describes the requirements of the proposed system, dividing them into *functional* and *non-functional requirements*, with a section dedicated to current limitations and constraints. The distinction between these categories follows the recommendations of *IEEE 830-1998*, which defines functional requirements as the description of interactions between the system and its users, while non-functional requirements as constraints on performance, security and other qualitative characteristics of the system [12].

3.1 Functional Requirements

Functional requirements are directly derived from the expected user interactions and the core functionalities necessary for automated network configuration.

Below there is a list of functionalities from the user's point of view.

- *Chatbot Interface*: The system offers a chatbot interface that allows users to interact using text and voice in natural language. If required configuration details are missing, the chatbot explicitly asks the user to provide them. Additionally, it validates input parameters (e.g., checking for valid IP addresses).
- *Automatic Generation Inventory file*: Once the chatbot has gathered and validated the required parameters, the system automatically generates an *inventory.ini* file containing all necessary configuration details. This file is then used for the automated configuration process, ensuring that the correct parameters are applied to the appropriate devices.

- *Automated Network Configuration*: The system automatically applies network configurations based on the *inventory.ini* file, ensuring consistency and accuracy across all managed devices.
- *Error Handling*: The system must be able to alert the user if any errors arise during configuration, and in this case restores the last valid configuration to prevent network instability.
- *Accuracy Check*: The system must be able to verify that the final device configuration matches the user's request. If discrepancies are detected, the user is notified with a detailed report highlighting inconsistencies.
- *Multi-Vendor and Service Support*: The system supports devices from multiple vendors and a variety of network services. Its modular design allows easy expansion to integrate additional vendors and services without major architectural changes.

These functional requirements define what the system must do, but it's also essential to talk about *non-functional requirements*.

3.2 Non-Functional Requirements

Non-functional requirements describe the system's qualitative characteristics such as *scalability, security*, and *performance*.

According to the IEEE 830-1998 standard, non-functional requirements specify constraints on the system's operation, ensuring that it meets performance benchmarks, reliability standards, and usability criteria [12]. These requirements are crucial for ensuring system efficiency, maintainability, and adaptability, especially in large-scale IT infrastructures that rely on automated network configuration.

Here follows a list of qualitative and technical aspects of the system.

- *Scalability*: The system must be independent with respect to the number of configured devices (*1,2,..,n*), and therefore must work in the same way with a single device, a complex infrastructure, multiple subnets and different vendors.
- *Reliability*: The system must be fault tolerant and prevent corrupt or incorrect configurations. In case of failure during configuration, it must guarantee an automatic rollback to the last valid configuration to avoid disruptions. Each configuration must be verified before deployment, ensuring correctness and preventing network errors.
- *Usability*: It is essential for the system to have an intuitive and user friendly interface for the chatbot with which the user interfaces. Through to it, the

user is able to interact with the system and configure the network devices without knowing the specific instructions. Furthermore, the chatbot guides the user towards the correct configuration in case of errors or doubts in the request made.

- *Performance*: The system must guarantee a fast response time, both for the chatbot and for the execution of the configuration. The generation of the *inventory.ini* file and the application of the configurations must take place in a few seconds, without introducing delays in the workflow.
- *Expandability*: The system is modular, allowing the addition of new services and new vendors without radically changing the architecture. The entire project is designed to facilitate the integration of new features.
- *Security*: The system implements mandatory SSH Authentication. Before any configuration, the user must provide valid SSH credentials for the target device. Without this authentication, no operation can be performed.

3.3 Constraints and Limitations

While the proposed system is designed to automate network configuration effectively, certain constraints and limitations must be acknowledged. Some of these limitations are inherent to the current implementation, while others derive from architectural choices and can be improved in future versions.

- *Single-device interaction in chatbot* – The chatbot is designed to handle one device at a time. If a user wants to configure multiple devices, they must initiate separate interactions for each. However, once the inventory file is generated, Ansible can configure multiple devices simultaneously, provided that the inventory file includes correct and consistent parameters.
- *Authentication requirements* – Before performing any configuration, the system requires SSH authentication. Currently, there is no centralized user management or role-based access control (RBAC), meaning authentication is handled manually for each session.
- *No graphical interface* – The system relies on a chatbot and command-line tools, with no graphical dashboard available for managing configurations visually. While this does not impact core functionality, a web-based interface could improve usability and monitoring.
- *Limited service support* – The system supports a predefined set of network services (e.g., VLAN, SSH, DHCP, SNMP, switch port management). More

advanced networking features such as firewall rules, routing protocols, or complex network policies are not yet implemented but can be added due to the modular architecture.

- *Limited vendor support* – The system actually supports ExtremeXOS e Cisco vendor, and it is always tested with ExtremeXOS. More vendors can be easily added in the system due to the modular architecture.
- *Ansible dependency* – Configuration deployment is entirely handled by Ansible. Consequently, incorrect inventory files or misconfigured playbooks could cause failures. While error handling is implemented, alternative deployment methods (e.g., direct API calls to network devices) are not yet available.
- *No versioning of configurations* – The system can rollback to the last applied configuration, but it does not currently maintain a full version history of past configurations, that would allow users to restore specific previous states if needed.

These limitations do not compromise the functionality of the system but highlight areas where future improvements could enhance usability, security, and flexibility.

Chapter 4

System Design

This chapter presents the architecture of the overall system, with a particular focus on the main components and their interactions to achieve a *full automatic configuration* of devices starting from the user's natural language input.

The architecture is designed to be easily expandable, allowing the integration of new network services and additional device vendors without requiring major modifications.

4.1 Overall System Architecture

The architecture of the system follows a *modular* and *distributed* design, ensuring flexibility and independence between its components, and each module is responsible for a specific task. This choice was made to guarantee that different components could evolve independently without requiring great changes to the entire system. A monolithic architecture would be initially easier to implement, but would have limited scalability and adaptability when introducing new vendors or services.

At the core of the system is *Amazon Lex V2* chatbot, which acts as the primary interface for user interactions. The chatbot allows users to configure networks using natural language through text or voice. It is vendor-agnostic, meaning it extracts the necessary parameters from user input without needing details about the device manufacturer. The output of this component is always the same, regardless the vendor of the device. By choosing Amazon Lex instead developing a custom NLP model from scratch with libraries (like SpaCy or TensorFlow), the focus of the project is on automation logic and system integration. This approach eliminated the need to spend time in fine-tuning and training a custom chatbot, leveraging Amazon Lex's pre-trained natural language understanding (NLU) capabilities.

Once all necessary information has been gathered, the chatbot triggers an *AWS Lambda function*, which performs input validation, guaranteeing that the provided parameters are valid (e.g., valid IP addresses, VLAN IDs). Once validated, the Lambda function generates a structured inventory file (*inventory.ini*), which contains all parameters needed for configuring the target network device. This file is then stored in an *Amazon S3 bucket.* The use of AWS Lambda instead of a traditional server-based approach was jutified by the serverless computing advantages, including:

- Automatic Scaling: Lambda executes only when triggered, guaranteeing that resources are allocated dynamically based on demand, eliminating the need to manage persistent servers.

- Cost Efficiency: unlike always-on virtual machines, Lambda charges only for execution time, reducing costs significantly for a system that operates on-demand.

- Integration with AWS Services: as an AWS-managed service, Lambda natively integrates with S3, API Gateway, and other AWS components.

Similarly, the choice of Amazon S3 as the storage backend for inventory files guarantees persistent and globally accessible storage, allowing configuration files to be retrieved by Ansible. Alternatives like traditional databases (e.g., MySQL, PostgreSQL) were considered, but they would have added unnecessary complexity, given that inventory files are simple structured text files rather than relational data requiring SQL queries. So a NoSQL database is the best choose.

At this point, the automation process is handled by *Ansible*, which is responsible for applying the network configuration based on the generated *inventory.ini* file. Unlike the chatbot, this part detects at run time the vendor of each network device via SSH. Based on this detection, it selects and executes the appropriate vendor-specific playbook, and also applies the correct services playbook according to what is written in the inventory file. The choice of this technology was motivated by several factors:

- Agentless Architecture: unlike other tools that require installing agents on network devices, Ansible operates via SSH reducing compatibility issues.

- Declarative Configuration Model: Ansible defines the desired state rather than executing imperative command.

- Built-in Multi-Vendor Support: Ansible provides modules for various network operating systems, simplifying the integration of different vendors without extensive re-engineering.

When Ansible has completed its task, a Python *validation script* ensures that the deployed configuration matches the intended state.

The decision to implement a custom validation script in Python rather than relying only on Ansible's built-in checks was based on the need for detailed verification and error reporting. While Ansible can detect immediate failures (e.g., if a command fails to execute), it does not verify if the final network state aligns with the user's request.

Python was chosen for its extensive libraries for SSH automation (Paramiko), regular expressions (re), and structured output handling. This allows the script to:

- Retrieve real-time configurations directly from the switch.
- Compare actual configurations with the expected values in the inventory.
- Detect discrepancies, such as misconfigured ports or wrong VLAN names, that Ansible alone might not identify.

This additional validation layer enhances the reliability of network automation, ensuring that the system enforces the desired network state.

The final stage of the process involves the network devices, which receive the applied configurations and are monitored for correctness.

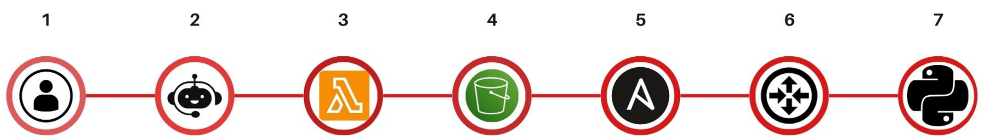

Figure 4.1. High-Level System Flowchart: From User Input to Network Configuration

4.2 Workflow and Data Flow

This section shows the complete workflow of a user request, explaining how data flows between different system components and how configurations are generated, stored, applied, and verified.

The workflow of the system can be broken down into the following key phases:

1. *User Input Processing* - The chatbot interprets the user request and extracts configuration parameters.

2. *Inventory File Generation* – An AWS Lambda function validates inputs, creates the *inventory.ini* file and store it in S3 Bucket.

3. *Automated Configuration Deployment* – Ansible retrieves the inventory and applies the requested configuration or rollbacks in case of failure.

4. *Configuration Validation and Accuracy Check* – A Python script verifies that the configuration matches the user request.

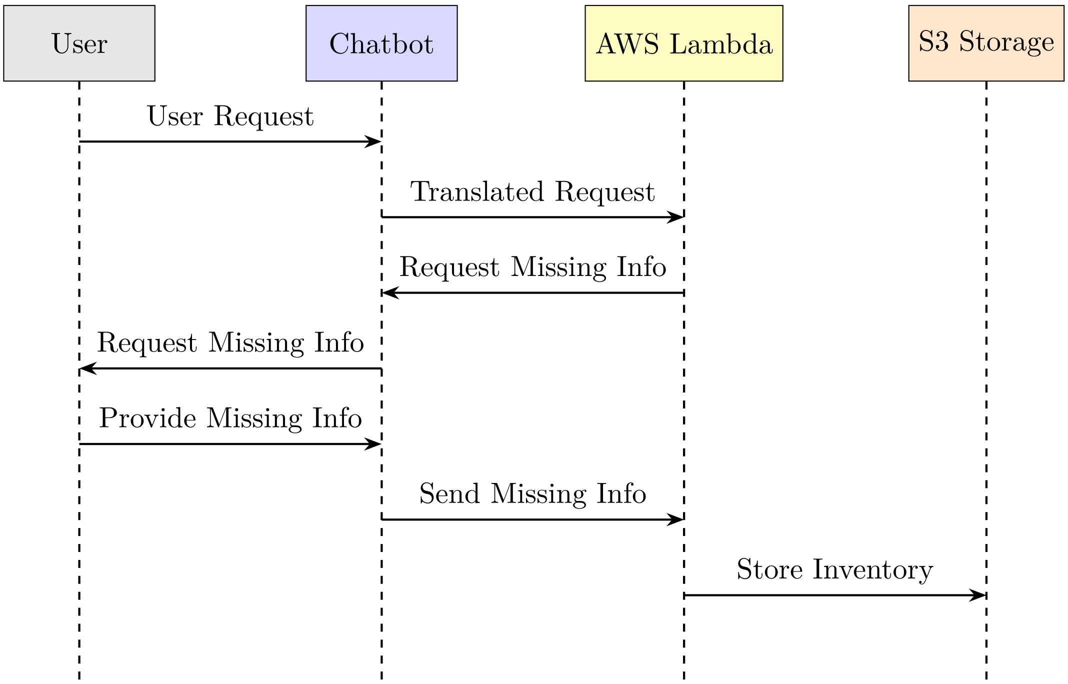

Figure 4.2. Sequence Diagram: Inventory.ini File Generation

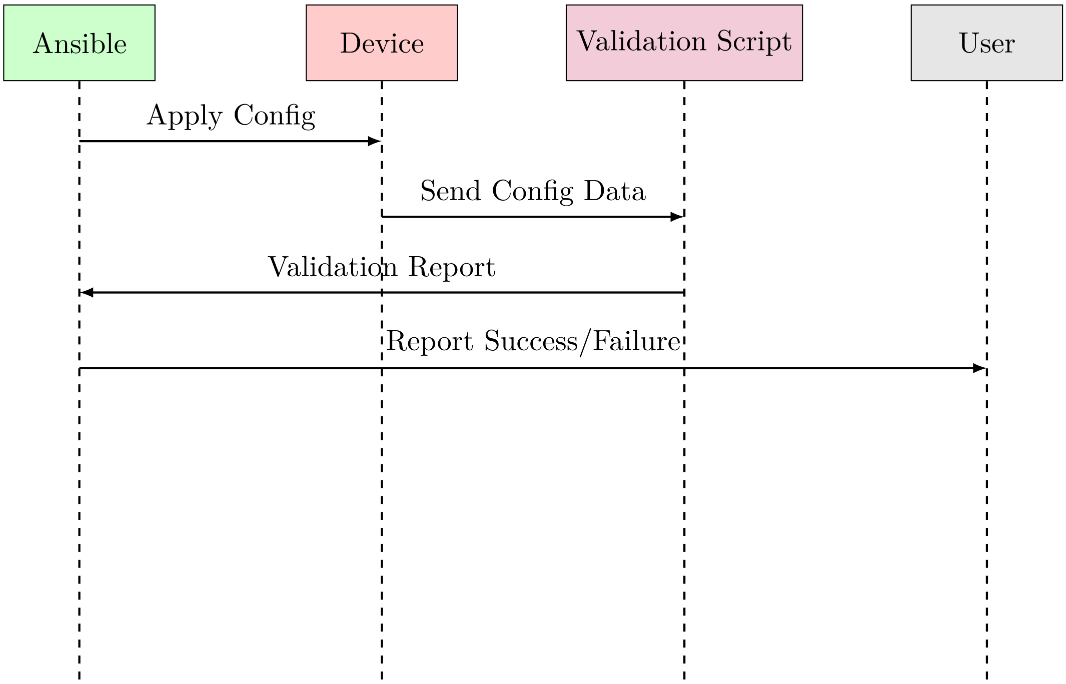

Figure 4.3. Sequence Diagram: Validation & Feedback

In the next paragraph it is shown in detail each working phase.

4.2.1 User Input Processing

The process starts when the user interacts with the chatbot interface. The chatbot leverages Natural Language Understanding (NLU) to interpret the user's intent and extract relevant configuration parameters.

Before proceeding with any configuration request, the chatbot requires the user to authenticate via *SSH credentials*. If the user fails to provide valid credentials, the process cannot continue. The chatbot explicitly prompts the user for a username and password, ensuring that access to network devices is secured before any configuration changes are made.

Once authentication is successfully completed, the chatbot continues *parsing* the user request. If it detects missing or incorrect values, it dynamically asks the user to provide the necessary details. For example, if an invalid IP address is provided, the chatbot immediately identifies the issue and requests a valid input before proceeding. The chatbot itself does not verify the feasibility of the requested configuration but ensures that the provided values adhere to syntactic correctness.

Once all required parameters are collected and validated, the chatbot *triggers* an AWS Lambda function to process the data. The Lambda function acts as an additional validation layer, guaranteeing that all necessary parameters are present and correctly formatted before proceeding with the next stage.

4.2.2 Inventory File Generation

After validating the input parameters, the Lambda function is responsible for generating the *inventory file*, a structured INI-style configuration file required for the automation process. The inventory file contains *essential connection details*, such as SSH credentials and backup paths, as well as a structured breakdown of the requested configurations. Instead of hardcoding vendor-specific information, the Lambda function organizes the data in a vendor-agnostic format, where each *network service*, such as VLAN, SNMP, and DHCP, is assigned to a dedicated section with *activation flags*.

Once the inventory file is generated, it is stored within an Amazon S3 bucket, ensuring that it remains accessible for the subsequent deployment phase. By centralizing the inventory storage, the system guarantees that configuration files are persistently available and can be referenced or reprocessed if needed.

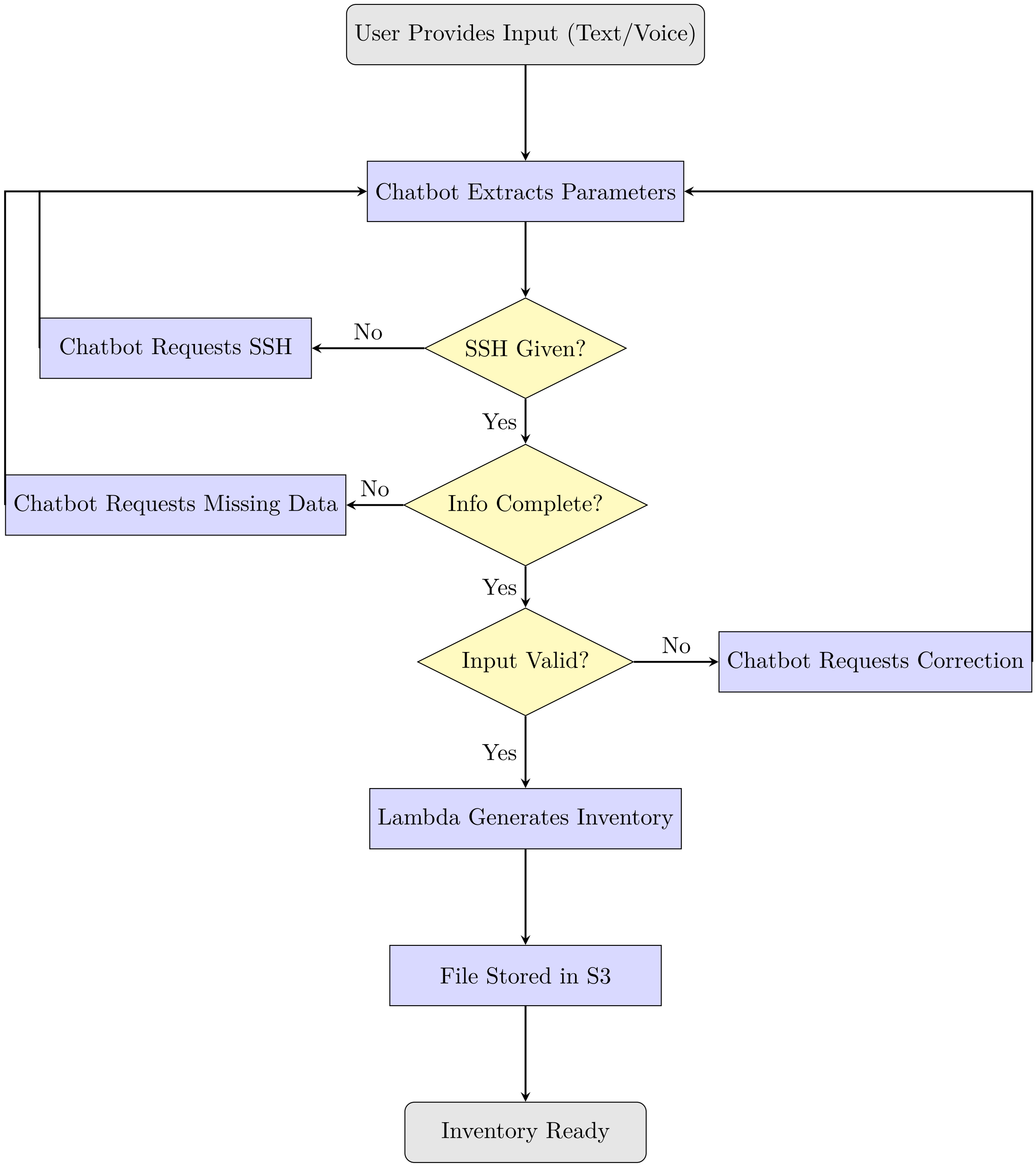

Figure 4.4. User Input Processing: Chatbot and Parameter Extraction

4.2.3 Automated Configuration Deployment

With the *inventory file* ready, Ansible retrieves the configuration and starts the *deployment phase*. The execution begins by parsing the inventory file to identify active services and extract the necessary parameters. Since the chatbot does not provide vendor-specific details, Ansible infers the device vendor dynamically by establishing an SSH connection and retrieving system information. Based on the detected vendor, Ansible selects the appropriate *playbooks* and executes the corresponding configuration tasks.

During deployment, Ansible applies the requested settings by executing the required commands on the network device. If a configuration step fails, the system immediately *stops* execution and *notifies* the user of the encountered error. In cases where a misconfiguration is detected, Ansible automatically rollbacks to the previously saved configuration, ensuring that the device remains in a stable and functional state.

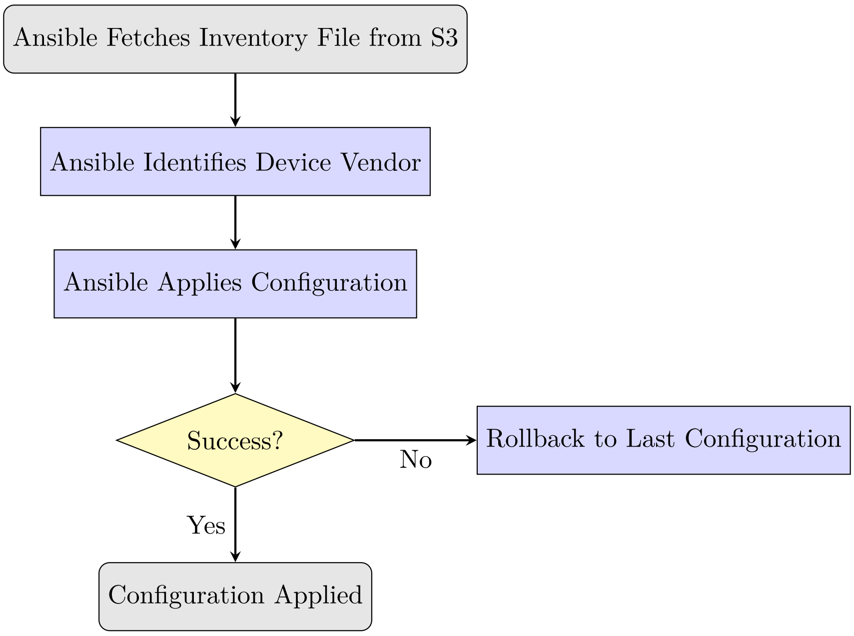

Figure 4.5. Ansible Deployment Workflow and Error Handling

4.2.4 Configuration Validation and Accuracy Check

Once Ansible completes the deployment, the system performs an additional verification step to ensure the accuracy of the applied configuration. A dedicated Python validation script connects to the configured network device via SSH and retrieves the current settings. Unlike Ansible, which only reports technical errors during deployment, this validation step directly compares the applied configuration with the original user request. If discrepancies are found, such as an unintended VLAN assignment or an incorrectly enabled port, the system generates a detailed report highlighting the inconsistencies.

This final validation ensures that the system does not merely execute configurations but actively enforces the intended network state. In this way the system allows users to confirm that their requests were processed as expected.

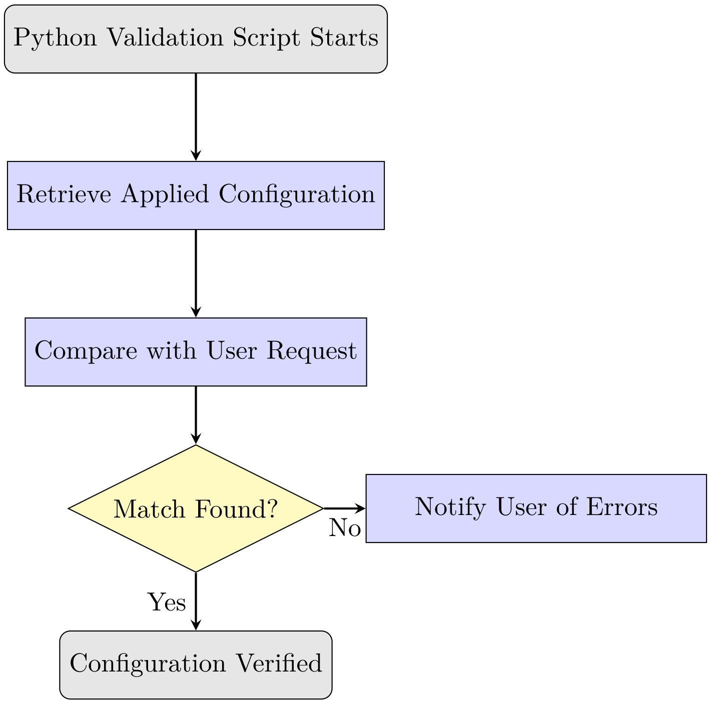

Figure 4.6. Post-Deployment Validation: Configuration Accuracy Check

4.3 Modularity and Scalability

The proposed system is designed with a *modular* and *scalable architecture* to ensure *flexibility* and *extensibility* in network automation. Its structure allows seamless integration of new functionalities, additional network vendors, and increased automation efficiency without requiring substantial modifications to the core components.

At the heart of this approach there is the *separation* of concerns among different modules. Each component of the system operates *independently*, following a structured workflow that minimizes dependencies between them. This modularity not only enhances maintainability but also facilitates future upgrades. For instance, if a new chatbot platform were to be introduced in place of Amazon Lex, the transition could be handled without affecting the inventory generation or configuration deployment processes. Similarly, adapting the system to support new network vendors does not require modifications to the chatbot or the inventory structure, as the deployment phase is dynamically managed by Ansible, which selects the appropriate configuration strategy based on the detected device type.

Scalability is another key consideration. The system is built to handle configurations for multiple network devices efficiently, ensuring that the complexity of the infrastructure does not impact performance. Leveraging Ansible's ability to execute tasks in parallel allows configurations to be applied across many devices simultaneously. Additionally, the architecture takes advantage of serverless computing, optimizing resource usage by executing processes only when necessary, reducing idle time and improving overall responsiveness.

Beyond scalability in terms of device management, the system is designed to accommodate future expansions in functionality. While the current implementation supports essential network automation tasks such as VLAN provisioning, SSH access management, and DHCP configurations, its structure allows for straightforward integration of additional network services, including advanced routing, firewall rule automation, and security policy enforcement. Since all configurations are defined dynamically, extending the range of supported operations primarily involves updating the chatbot's intent recognition model and expanding the Ansible playbooks accordingly.

The balance between modularity and scalability guarantees that the system remains adaptable to different operational contexts. Whether applied to a small-scale deployment or a complex enterprise network with heterogeneous devices, the automation framework remains efficient, flexible, and easily maintainable, providing a foundation for future enhancements without introducing significant architectural complexity.

Chapter 5

Implementation

Until now, the explanation was focused on a high-level description of the system, but in the section below there is a detailed report about how each component works.

5.1 Chatbot Development with Amazon Lex V2

The chatbot developed for this system is designed to allow users to interact with the network automation framework in natural language, specifying configuration requests that are then processed by backend services.

The chatbot is structured around three key components: *intents*, *slots*, and *utterances*. These fundamental elements define how the chatbot interprets user inputs and extracts relevant information.

5.1.1 Intent

An intent corresponds to a well-defined action, such as configuring a VLAN, enabling ports, or setting up DHCP services. It is a structured representation of a user request, guiding the chatbot on how to process and respond to interactions. [15]

The chatbot currently supports six main intents:

- *AuthenticateSSH* intent: Handles user authentication via SSH before any configuration action can be performed.
- *ConfigureVLAN* intent: Manages the creation and assignment of VLANs based on user specifications.
- *EnablePorts* intent: Allows users to enable or disable specific switch ports.
- *ConfigureSNMP* intent: Configures SNMP settings for network monitoring.

- *ConfigureDHCP* intent: Sets up DHCP server configurations on the network devices.
- *FallbackIntent*: Guides users when their request does not match any predefined intent.

Each intent includes one or more slots (parameters), which Lex gatheres before execution.

5.1.2 Slot

A slot is a parameter required to fulfill an intent. Each intent has a predefined set of slots that must be *filled* with user-provided values. Slots are placeholders that help to capture structured data from user input, allowing Amazon Lex to construct meaningful responses. [15]

For example, the *ConfigureVLAN* intent requires the following slots:

- VLANIPAddress: The IP address assigned to the VLAN.
- VLANName: The name of the VLAN.
- VLANPorts: The list of ports associated with the VLAN.
- VLANID: The numerical identifier of the VLAN.

If any of these slots are missing, the chatbot prompts the user for the necessary information before proceeding.

Intent	**Description**	**Required Slots**
`AuthenticateSSH`	Authenticates the user via SSH before configuring the device.	`SwitchIP`, `Username`, `Password`
`ConfigureVLAN`	Creates a VLAN on the device based on user input.	`VLANID`, `VLANName`, `VLANIPAddress`, `VLANPorts`
`ConfigureDHCP`	Configures DHCP with an IP range and VLAN.	`DHCPStartIP`, `DHCPEndIP`, `DHCPGateway`, `DHCPVlanName`, `DHCPPorts`
`EnablePorts`	Enables specific ports on the device.	`PortList`, `NumberOfPorts`
`ConfigureSNMP`	Sets up SNMP for monitoring.	`SNMPUsername`, `SNMPAuthPassword`, `SNMPGroup`

Table 5.1. Summary of chatbot intents and required slots

5.1.3 Utterance

An utterance is a phrase or sentence that a user may say to trigger an intent. Utterances are examples used to train Amazon Lex to recognize user requests and allow natural variations in speech and text input to be understood by the chatbot. [15]

Example utterances for the *ConfigureVLAN* intent are:

- Configure VLAN with IP address {*VLANIPAddress*}, name {*VLANName*}, ports {*VLANPorts*}, and ID {*VLANID*}.
- Set up a VLAN named {*VLANName*} with ID {*VLANID*} and assign ports {*VLANPorts*}.

When an utterance is detected, Amazon Lex maps it to the corresponding intent and extracts slots values from the user's input. If a user request does not match any intent, Lex triggers the FallbackIntent.

5.1.4 Handle a request

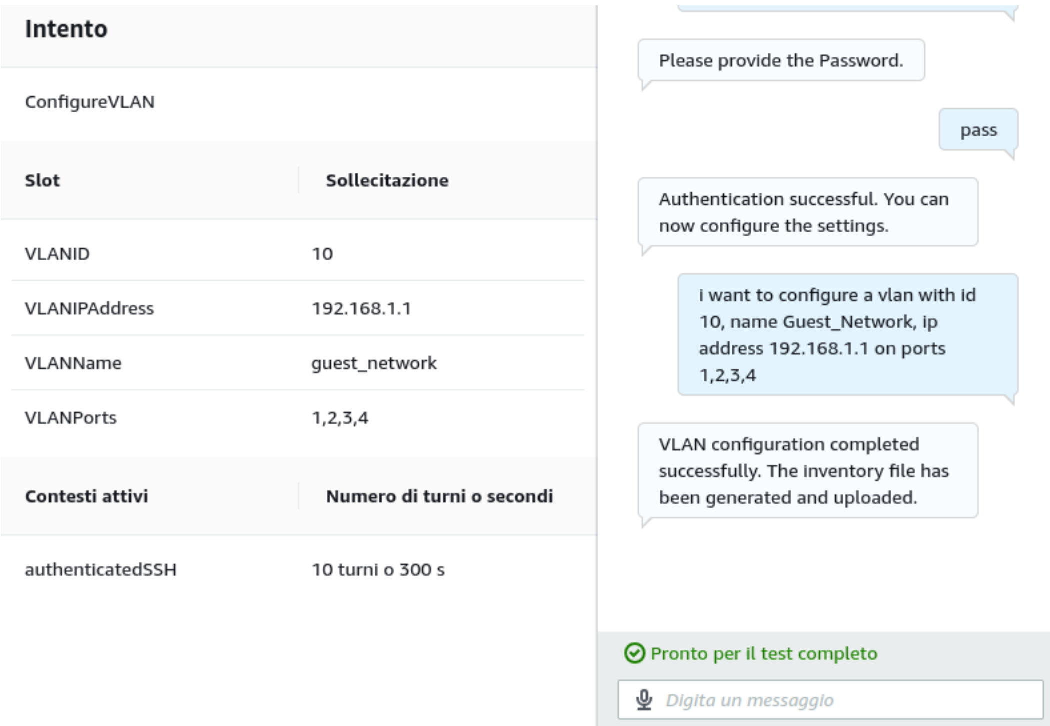

Figure 5.1. Vlan Configuration Request

When a user sends a message, AWS Lambda determines which intent was activated. The chatbot sends the request in the following format:

```json
{
  "sessionState": {
    "intent": {
      "name": "ConfigureVLAN",
      "slots": {
        "VLANID": {"value": {"interpretedValue": 10}},
        "VLANName": {"value": {"interpretedValue": "Guest_Network"}},
        "VLANIPAddress": {"value": {"interpretedValue": "192.168.1.1"}},
        "VLANPorts": {"value": {"interpretedValue": "1,2,3,4"}}
      }
    }
  }
}
```


The Lambda function extracts the intent name to determine the correct processing flow:

```python
def lambda_handler(event, context):
    try:
        # Extract the intent name from the event
        intent_name = event['sessionState']['intent']['name']

        # Handle the request based on the recognized intent
        if intent_name == "AuthenticateSSH":
            return handle_authenticate_ssh(event)
        elif intent_name == "ConfigureVLAN":
            return handle_configure_vlan(event)
        elif intent_name == "EnablePorts":
            return handle_enable_ports(event)
        elif intent_name == "ConfigureSNMP":
            return handle_configure_snmp(event)
        elif intent_name == "ConfigureDHCP":
            return handle_configure_dhcp(event)
        else:
            return handle_fallback(event)
    except Exception as e:
        return generate_error_response(intent_name, f"Error processing
        ↪ {intent_name}: {str(e)}")
```

Before executing a configuration, AWS Lambda verifies that all required slots are provided.

5.1.5 Verifying Slot Presence

Each intent defines mandatory slots, and if any slot is missing, Lex prompts the user to provide it. Before proceeding with VLAN configuration, the function checks if all required slots have been filled.

Each slot in `event['sessionState']['intent']['slots']` is a dictionary containing user-provided values and the verification is handled in the Lambda function, as shown in the example below:

```
def handle_configure_vlan(event):
    required_slots = ['VLANID', 'VLANName', 'VLANIPAddress', 'VLANPorts']

    slots = event['sessionState']['intent']['slots']
    missing_slots = [slot for slot in required_slots if not slots.get(slot) or not slots[slot].get('value')]

    if missing_slots:
        return elicit_slot(event, missing_slots[0], f"Please provide the {missing_slots[0]}.")

    # Proceed only if all slots are provided
    return process_vlan_configuration(event, slots)
```

The system iterates over all required slots and check if they exist in the request. The function `slots.get(slot)` verifies whether a slot is present, while `slots[slot].get('value')` ensures that the provided value is not empty. If either condition fails, the slot is considered missing, and the chatbot will prompt the user to enter the missing information before continuing. In this case if a user forgets to specify VLANPorts, the chatbot stops execution and asks: "*Please provide the VLAN ports.*"

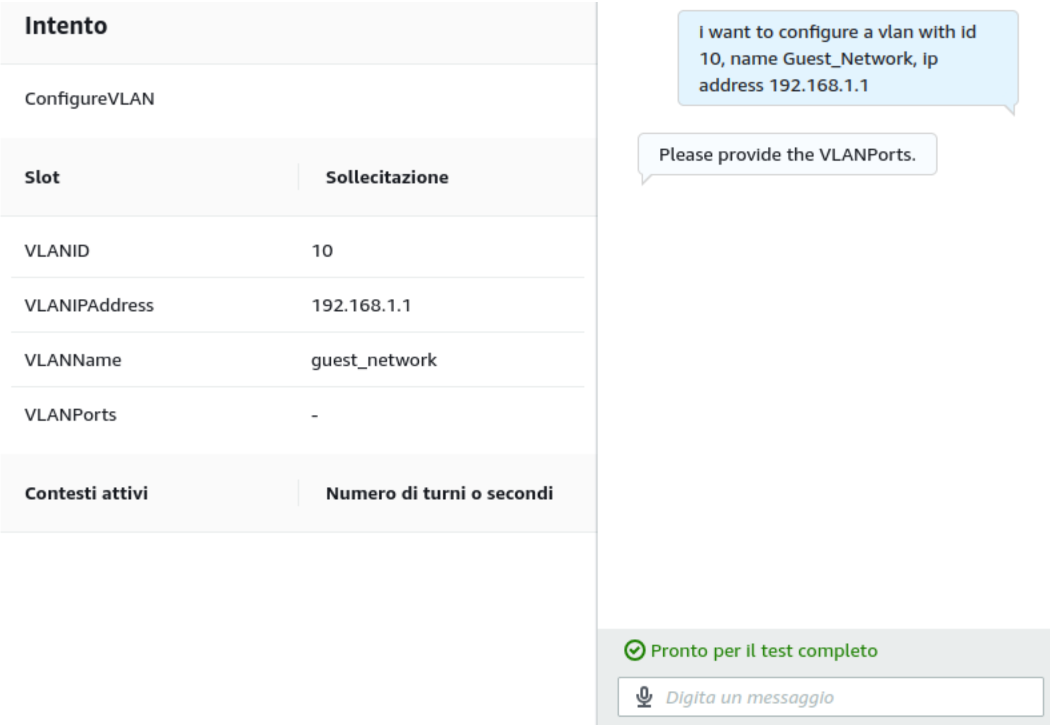

Figure 5.2. Vlan Configuration Request - Missing Slot

Once *all* required slot values are collected, the function calls `process_vlan_configuration(event, slots)`. This function extracts the collected parameters and prepares them for the next stage, where they will be processed into an actual network configuration. The details of how the configuration is stored and executed are covered in later sections.

5.1.6 Verifying Slot Validity

In addition to checking for missing slots, AWS Lambda validates the format of each slot's value.

For example, the VLANIPAddress slot must be a valid IPv4 address. Lambda uses a regular expression to guarantee the IP format is correct:

```python
import re

def is_valid_ip(ip):
    pattern = re.compile(r"^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$")

    if not pattern.match(ip):
        return False
```

```
    parts = ip.split('.')
    return all(0 <= int(part) <= 255 for part in parts)
```

If a user provides an invalid IP (e.g., 999.168.1.1), the chatbot rejects it and prompts the user: "*The provided IP address is invalid. Please enter a valid IPv4 address.*"

5.1.7 Session Attribute

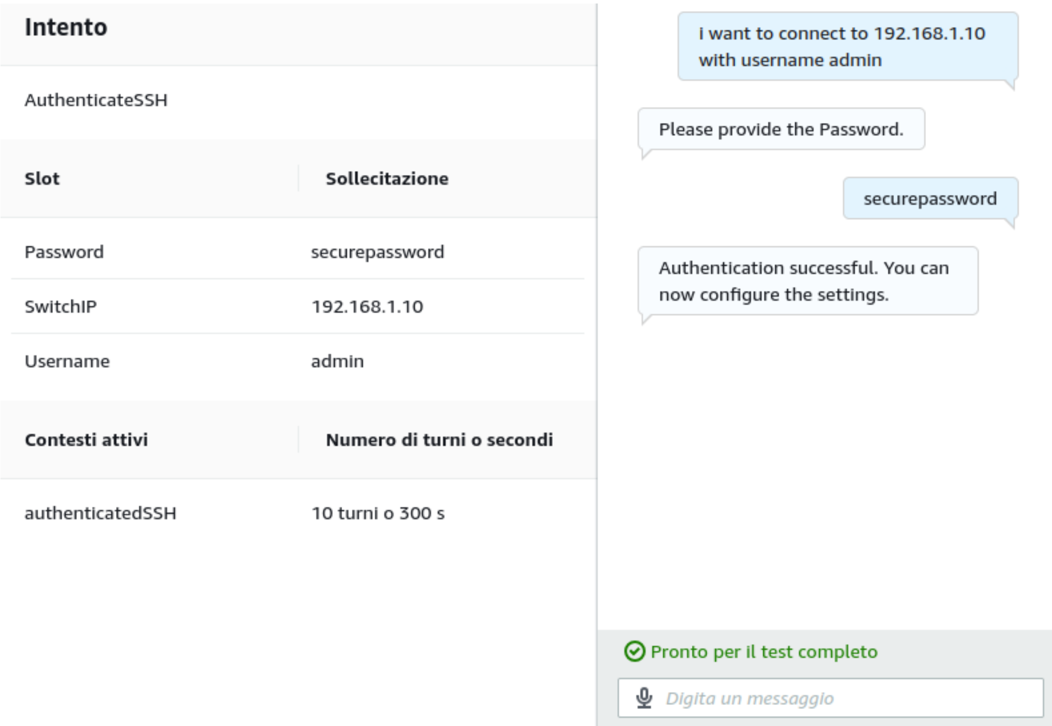

Figure 5.3. SSH Authentication Request

Once the chatbot has processed a user request, it maintains the context of authentication and configurations through session attributes. These attributes allow Lex to remember previous actions and maintain continuity between user interactions.

For example, after a successful authentication using the `AuthenticateSSH` intent, the chatbot stores the credentials in the session state:

```
{
    "sessionAttributes": {
        "authenticatedSSH": "true",
        "SwitchIP": "192.168.1.10",
```

```
        "Username": "admin",
        "Password": "securepassword"
    }
}
```

By leveraging session attributes, subsequent requests (e.g., VLAN configuration) do not require the user to re-authenticate. Instead, the chatbot can verify the presence of `"authenticatedSSH": "true"` and proceed directly to network configuration.

5.1.8 Fallback Intent

When a user provides an input that does not match any predefined intents, the chatbot automatically triggers the Fallback Intent, preventing incorrect responses or unexpected behaviors.

The Fallback Intent is the only intent that differs from the others. Unlike standard intents, it is not associated with a specific configuration action and does not require slots or utterances. Its primary function is to handle unrecognized input and guide the user toward a valid request.

The workflow is the following.

1. The chatbot receives an input that does not match any registered intent.
2. The *Fallback Intent* is automatically activated by Amazon Lex.
3. The chatbot analyzes the user's input to check if it can infer the requested service.
4. If the chatbot detects keywords related to an existing intent, it redirects the user to that intent and asks the missing slots.
5. If the chatbot cannot determine the user's intent, it provides help by listing the available configurable services.

This approach guarantees that the chatbot does not simply return a generic error message when it does not understand a command but instead actively helps the user to express a correct request. If a user provides a generic input such as: ”*I want to configure the network.*” since this request does not directly match any predefined intent, the chatbot triggers the Fallback Intent and responds with a help message.

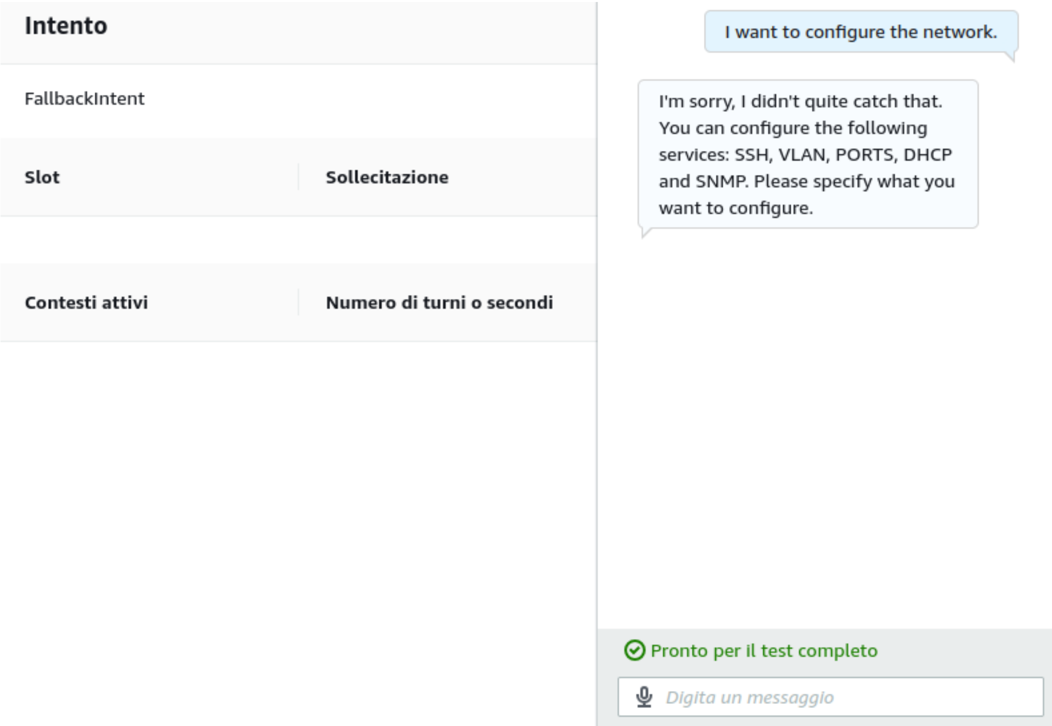

Figure 5.4. Misunderstood Request

However, if the user enters: "*I need to set up a VLAN.*" the chatbot recognizes that the term VLAN is associated with the ConfigureVLAN intent and guides the user to provide the required parameters.

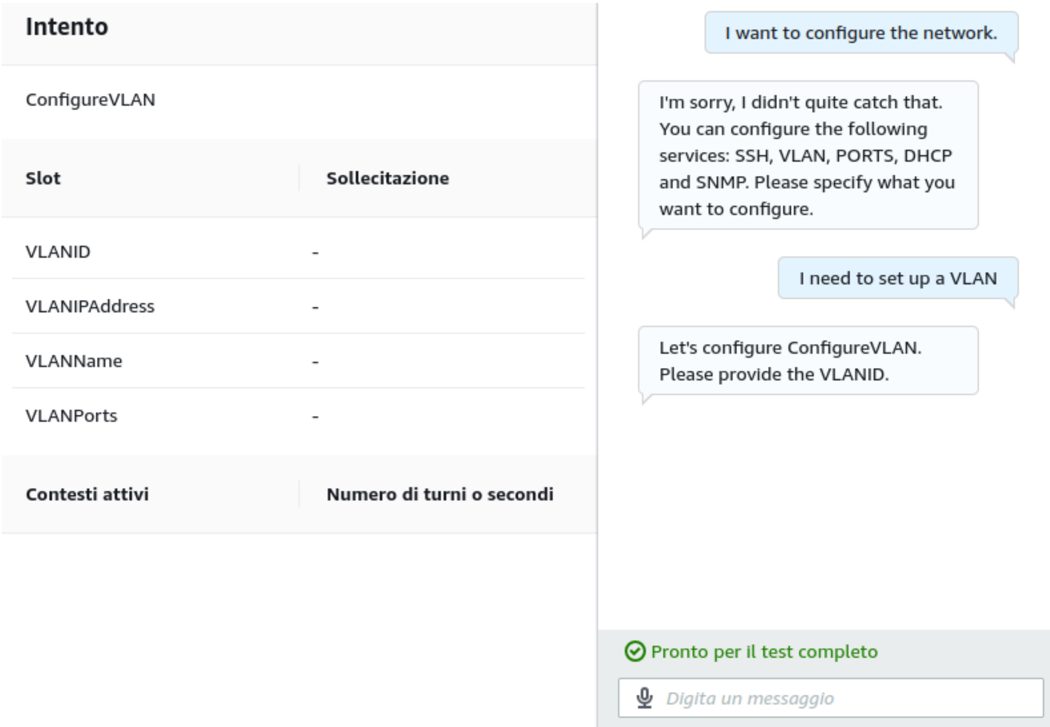

Figure 5.5. Vlan Request After Fallback Intent

The Fallback Intent is handled in the AWS Lambda function through the `handle_fallback function`. This function analyzes the user's input and attempts to infer the requested service based on keywords present in the text:

```
def handle_fallback(event, slots, session_attributes, invocation_source):
    user_input = event['inputTranscript'].lower()

    if "ssh" in user_input:
        return elicit_slot_for_intent(event, "AuthenticateSSH",
        ↪  ["SwitchIP", "Username", "Password"])
    elif "vlan" in user_input:
        return elicit_slot_for_intent(event, "ConfigureVLAN", ["VLANID",
        ↪  "VLANName", "VLANIPAddress", "VLANPorts"])
    elif "dhcp" in user_input:
        return elicit_slot_for_intent(event, "ConfigureDHCP",
        ↪  ["DHCPStartIP", "DHCPEndIP", "DHCPGateway", "DHCPVlanName",
        ↪  "DHCPPorts"])
    elif "ports" in user_input or "enable ports" in user_input:
        return elicit_slot_for_intent(event, "EnablePorts", ["PortList",
        ↪  "NumberOfPorts"])
    elif "snmp" in user_input:
```

```
        return elicit_slot_for_intent(event, "ConfigureSNMP",
        ↪  ["SNMPUsername", "SNMPAuthPassword", "SNMPGroup"])
    else:
        return generate_fallback_response()
```

If the chatbot detects a clear intent based on user-provided keywords, it directly calls `elicit_slot_for_intent`, which runs the correct intent and prompts any missing slots.

However, if the user's input is too vague or does not contain recognizable keywords, the chatbot calls `generate_fallback_response()`, which returns a more general help message.

5.2 Inventory File Generation and Management

In an Ansible-based network automation system, the *inventory file* acts as a central configuration point, defining which network devices to manage and how to connect to them. This file is *dynamically* generated based on user requests captured by the chatbot.

Amazon Lex and AWS Lambda work together to construct this file by enabling only the requested services, ensuring that unnecessary configurations are not applied.

5.2.1 Inventory File Structure

The inventory file is structured into *multiple sections*, each corresponding to a specific network configuration.

```
[all:vars]
# ansible_connection=ansible.netcommon.network_cli

[switches:children]
exos1

###########################################################
#
# Configuration Parameters for EXOSVM32.1.1.6-1
#
###########################################################

[exos1]
###########################################################
```

```
# Hosts list
exos_host1 ansible_ssh_host={switch_ip}

[exos1:vars]
### Do not change this section
ansible_network_os=community.network.exos
ansible_user='{username}'
ansible_password='{password}'

ansible_ssh_user={username}
ansible_ssh_pass={password}

############################################################
# Save config vars
local_file_cfg="./backup_cfg/{{ ansible_ssh_host }}.cfg"
remote_file_cfg="/usr/local/cfg/primary.cfg"

local_file_applied_cfg="{{ local_file_cfg }}.new"

############################################################
# VLAN configuration
vlan_set=off
vlan_delete=off

############################################################
# DHCP configuration
dhcp_server=off
dhcp_del=off

############################################################
# SNMP configuration
snmp=off
snmp_del=off

############################################################
# Ports configuration
enable_ports=off
disable_ports=off
```

Below is a breakdown of these sections:

1. `[all:vars]` – This section contains global variables that apply to all devices within the inventory. It stores SSH credentials, defines the Ansible connection method, and specifies the operating system type of the managed devices.

2. `[switches:children]` – This section defines the groups of devices that will be configured. The chatbot dynamically assigns the requested switch or device to this group based on user input.

3. `[exos1]` (or other device groups) – This section lists individual network devices that belong to the defined group. Each device is assigned a unique entry, and its IP address is set according to the user-provided input.

4. `[exos1:vars]` – In this part, device-specific variables are stored, including authentication credentials such as the username and password, which are collected via the chatbot.

5. Configuration sections – These sections define the settings for VLAN, SNMP, DHCP, and Ports. At first, all services are disabled, meaning their respective flags are set to off. However, as the user requests specific configurations, the chatbot, through AWS Lambda, dynamically updates the inventory file to enable only the required services while leaving the others unchanged.

This approach ensures that no preconfigured settings are applied to the network devices until explicitly requested. Only the necessary sections will be modified in later steps, allowing precise customization and avoiding unintended changes.

In the next paragraph we will examine how AWS Lambda dynamically modifies the inventory file in response to user requests, guaranteeing that only the necessary configurations are applied.

5.2.2 Dynamic Inventory Update Logic

Every time the user requests a service (e.g., VLAN configuration), AWS Lambda updates the inventory file, ensuring that the flag for the requested service is set to *on*, the corresponding section is populated with the user provided values and unrequested sections remain unchanged (i.e., stay disabled). This guarantees that the system applies only the necessary configurations, avoiding unnecessary changes to the network devices.

Each user request follows a structured process:

1. The chatbot identifies the intent (e.g., ConfigureVLAN).

2. It verifies that all required slots are present (e.g., VLAN ID, name, ports, IP).

3. AWS Lambda modifies the inventory file, enabling the correct flag and inserting the required configuration.

4. The updated file is stored, ensuring the correct configurations are later applied.

For example, if a user requests to configure a VLAN, before the lambda update the vlan section in the inventory file is:

```
vlan_set=off
vlan_delete=off
```

and after:

```
vlan_set=on
vlan_id=10
vlan_name="Guest_Network"
vlan_ip_addr="192.168.1.1"
vlan_ports=1,2,3,4
vlan_delete=off
```

This dynamic update process ensures that the configuration remains customized, flexible and efficient, avoiding unnecessary settings.

5.2.3 Inventory File Generation in AWS Lambda

The generation process follows a structured approach. At first, the inventory file is created with a predefined template that includes global settings, authentication parameters, and placeholders for device configurations. Lambda updates *only* the relevant sections, leaving the other parts unchanged. Before applying these modifications, the function verifies that all necessary parameters have been provided to prevent incomplete or invalid configurations. Finally, the updated inventory file is formatted and ready to storage.

Initializing the Inventory File Structure

The inventory file begins with a default structure that establishes the authentication details and network device groups. This template remains *unchanged* throughout the process, ensuring a consistent format for all configurations. The initial structure is defined as follows:

```
HEADER_TEMPLATE = """[all:vars]
ansible_ssh_user={username}
ansible_ssh_pass={password}

[switches:children]
exos1
```

```
[exos1]
exos_host1 ansible_ssh_host={switch_ip}

[exos1:vars]
ansible_network_os=community.network.exos
ansible_user='{username}'
ansible_password='{password}'
"""
```

At this stage, the authentication credentials such as the username, password, and switch IP address are retrieved from the chatbot session attributes. These values are inserted into the template before any additional configurations are applied.

Although the base structure is always present, all service sections remain disabled until explicitly requested by the user.

Updating Inventory Sections Based on User Requests

Once the chatbot determines which configuration the user wants to modify, AWS Lambda updates the corresponding section of the inventory file while preserving the existing structure. Each configuration block, such as VLAN settings, is stored in a dedicated section, and only the requested sections are activated.

To implement this, a function is in charge for retrieving the *current state* of the inventory file, modifying the requested section, and guaranteeing that all required components remain untouched. The function operates as follows:

```python
def update_inventory_section(section_name, new_content,
↪  session_attributes):
    header, sections =
     ↪  parse_inventory(session_attributes['inventory_content'])

    # Enable the requested section by setting its flag to 'on'
    sections[section_name] = new_content.splitlines()

    ensure_default_sections(sections)
    session_attributes['inventory_content'] = serialize_inventory(header,
     ↪  sections)
```

Generating VLAN Configuration

When a user requests VLAN configuration, the Lambda function extracts the provided slot values and formats them into a structured configuration block. The function responsible for this conversion takes the VLAN ID, name, IP address, and associated ports and generates the corresponding inventory entry:

```
def generate_vlan_content(vlan_id, vlan_name, vlan_ip, vlan_ports_list):
    vlan_ports = ','.join(vlan_ports_list)
    return f"""
vlan_set=on
vlan_ip_addr='{vlan_ip}'
vlan_id={vlan_id}
vlan_name='{vlan_name}'
vlan_ports={vlan_ports}
vlan_delete=off
"""
```

Serializing and Writing the Inventory File

Once all necessary modifications have been made, the inventory file is formatted and stored in memory before being saved to AWS S3. This process guarantees that the final file complies with the correct structure and includes only the relevant configurations. The formatting function builds the final inventory file as follows:

```
def serialize_inventory(header, sections):
    content = [header]
    for section, lines in sections.items():
        if lines:
            content.append(f"\n#{section}")
            content.extend(lines)
    return "\n".join(content)
```

At this stage, the whole file is reconstructed by combining the template with all the enabled configuration sections. This guarantees that the final inventory file is correctly structured and ready for use. The final step in the pipeline implies securely storing the inventory file and making it readily accessible. AWS S3 is used for this purpose, as outlined in the next section.

5.2.4 Storing in AWS S3

In this system, the inventory file is uploaded to an AWS S3 Bucket, where it remains available for the network automation framework. The following code performs this task:

```
import boto3

s3 = boto3.client('s3')
S3_BUCKET_NAME = 'my-inventory-bucket'

def store_inventory_file(session_attributes):
    inventory_content = session_attributes.get('inventory_content', '')

    if not inventory_content:
        raise ValueError("Inventory content is empty. Cannot store the
        ↪  file.")

    s3.put_object(
        Bucket=S3_BUCKET_NAME,
        Key="inventory.ini",
        Body=inventory_content.encode('utf-8')
    )

    print("Inventory file successfully uploaded to S3.")
```

The function first retrieves the latest version of the inventory file stored in the chatbot session attributes. If the content is missing or empty, an exception is raised to prevent uploading an invalid file. After validating the content, the `put_object()` method writes the file to the S3 bucket under the name inventory.ini.

By calling this function at the end of each configuration step, the system ensures that the most recent version of the inventory file is always available in S3.

Whenever a new configuration is requested, such as VLAN creation, SNMP setup, or DHCP configuration, the inventory file is modified to reflect the latest user input. After updating the content, the modified file is uploaded again to S3 to replace the previous version. The following snippet shows how this process is integrated within the VLAN configuration function:

```
def handle_configure_vlan(event, slots, session_attributes,
↪   invocation_source, active_contexts):
    try:
        vlan_id = slots['VLANID']['value']['interpretedValue']
        vlan_name = slots['VLANName']['value']['interpretedValue']
        vlan_ip = slots['VLANIPAddress']['value']['interpretedValue']
        vlan_ports = slots['VLANPorts']['value']['interpretedValue']

        vlan_ports_list = [port.strip() for port in vlan_ports.split(',')]
```

```
        vlan_content = generate_vlan_content(vlan_id, vlan_name, vlan_ip,
        ↪  vlan_ports_list)

        # Update inventory content
        header, sections =
        ↪  parse_inventory(session_attributes['inventory_content'])

        # Remove existing VLAN section if present
        if 'vlan' in sections:
            del sections['vlan']

        # Add new VLAN configuration
        sections['vlan'] = vlan_content.splitlines()

        ensure_default_sections(sections)

        # Serialize and update session attributes
        session_attributes['inventory_content'] =
        ↪  serialize_inventory(header, sections)

        # Upload the updated inventory to S3
        store_inventory_file(session_attributes)

        return {
            "sessionState": {
                "dialogAction": {"type": "Close"},
                "intent": {"name": "ConfigureVLAN", "state": "Fulfilled"},
                "sessionAttributes": session_attributes
            },
            "messages": [{"contentType": "PlainText", "content": "VLAN
            ↪  configuration saved and uploaded."}]
        }
    except Exception as e:
        return generate_error_response("ConfigureVLAN", f"Error in
        ↪  handle_configure_vlan: {str(e)}")
```

In this function, the chatbot:

1. Extracts VLAN parameters from the slots filled by the user.
2. Generates the VLAN configuration content using the `generate_vlan_content()` function.
3. Parses the current inventory file and removes any previous VLAN configuration.

4. Adds the new VLAN section and guarantees that the file structure remains consistent.

5. Serializes the updated inventory and saves it in the chatbot's session attributes.

6. Calls `store_inventory_file(session_attributes)`, which uploads the modified file to S3.

By following this approach, the inventory file is updated with each user interaction, ensuring that the network automation framework always operates with the latest configurations.

Once stored in S3, the inventory file needs to be retrieved for execution by the network automation system. The following function downloads the latest version of the inventory file when needed:

```
def retrieve_inventory_file():
    response = s3.get_object(Bucket=S3_BUCKET_NAME, Key="inventory.ini")
    inventory_content = response['Body'].read().decode('utf-8')
    return inventory_content
```

This section focused on how the chatbot dynamically generates, updates, and stores the inventory file. Using AWS Lambda and S3, the system maintains an update configuration that ensures seamless network automation. The next section will show how these configurations are applied and executed within the network environment.

5.3 Automated Configuration with Ansible

Ansible works as the automation engine responsible for applying the configurations dynamically defined in the inventory file. The `main` playbook acts as the orchestrator, guaranteeing that only the necessary configurations are executed based on user requests.

Unlike the chatbot, which processes one request at a time, Ansible can execute configurations across multiple network devices *simultaneously*. This capability significantly enhances scalability, making it a powerful solution for both small-scale and large-scale network automation.

The main playbook follows a structured approach:

- It retrieves information from the inventory file generated earlier.

- It determines the network device vendor dynamically at runtime to apply the correct configurations.
- It activates and executes only the necessary configurations, based on the flags set in the inventory.
- It guarantees that configurations are applied *only* if needed, avoiding redundant changes.
- If multiple devices are present, they are all configured in a *single* execution.

In this section, we will explore the *structure* and *execution flow* of the main playbook, the mechanisms used for backing up and restoring configurations, and how individual service playbooks are applied to devices.

5.3.1 Main Playbook

The main playbook is the core component of Ansible's automation process and it follows a structured workflow that ensures each configuration is applied correctly.

Ansible establishes an SSH connection to the target devices using the credentials defined in the inventory file, so it is able to interact directly with it, executing commands remotely just as a network administrator. Since each playbook follows this same approach, this technology guarantees a consistent and agentless automation process, applying configurations across multiple devices in a structured and efficient manner.

Detecting Device Vendor

Before executing any configuration, Ansible needs to *determine* the vendor of the target network device, in order to apply the appropriate commands. It knows it through the following step:

```yaml
- name: Determine switch vendor
  raw: show version
  register: switch_version
```

The `show version` command is executed on the switch, it retrieves the vendor information, and the output is stored in the variable `switch_version`. This value is later used to apply vendor-specific configurations.

Evaluating Inventory Flags

The playbook then reads the *inventory file* generated by Lambda function, where each service (VLAN, SNMP, DHCP, etc.) has a corresponding *on/off flag*. If a flag is set to *on*, the respective configuration will be applied, otherwise it will be skipped. Initially, all flags are set to *off*:

```
vlan_set=off
```

If the user requests VLAN configuration, the chatbot updates this section:

```
vlan_set=on
vlan_id=10
vlan_name="Guest_Network"
vlan_ip_addr="192.168.1.1"
vlan_ports=1,2,3,4
```

Selectively Executing Configuration Tasks

The main playbook is designed to conditionally execute playbooks based on the *on/off flags* in the inventory file. The `when` condition ensures that only requested configurations are applied. For example, VLAN creation is executed only if `vlan_set` is *on* :

```
- name: Create VLAN
  include_tasks: ./vlan/create_vlan.yml
  when: vlan_set | bool
```

This logic is the same for all the services, guaranteeing that no unnecessary configurations are applied, keeping the automation process efficient.

Handling Multiple Devices in a Single Execution

A key advantage of Ansible over the chatbot is its ability to configure multiple devices *simultaneously*. If the inventory file contains multiple switches:

```
[switches]
switch1 ansible_ssh_host=192.168.1.10
switch2 ansible_ssh_host=192.168.1.11
switch3 ansible_ssh_host=192.168.1.12
```

Ansible will execute the configurations on all devices in a single execution. This allows efficient scalability, it can manage hundreds of devices in a single run.

Ensuring Configurations Are Applied Correctly

At the end of execution, the playbook ensures that all configurations are saved permanently on the network device. This *prevents* the loss of changes in case of a reboot.

```yaml
- name: Apply Configuration
  include_tasks: ./save_cfg/save_configuration.yml
```

Full Ansible Main Playbook

The following is the complete *main playbook*, including all the discussed elements described above:

```yaml
---
- name: "Network Automation R&D Project [rev 0.1.0] - Start process"
  hosts: switches
  gather_facts: no
  vars:
    save_initial_conf: false
  tasks:
    - block:
      # Step 1: Detect Network Device Vendor
      - name: Determine switch vendor
        raw: show version
        register: switch_version

      ###########################################
      # Step 2: Save Initial Configuration
      - name: Save configuration
        include_tasks: ./save_cfg/save_configuration.yml
      - name: Set save_initial_conf variable to true
        set_fact:
          save_initial_conf: true

      ###########################################
      # Step 3: Apply Configurations Based on Inventory Flags
      - name: Create VLAN
        include_tasks: ./vlan/create_vlan.yml
        when: vlan_set | bool
```

```yaml
      - name: Delete VLAN
        include_tasks: ./vlan/delete_vlan.yml
        when: vlan_delete | bool

      - name: Enable Ports
        include_tasks: ./ports/enable_ports.yml
        when: enable_ports | bool
      - name: Disable Ports
        include_tasks: ./ports/disable_ports.yml
        when: disable_ports | bool

      - name: Create SNMP
        include_tasks: ./snmp/create_snmp.yml
        when: snmp | bool
      - name: Delete SNMP
        include_tasks: ./snmp/delete_snmp.yml
        when: snmp_del | bool

      - name: Enable DHCP Server
        include_tasks: ./dhcp_server/enable_dhcp_server.yml
        when: dhcp_server | bool
      - name: Disable DHCP Server
        include_tasks: ./dhcp_server/disable_dhcp_server.yml
        when: dhcp_del | bool

      ##########################################
      # Step 4: Save Configurations on Device
      - name: Apply Configuration
        include_tasks: ./save_cfg/save_configuration.yml

      ##########################################
      # Step 5: Error Handling - Restore Configuration if Failure Occurs
      rescue:
        - name: Restore Previous Configuration
          include_tasks: ./load_cfg/load_configuration.yml
        - name: Reboot Device After Restore
          include_tasks: ./reboot/reboot_device.yml
...
```

5.3.2 Configuration Backup & Error Handling

Ensuring configuration reliability is a crucial aspect of network automation. Before applying any changes, the system must securely store the existing configuration to prevent potential disruptions. If an error occurs during execution, the playbook should be able to restore the previous configuration, guaranteeing network stability.

This is achieved through a structured approach.

1. Save the current configuration before applying changes.

2. Apply requested configurations from the inventory file.

3. If an error occurs, automatically restore the previous configuration.

In this section, the focus is on *ExtremeXOS* devices.

Creating a Configuration Backup

To prevent data loss, the first step in this process is to create a backup of actual configuration of the switch. This is done using the `save configuration primary` command, ensuring that the current state of the device is stored persistently before any modifications are applied. The following Ansible task does this:

```
- name: Save current configuration on ExtremeXOS switches
  raw: save configuration primary
  when: "'ExtremeXOS' in switch_version.stdout"
```

Saving the configuration directly on the switch is not enough. To ensure external backup storage, Ansible retrieves the configuration file and copies it to the local machine using *SCP* (Secure Copy Protocol). This process guarantees that even if the switch crashes, the configuration can be restored from a safe location.

```
- name: "Copy ExtremeXOS configuration from remote to local (initial)"
  ansible.builtin.shell:
    cmd: "sshpass -p {{ansible_ssh_pass}} scp
     ↪  {{ansible_ssh_user}}@{{ansible_ssh_host}}:{{remote_file_cfg}}
     ↪  ./{{local_file_cfg}}"
  delegate_to: localhost
  when: "'ExtremeXOS' in switch_version.stdout and save_initial_conf ==
   ↪  false"
```

This approach ensures that the most recent successful configuration is always available for rollback, minimizing the impact of failures.

Applying Configurations Safely

With a backup in place, Ansible proceeds to apply the requested configurations only if they are explicitly enabled in the *inventory file*. This mechanism guarantees that no unnecessary modifications are introduced into the system.

Despite this, network automation involves complex and dynamic environments, making impossible to avoid failures. If an error occurs during configuration deployment, Ansible must detect it and automatically revert to the last working state to prevent service disruptions.

Restoring Configuration After an Error

In the event of a failure, the playbook must restore the backup created in Step 1. This is done by copying the previously stored configuration file back to the switch:

```
- name: "Copy configuration back to switch (restore from backup)"
  ansible.builtin.shell:
    cmd: "sshpass -p {{ansible_ssh_pass}} scp {{local_file_cfg}}
    ↪  {{ansible_ssh_user}}@{{ansible_ssh_host}}:{{remote_file_cfg}}"
  delegate_to: localhost
  when: "'ExtremeXOS' in switch_version.stdout"
```

This ensures that all unwanted changes are discarded, and the switch returns to a previously known working state. In some cases, a reboot is required to fully reload the restored configuration and ensure that all network services are functioning correctly.

```
- name: "Reboot ExtremeXOS switch after restoring configuration"
  raw: reboot
  when: "'ExtremeXOS' in switch_version.stdout"
```

Rebooting the switch forces it to reload the restored settings, effectively reverting the system to its previous stable state. This step is particularly important for configuration changes that require a full system reload to take effect, such as VLAN modifications or DHCP adjustments.

This structured backup and recovery mechanism ensures that every change applied to ExtremeXOS devices is fully reversible, providing a fail-safe against misconfigurations or unexpected errors.

- All configurations are backed up before execution.

- Changes are only applied if explicitly requested.
- Errors trigger an automatic rollback to prevent failures.
- Rebooting ensures configurations are properly restored.

With the backup and error-handling strategy in place, the next step is to see how Ansible executes the requested network configurations, ensuring seamless automation.

5.3.3 Application of Service Configurations

Ansible dynamically applies user-requested configurations to network devices. Each network service has a dedicated playbook that guarantees the correct configuration is applied based on the vendor-specific requirements.

Ansible determines the appropriate configuration steps by first identifying the vendor of the network device, as previously discussed in Section 5.3.1. Once the vendor is known, the system executes the corresponding service-specific playbook, ensuring that the correct syntax and command structure are used for each device.

Among the various service configurations automated by Ansible, *VLAN configuration* is as a representative example of how different network devices are handled dynamically: the following playbook illustrates how Ansible applies VLAN settings based on the detected vendor. Once the configuration flag in the inventory file is set to *on*, the main playbook triggers the service tasks. The corresponding playbook executes vendor-specific commands in according to the detected switch. Below is the structure of the VLAN playbook, which applies the necessary configurations depending on the vendor:

```
- name: Create VLAN and assign ports on Cisco switch
  raw: |
    configure terminal
    vlan {{ vlan_id }}
    name {{ vlan_name }}
    {% for port in cisco_ports %}
    interface {{ port }}
    switchport mode access
    switchport access vlan {{ vlan_id }}
    {% endfor %}
  when: "'Cisco' in switch_version.stdout"

- name: Create VLAN and assign ports on Extreme Networks switch
  raw: |
```

```
    create vlan {{ vlan_name }} tag {{ vlan_id }}
    configure vlan {{ vlan_name }} add ports {{ vlan_ports | join(',') }}
    configure vlan {{ vlan_name }} ipaddress {{ vlan_ip_addr }}/24
  when: "'ExtremeXOS' in switch_version.stdout"
```

This playbook dynamically applies the correct VLAN configuration based on the brand detected in the `switch_version` variable: it operates in a structured manner to ensure accurate configuration across different vendors. The playbook first checks the `switch_version` output to confirm the vendor, then extracts VLAN parameters (ID, name, ports, and IP) from the inventory file and translates them into vendor-specific commands. If the device runs Cisco IOS, the playbook follows the Cisco VLAN configuration syntax, while if it runs ExtremeXOS, the corresponding Extreme Networks commands are applied. The system iterates over the list of ports provided in the inventory file, ensuring each port is properly configured to belong to the requested VLAN.

Once the VLAN configuration is applied, the final step is to guarantees that changes stay across device reboots. This is achieved through a configuration save command:

```
- name: Save VLAN Configuration
  raw: save configuration
  when: "'ExtremeXOS' in switch_version.stdout"
```

By saving the configuration, the system ensures that all applied VLAN settings remain intact, even if the device is restarted. This structured playbook approach allows Ansible to dynamically configure VLANs across different vendors efficiently. The same methodology applies to other network services such as SNMP and DHCP, where vendor-specific adaptations are required to ensure correct execution.

At this stage, the final step in the automation workflow is the execution of a Python verification script, which validates the correctness of the applied configurations. This verification process ensures that the configured network matches the user's intent, and it will be discussed in the next section.

5.4 Validation with Python Scripts

Once Ansible has completed its automation tasks, ensuring that the applied configurations match the user's request is a crucial step in the process. Ansible provides feedback only if an error occurs during execution, meaning that a configuration might be successfully applied even if the actual parameters differ from what was specified in the inventory file. This is where the Python verification script comes into play.

The verification script acts as an *additional* layer of validation, checking if the network devices have been configured as wanted. It retrieves the current configuration from the switches and compares it against the expected values stored in the inventory file. Any discrepancies are flagged, allowing administrators to quickly identify and correct potential misconfigurations.

Just like the Ansible playbooks, the verification script follows a structured approach based on the *on/off flags* in the *inventory file*:

- If a service (e.g., VLAN configuration) was requested (`on` in the inventory), the corresponding verification function is executed.
- If a service was not requested (`off` in the inventory), the script skips the check, optimizing performance and avoiding unnecessary processing.
- The verification script does not make any changes to the device configurations. It only reports inconsistencies, providing detailed information on any detected errors.

Since different vendors require different commands to retrieve device configurations, the script is designed to execute vendor-specific queries. In this section, we will focus on the implementation for *ExtremeXOS* devices, illustrating how the script validates VLAN configurations.

The verification process is structured into two main components:

1. A centralized execution script, which manages the verification logic, determines which services need to be validated, and triggers the appropriate checks based on the inventory flags.
2. Service-specific validation scripts, which extract and analyze the relevant configuration data from the network device, checking for inconsistencies with the expected settings.

In the following sections, we will explore these components in detail, starting with the centralized execution logic and then focusing on how individual services, such as VLAN configuration, are verified.

5.4.1 Centralized Execution Logic

To ensure that network configurations have been correctly applied, the verification script is executed *immediately* after Ansible completes its tasks.

The verification process is managed by a centralized Python script, which orchestrates the execution of individual service-specific verification scripts. Similar to how the main Ansible playbook operates, the *centralized verification script* reads the inventory file, checks which services have been enabled (i.e., have their flags set to `on`), and runs only the relevant verification scripts. This guarantees that the system only validates configurations that were explicitly requested, optimizing performance and avoiding unnecessary operations.

The verification script is exclusively an *analytical tool*, guaranteeing that the applied configurations match the expected parameters. It does not modify the device but simply retrieves, compares, and reports inconsistencies, providing a clear diagnostic *report* for administrators.

The Python script that implements this logic first loads the inventory file and extracts the relevant information:

```python
import yaml

INVENTORY_FILE = "inventory.ini"

def load_inventory():
    """Reads the inventory file and extracts configuration flags."""
    with open(INVENTORY_FILE, "r") as file:
        inventory_data = file.read()

    inventory = {}
    for line in inventory_data.split("\n"):
        if "=" in line:
            key, value = line.split("=")
            inventory[key.strip()] = value.strip()

    return inventory

inventory = load_inventory()
```

This function parses the inventory file and stores its contents in a dictionary, allowing the script to dynamically determine which services require validation.

The script then determines which verification tasks to execute:

```python
def run_verification():
    """Executes verification scripts based on enabled services."""

    if inventory.get("vlan_set") == "on":
```

```
        print("[INFO] Running VLAN verification...")
        verify_vlan()

    if inventory.get("snmp") == "on":
        print("[INFO] Running SNMP verification...")
        verify_snmp()

    if inventory.get("dhcp_server") == "on":
        print("[INFO] Running DHCP verification...")
        verify_dhcp()

run_verification()
```

The script checks whether each service has been enabled (`on`) in the inventory file. If it happens, it calls the corresponding verification function for that service. The functions `verify_vlan()`, `verify_snmp()`, and `verify_dhcp()` handle the validation for VLAN, SNMP, and DHCP, respectively.

Since the verification logic is centralized, only the required checks are executed, making the process both *efficient* and *scalable*.

The next section will focus on the detailed execution of a specific verification function to show how the script retrieves and validates the network configuration against the expected values.

5.4.2 Verification of Individual Services

Each network service requires a specific validation process to guarantee that the applied configurations match those defined in the inventory file.

Each verification script follows a structured process: it loads expected values from the inventory, connects to the device via SSH, retrieves the actual configuration, and compares the results. If discrepancies are found, the script generates a detailed error report, enabling administrators to intervene before issues escalate.

This approach enhances automation reliability by detecting inconsistencies in real-time, ensuring that configurations match the intended settings.

To illustrate how this process works, we analyze VLAN verification as an example. The VLAN validation script begins by reading the inventory file, retrieving values such as VLAN ID, name, and assigned IP address; once the expected values are obtained, the script connects to the network device via SSH to retrieve the current VLAN configuration. This is achieved by executing the show vlan command, which provides the list of VLANs currently configured on the switch.

```
import paramiko
```

```python
def get_current_vlan_config(ip, user, password):
    """Connects to the switch via SSH and retrieves VLAN configuration."""

    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    try:
        client.connect(ip, username=user, password=password)
        stdin, stdout, stderr = client.exec_command('show vlan')
        output = stdout.read().decode('utf-8')
        return parse_vlan_output(output)

    except Exception as e:
        print(f"[ERROR] Connection failed: {e}")
    finally:
        client.close()
```

The retrieved data is initially in raw text format. To make it usable, the script extracts key values such as VLAN ID, name, and assigned IP address using regular expressions:

```python
import re

def parse_vlan_output(output):
    """Parses VLAN configuration output and extracts relevant details."""

    vlan_info = []
    lines = output.splitlines()

    for line in lines:
        match = re.search(r'(\S+)\s+(\d+)\s+.*?(\S+)?', line)
        if match:
            vlan_entry = {
                'name': match.group(1),
                'id': match.group(2),
                'ip': match.group(3) if match.group(3) != '---' else None
            }
            vlan_info.append(vlan_entry)

    return vlan_info
```

At this stage, the script has the expected VLAN configuration (from the inventory) and the actual VLAN configuration retrieved from the device. The final step involves comparing these values and reporting discrepancies.

```
from colorama import Fore

def check_vlan(inventory_file, debug=False):
    """Validates VLAN configuration against expected inventory values."""

    inventory = read_inventory(inventory_file, debug)
    current_vlan_configs = get_current_vlan_config(inventory['ip'],
    ↪  inventory['user'], inventory['password'])

    expected_name = inventory.get('vlan_name')
    expected_id = inventory.get('vlan_id')
    expected_ip = inventory.get('vlan_ip')

    print("\n" + "="*50)
    print("VLAN CHECK: VLAN Details")
    print(f"  Expected VLAN Name : {expected_name}")
    print(f"  Expected VLAN ID   : {expected_id}")
    print(f"  Expected VLAN IP   : {expected_ip}")
    print("="*50 + "\n")

    current_vlan = next((vlan for vlan in current_vlan_configs if
    ↪  vlan['name'] == expected_name and vlan['id'] == expected_id), None)

    if current_vlan:
        if current_vlan.get('ip') == expected_ip:
            print(Fore.GREEN + f"CORRECT: The VLAN '{expected_name}' (ID:
            ↪  {expected_id}) is correctly configured with IP
            ↪  {expected_ip}.")
        else:
            print(Fore.RED + f"NOT CORRECT: The VLAN '{expected_name}' (ID:
            ↪  {expected_id}) has an incorrect IP address.")
    else:
        print(Fore.RED + f"NOT CORRECT: The VLAN '{expected_name}' (ID:
        ↪  {expected_id}) is not configured on the device.")
```

This structured approach applies to all network services. For example, DHCP verification follows the same methodology: extracting expected DHCP parameters from the inventory, retrieving the current DHCP settings from the switch, parsing the output, and reporting mismatches. The same principle applies to SNMP and port configurations.

By keeping the logic modular and reusable, the verification framework ensures that any network service can be validated with minimal code duplication. The only variations are in the CLI commands used to retrieve configurations and the parsing rules applied to interpret the output.

In summary, the use of Python scripts for configuration validation provides an additional layer of control compared to Ansible's standard execution feedback.

The ability to detect discrepancies between the intended and actual configurations is crucial for keeping a consistent and error-free network infrastructure. By identifying potential misconfigurations early, administrators can quickly intervene and ensure network stability.

In the next chapter, we will discuss scalability and extensibility considerations, analyzing how this approach can be adapted to larger deployments and evolving network architectures.

5.5 Scalability and Extensibility Considerations

The implemented system combines multiple components (the chatbot, AWS Lambda functions, Ansible playbooks, and Python verification scripts) each contributing to the automation workflow. Ensuring *scalability* (handling larger networks efficiently) and *extensibility* (adapting to new requirements and services) is essential for the long-term viability of the solution.

This section examines how the system can scale to support larger environments and how it can be extended to integrate additional features. The analysis includes practical considerations on modifying the logic of the chatbot, optimizing Lambda executions, expanding inventory management handled by Ansible, and adding support for new network services and vendors in verification scripts.

5.5.1 Scalability Analysis

Scalability determines how well the system handles increasing workloads, such as a higher number of network devices or more frequent configuration requests. Each component in the automation pipeline plays a role in keeping efficiency at scale:

- *Chatbot* (Amazon Lex): The chatbot is designed to handle one request per device, ensuring that each configuration is processed sequentially. Since the user must specify all parameters for each device, the system was *intentionally* structured in this way. However, as the chatbot executes multiple requests, each new execution appends a new device to the inventory file. This allows

Ansible to process multiple configurations in a single run, even though the chatbot handles them one at a time.

- *AWS Lambda*: Each Lambda function executes the logic to update the inventory file and trigger Ansible. While this approach is lightweight, it introduces execution overhead when handling many devices sequentially. A parallel execution model using AWS Step Functions or Lambda concurrency features could be implemented to process multiple device configurations simultaneously.

- *Ansible*: Unlike the chatbot, Ansible already supports parallel execution of configurations across multiple devices, leveraging its inventory-based execution. To further optimize large-scale deployments, inventory file structuring could be improved by dynamically grouping devices based on criteria such as vendor type or location, reducing unnecessary task execution and improving efficiency.

- *Python Verification Scripts*: Since each verification script connects to the network device via SSH, executing them for multiple devices can introduce latency. A scalable approach would involve executing the verification scripts asynchronously.

The system can handle hundreds or thousands of devices with minimal performance degradation, ensuring that automation remains efficient even in large-scale network environments.

5.5.2 Extensibility Analysis

Extensibility refers to the case with which new functionalities, device types, or network services can be integrated into the automation framework. The modular design of the system allows for simple extensions in several key areas:

- *Chatbot* (Amazon Lex & Lambda Functions): To support new network services, new intents and slot types can be added to Lex. Lambda functions must then be extended to interpret these new requests and update the inventory accordingly. A structured intent-to-configuration mapping in Lambda can facilitate adding new functionalities without modifying existing code.

- *Ansible Playbooks*: New network services (e.g., firewall rules, QoS settings) can be added by creating service specific playbooks, following the existing model of conditionally executing tasks based on inventory flags. This requires defining the corresponding variables in the inventory and ensuring that the main playbook correctly invokes the new playbooks when needed.

- *Python Verification Scripts*: Each service-specific validation follows the same pattern: extracting expected values from the inventory, retrieving actual configurations via SSH, and comparing results. To add new verification functions only requires to define additional parsing logic and ensuring that the centralized execution script includes the new checks when appropriate.
- *Multi-Vendor Support*: The system is currently designed to handle specific vendors (e.g., ExtremeXOS and Cisco). In order to extend support to additional vendors, it is needed to map vendor-specific CLI commands in both Ansible playbooks and Python verification scripts. Since vendor detection is already integrated into the main playbook, to add a new vendor would involve to expand the existing conditional execution logic and to define the appropriate command syntax for that platform. By maintaining a consistent structure for inventory variables and verification logic, integrating a new vendor becomes an easy process.

By keeping a modular and loosely coupled architecture, the system can be extended with minimal disruption to existing functionality. These enhancements guarantee adaptability to follow network requirements while maintaining efficiency and reliability.

The scalability and extensibility of the automation system are key to its long-term success. By optimizing execution flow and structuring configurations efficiently, the system can scale to manage complex networks. Meanwhile, the modular architecture allows for seamless integration of new features, ensuring the framework remains adaptable to future networking needs.

Additionally, the system is designed with vendor flexibility in mind. The structured approach to CLI command execution allows new vendor support to be added efficiently, making the system more versatile in heterogeneous network environments.

With these considerations, the next chapter will focus on real-world use cases and performance evaluations, analyzing how the system behaves in different network scenarios.

Chapter 6

Use Cases and Results

6.1 Testing Scenarios and Goals

To guarantee that the developed automation framework performs as expected is a crucial step in validating its *effectiveness* and *reliability.* The primary objective of the testing phase is to verify that the system correctly configures network devices based on user requests and that it can dynamically handle different configurations while maintaining robustness and flexibility. The tests cover the complete workflow.

Each test is designed to address a specific use case, allowing us to evaluate different aspects of the system. The key goals of the testing phase are:

1. *Correctness*: Guarantee that configurations match the user's intent and are successfully applied to network devices.

2. *Scalability*: Validate how the system performs when configuring multiple devices simultaneously.

3. *Robustness*: Assess the ability of the system to handle incorrect inputs or incomplete configurations.

4. *Automation Consistency*: Verify that the full workflow operates without requiring manual intervention.

All tests are conducted on ExtremeXOS virtual switches simulated in GNS3, with the assumption that the same approach can be extended to other vendors. The validation phase ensures that configurations are not only successfully applied but also accurately match the expected settings.

In the following sections, we detail the specific test cases and their results, demonstrating the full process from user request to final network validation.

6.2 Use Case 1: Simple Vlan Configuration

In this first test scenario, it is shown the ability of the automation framework to correctly configure a VLAN and enable communication between two hosts in a simple network topology. The objective is to verify that the VLAN is successfully created, the required ports are enabled, and devices within the VLAN can communicate properly.

Network Topology

The network consists of a single ExtremeXOS switch with two PCs connected to it. Initially, without VLAN configuration, the two PCs belong to different network segments and are unable to communicate.

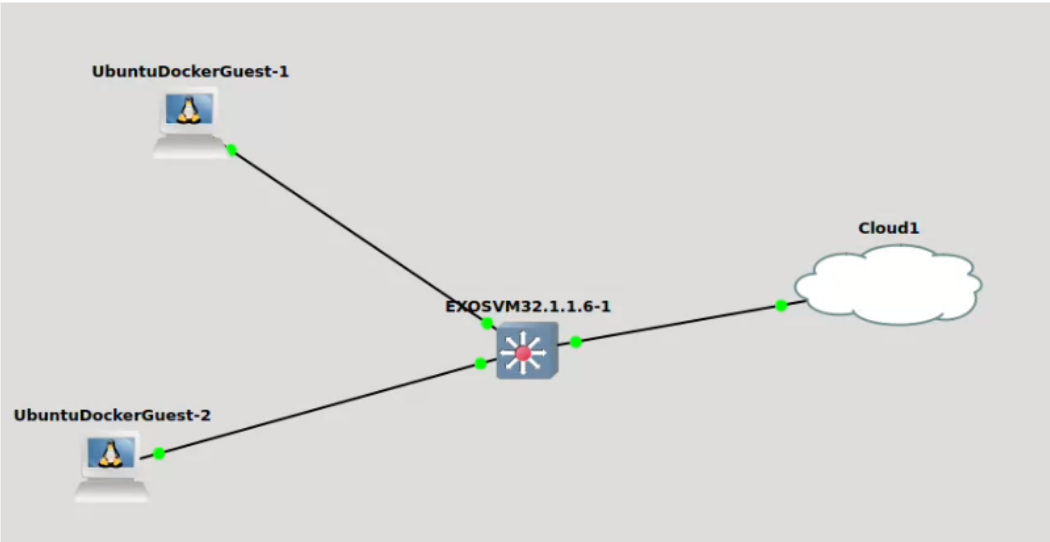

Figure 6.1. Network Topology: 1 switch & 2 PCs

Configuration Process

The configuration begins with a user request, which starts the process of VLAN creation. The chatbot collects the required parameters:

- VLAN ID
- VLAN Name
- IP Address
- Ports to be assigned to the VLAN

Below is an example interaction with the chatbot:

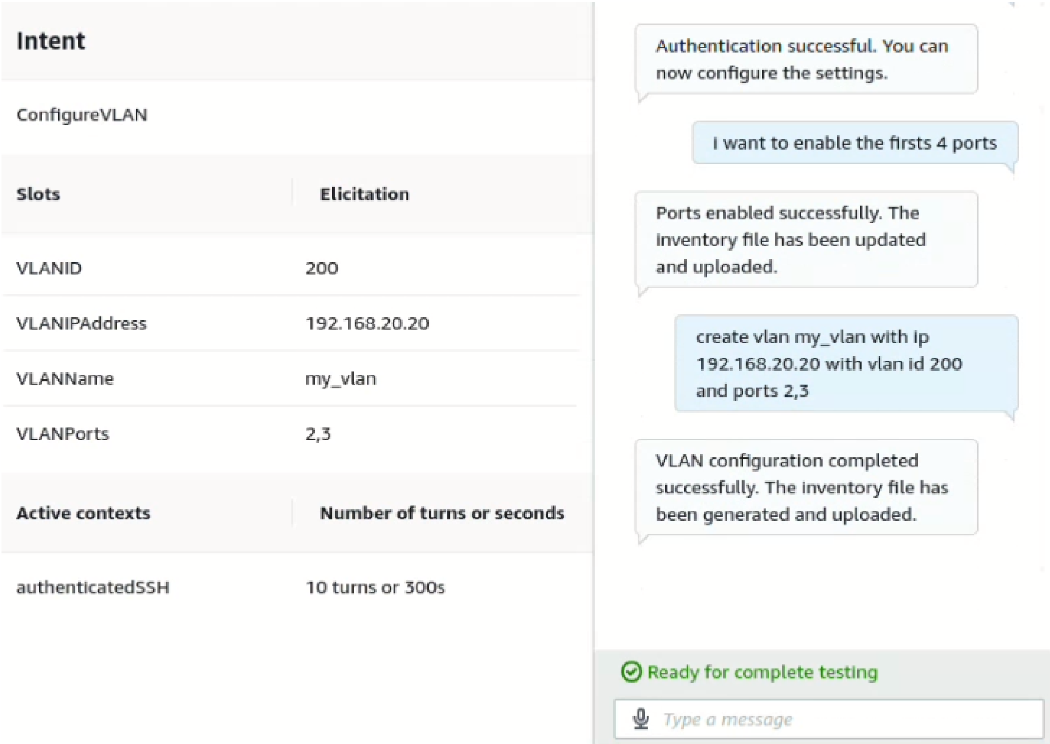

Figure 6.2. User VLAN Request

Once the chatbot collects all necessary inputs, the system updates the inventory file and stores it in AWS S3. The relevant section of the generated inventory file is as follows:

```
############################################################
# ports
enable_ports=on
enable_nports=1,2,3,4
disable_ports=off

############################################################
# dhcp server
dhcp_server=off
dhcp_del=off

############################################################
# snmp
snmp=off
snmp_del=off

############################################################
# vlan
vlan_set=on
vlan_ip_addr='192.168.20.20'
vlan_id=200
vlan_name='my_vlan'
vlan_ports=2,3
```

Figure 6.3. `Inventory` file updated by the chatbot and stored in AWS S3

Execution of Ansible Playbook & Validation

With the updated inventory, Ansible is triggered to execute the VLAN configuration playbook. The playbook dynamically applies the correct commands based on the vendor detected. Ansible confirms the successful execution of the playbook, ensuring that the configuration has been applied to the switch.

To prove that the VLAN has been correctly configured, the Python validation script retrieves the current VLAN settings from the switch and checks them with the expected values from the inventory file.

```
============================================================
                    STARTING CHECK_VLAN
============================================================


==================================================
🔍 VLAN CHECK: VLAN Details
  Expected VLAN Name : my_vlan
  Expected VLAN ID   : 200
  Expected VLAN IP   : 192.168.20.20
==================================================

CORRECT: The VLAN 'my_vlan' (ID: 200) is correctly configured with IP 192.168.20.20.

Current VLAN Configuration:
--------------------------------------------------
  VLAN Name : data
  VLAN ID   : 100
  VLAN IP   : 192.168.122.10
--------------------------------------------------
  VLAN Name : Default
  VLAN ID   : 1
  VLAN IP   : ----------------------------------------------
--------------------------------------------------
  VLAN Name : Mgmt
  VLAN ID   : 4095
  VLAN IP   : ----------------------------------------------
--------------------------------------------------
  VLAN Name : my_vlan
  VLAN ID   : 200
  VLAN IP   : 192.168.20.20
--------------------------------------------------
  VLAN Name : :
  VLAN ID   : 4
  VLAN IP   : None
--------------------------------------------------

✔ CHECK_VLAN completed without discrepancies.

============================================================
                    STARTING CHECK_PORTS
============================================================


==================================================
🔍 PORT CHECK: Enabled Port Details
  Expected enabled ports: [1, 2, 3, 4]
==================================================

CORRECT: All expected ports ([1, 2, 3, 4]) are correctly enabled.

Current enabled port configuration:
--------------------------------------------------
  Currently enabled ports: [1, 2, 3, 4]
```

Figure 6.4. Output Verification Script

As seen in Figure 6.4, the verification script confirms that all VLAN parameters match the requested configuration, guaranteeing that the automation process was completed successfully.

Connectivity Verification

Before the VLAN configuration, the two PCs were unable to communicate:

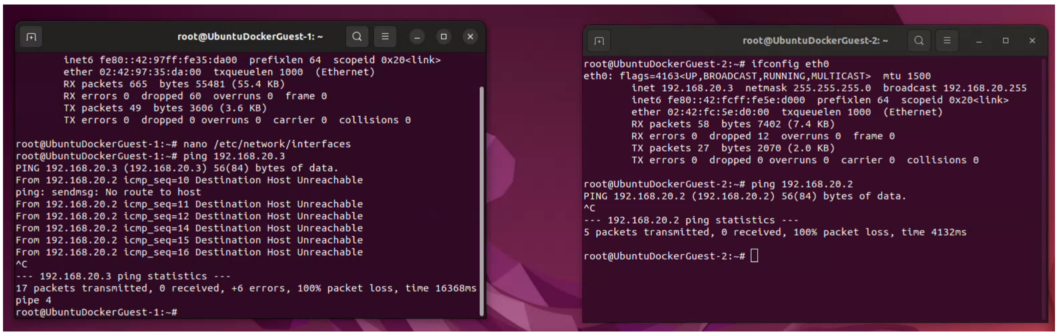

Figure 6.5. Ping test before configuration

After the VLAN has been created and the ports assigned, the two PCs can successfully ping each other:

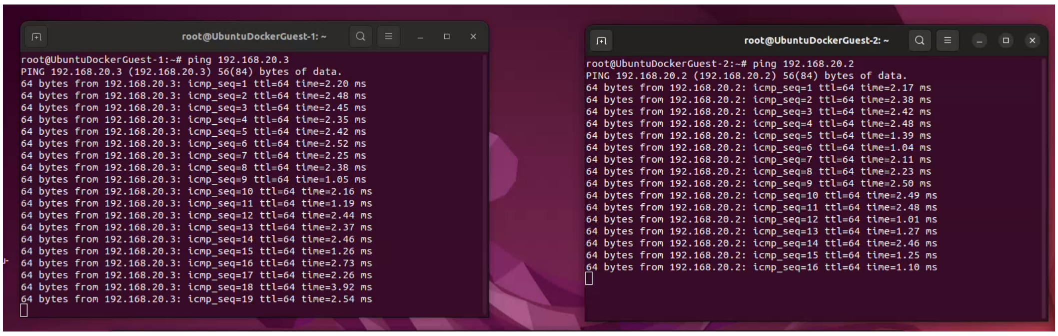

Figure 6.6. Ping test after configuration

This test confirms that the system can correctly interpret user input, dynamically configure the network, and validate the applied settings. The automated workflow successfully integrates all components, from chatbot interaction to final validation, proving the effectiveness of the framework.

6.3 Use Case 2: Dynamic Addressing with DHCP

This test evaluates the capability of the system to dynamically configure a DHCP server on a network switch, allowing connected hosts to automatically obtain IP addresses.

Network Topology

The test environment consists of an ExtremeXOS switch with four PCs connected. Initially, these devices have no manually assigned IP addresses and are unable to communicate. The goal is to configure the switch so that clients can dynamically receive IP addresses within the specified range.

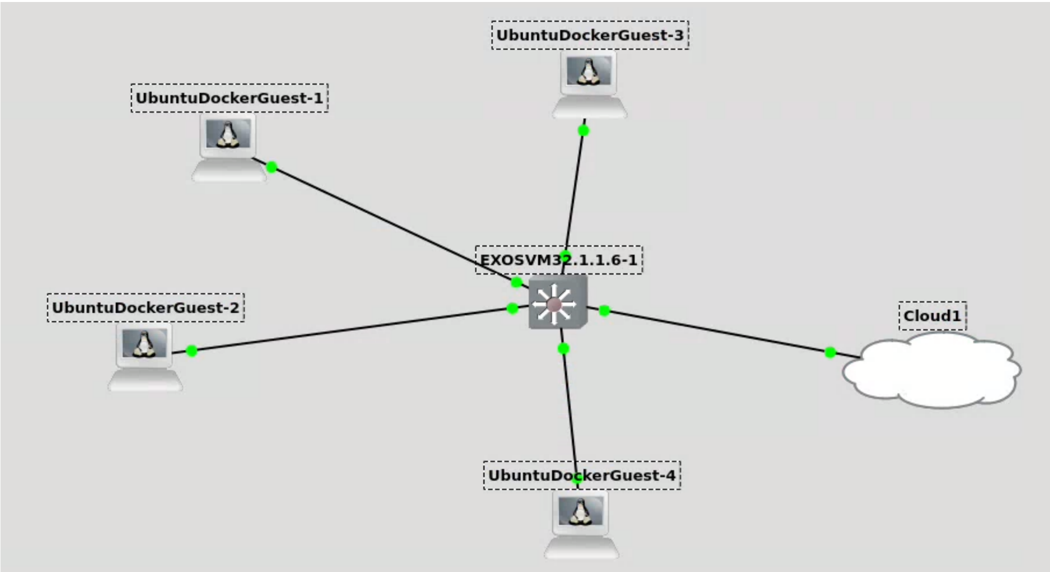

Figure 6.7. Network Topology: Switch with DHCP Server and 4 Clients

Configuration Process

The configuration process starts when the user talk with the chatbot, specifying the following parameters:

- VLAN Name: The VLAN where DHCP should be enabled.
- IP Address Range: The start and end addresses of the DHCP pool.
- Default Gateway: The gateway to be assigned to clients.
- Lease Time: The duration for which an IP address is leased.
- Ports: The switch interfaces where DHCP should be enabled.

The chatbot interaction proceeds as follows:

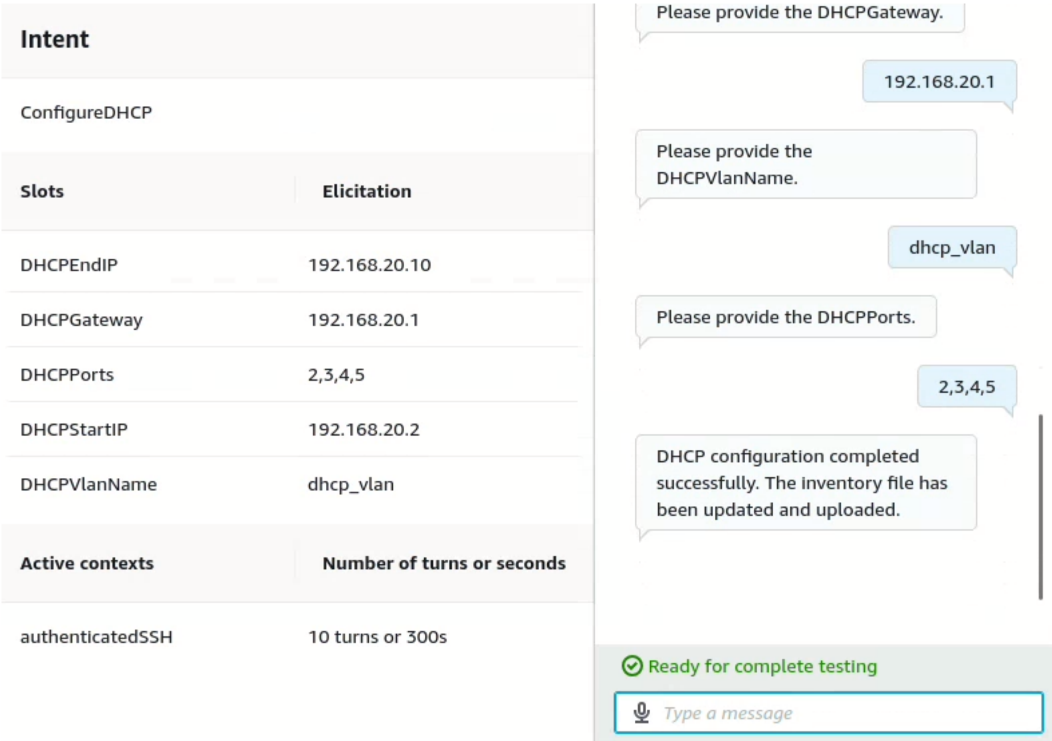

Figure 6.8. User Request for DHCP Configuration

After receiving the parameters, the system updates the inventory file and stores it in AWS S3 and the DHCP section in the inventory appears as follows:

```
###########################################################
# ports
enable_ports=on
enable_nports=1,2,3,4,5
disable_ports=off

###########################################################
# snmp
snmp=off
snmp_del=off

###########################################################
# vlan
vlan_set=on
vlan_ip_addr='192.168.20.20'
vlan_id=200
vlan_name='dhcp_vlan'
vlan_ports=2,3,4,5
vlan_delete=off

###########################################################
# dhcp server
dhcp_server=on
dhcp_start_addr_ip='192.168.20.2'
dhcp_end_addr_ip='192.168.20.10'
dhcp_addr_gateway='192.168.20.1'
dhcp_vlan_name='dhcp_vlan'
dhcp_ports=2,3,4,5
dhcp_lease_time=10
dhcp_del=off
```

Figure 6.9. Generated `Inventory.ini` with DHCP Configuration

Execution of Ansible Playbook & Validation

Ansible processes the updated inventory and applies the DHCP configuration on the switch. The DHCP playbook ensures that:

- The DHCP server is enabled on the correct VLAN.
- The IP address range, gateway, and lease time are correctly set.
- The assigned switch ports have DHCP enabled.

The Python validation script verifies that the applied DHCP settings match the requested parameters by retrieving the switch's configuration and comparing it with the expected values from the inventory file.

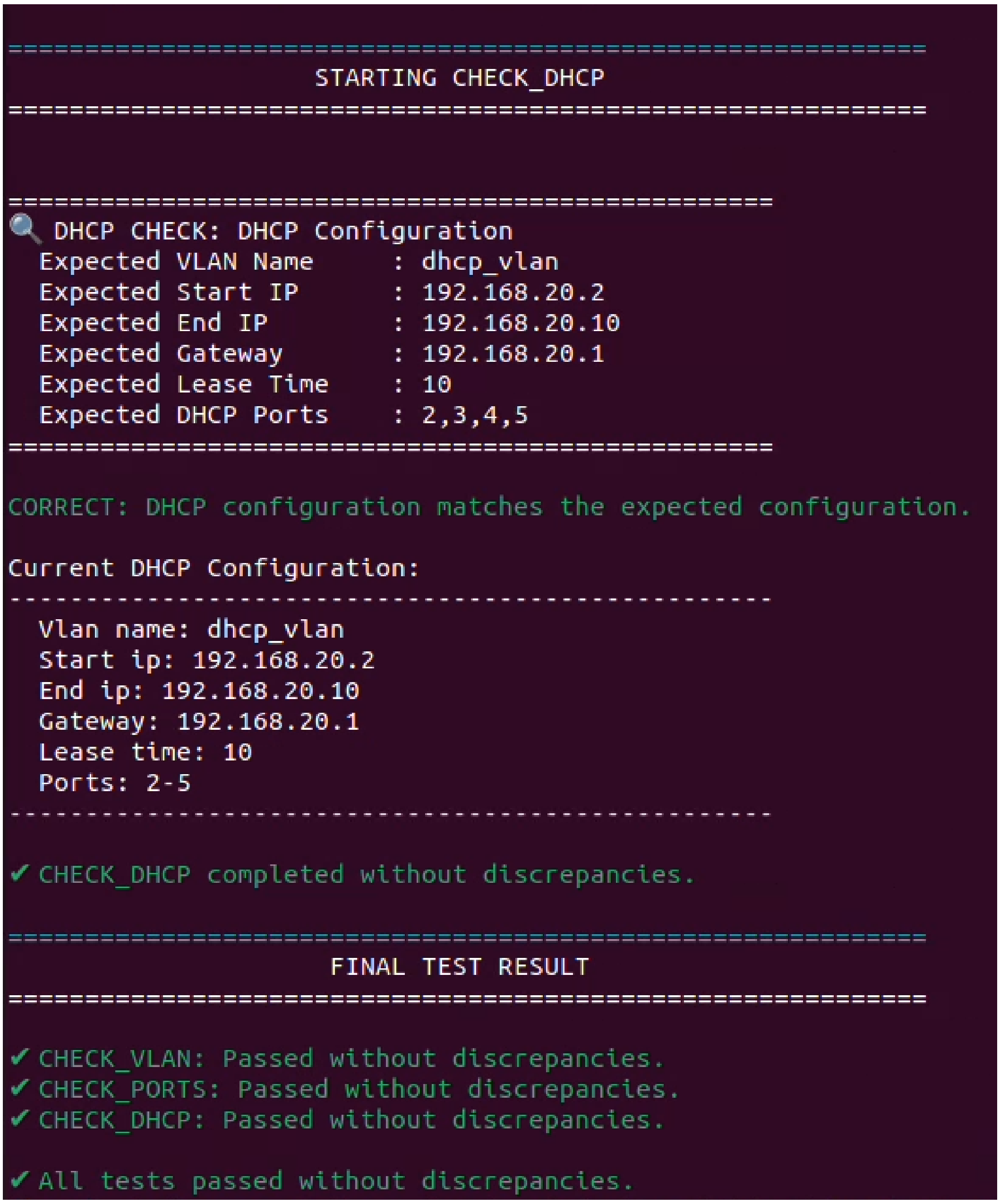

Figure 6.10. Output of the DHCP Verification Script

The script confirms that the DHCP configuration is correctly applied and all parameters match the requested values.

Client Connectivity Verification

Before DHCP configuration, the clients had no assigned IP addresses, as shown in the following output:

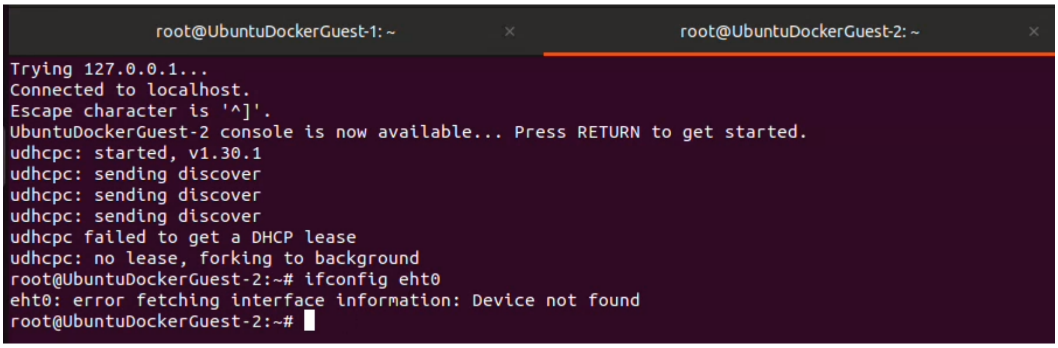

Figure 6.11. Client Devices Unable to Obtain IP Addresses Before DHCP Configuration

After the DHCP server is configured and activated, the clients successfully obtain IP addresses dynamically:

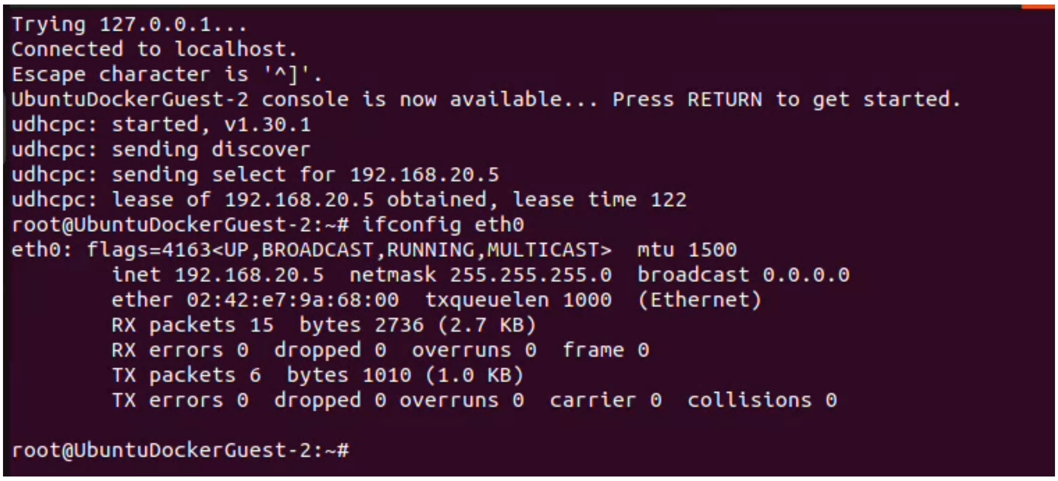

Figure 6.12. Clients Receiving IP Addresses via DHCP

The DHCP server is correctly configured on the EXOS switch, and the connected devices have successfully received dynamically assigned IP addresses:

```
EXOS-VM.4 # show dhcp-server
VLAN "dhcp_vlan":
    DHCP Address Range   : 192.168.20.2->192.168.20.10
    Netlogin Lease Timer : Not configured (Default = 10 seconds)
    DHCP Lease Timer     : 10 seconds
    Default Gateway      : 192.168.20.1
    Ports DHCP Enabled   : 2-5

    ===========================================================================
    IP                 MAC                 State      Lease Time Left
    ===========================================================================
       192.168.20.2    02:42:9d:4a:37:00   Expired    0000:00:00
       192.168.20.3    02:42:ab:ab:d2:00   Expired    0000:00:00
       192.168.20.4    02:42:98:dd:86:00   Expired    0000:00:00
       192.168.20.5    02:42:e7:9a:68:00   Assigned   0000:00:07
```

Figure 6.13. DHCP Server configured on the switch

A connectivity test is then performed, verifying that the clients can now communicate within the network:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
UbuntuDockerGuest-3 console is now available... Press RETURN to get started.
udhcpc: started, v1.30.1
udhcpc: sending discover
udhcpc: sending select for 192.168.20.2
udhcpc: lease of 192.168.20.2 obtained, lease time 122
root@UbuntuDockerGuest-3:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.20.2  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 02:42:9d:4a:37:00  txqueuelen 1000  (Ethernet)
        RX packets 12  bytes 1972 (1.9 KB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 6  bytes 1010 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@UbuntuDockerGuest-3:~# ping 192.168.20.5
PING 192.168.20.5 (192.168.20.5) 56(84) bytes of data.
64 bytes from 192.168.20.5: icmp_seq=1 ttl=64 time=5.00 ms
64 bytes from 192.168.20.5: icmp_seq=2 ttl=64 time=1.95 ms
64 bytes from 192.168.20.5: icmp_seq=3 ttl=64 time=2.43 ms
64 bytes from 192.168.20.5: icmp_seq=4 ttl=64 time=2.08 ms
64 bytes from 192.168.20.5: icmp_seq=5 ttl=64 time=2.49 ms
```

Figure 6.14. Successful Ping Test Between Clients After DHCP Configuration

This test proves the capability of the system to automate the deployment of a DHCP service on a network switch, guaranteeing that connected devices receive IP addresses dynamically. The end-to-end process, from the user request in the chatbot to the validation of applied configurations, shows that the framework correctly interprets user input, executes the necessary configurations through Ansible, and verifies consistency with the Python validation script. The successful assignment of IP addresses and the ability of client devices to communicate validate the accuracy and reliability of the automation workflow.

The same approach was used to configure and validate SNMP settings on the switch, ensuring that the service was correctly enabled and responding as expected. The chatbot facilitated the configuration request, Ansible applied the necessary commands, and the Python script confirmed that SNMP was properly set up. While the details are not discussed in a separate section, this test follows the same validation process used for VLAN and DHCP services, further proving the versatility of the framework in handling different network automation tasks.

6.4 Use Case 3: Large-Scale Multi-Vendor Deployment

As network infrastructures grow in complexity, to guarantee that the automation framework scales efficiently becomes a critical requirement. While initial tests focused on single-switch environments, a successful validation one was conducted in a multi-switch scenario, where VLAN configurations were applied across two interconnected ExtremeXOS switches. The automated deployment completed successfully, and devices in different VLANs across both switches were able to communicate as expected. This result strongly suggests that the system is scalable, as extending the same process to a larger number of switches follows the same principles.

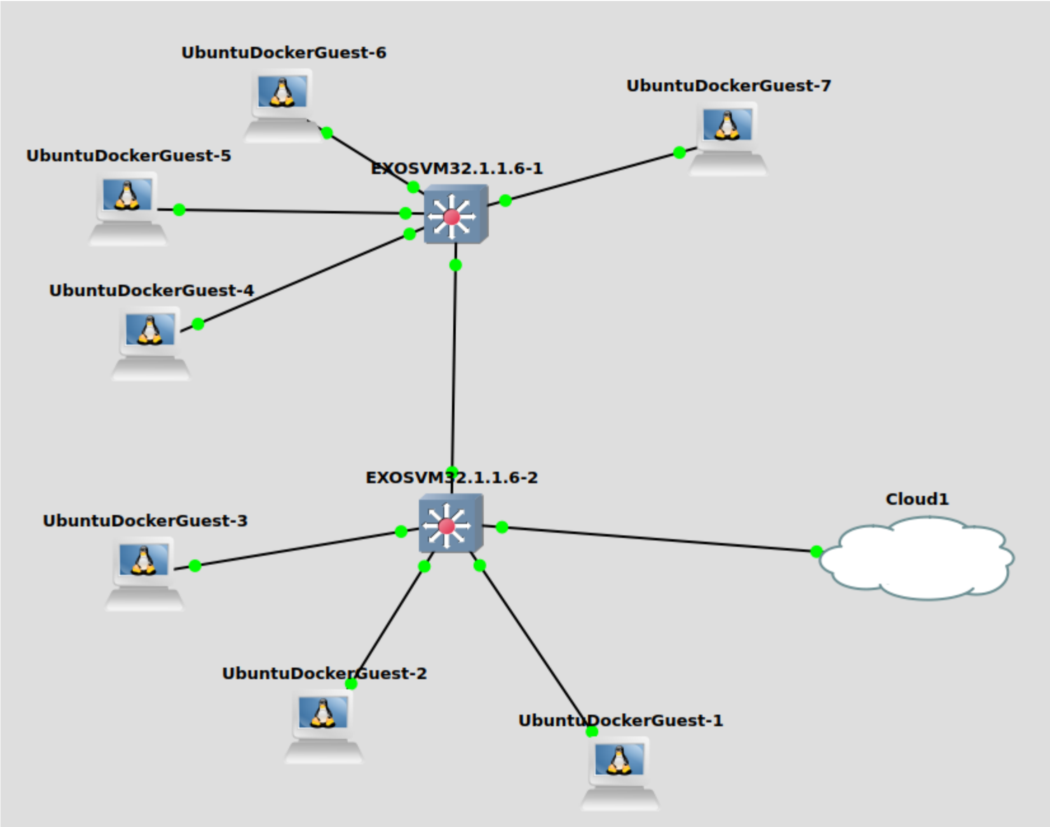

Figure 6.15. Topology Network with 2 switches

The scalability of the framework is primarily driven by parallel execution capabilities of Ansible: it can configure multiple devices in a single execution, provided they are listed in the inventory file. Consequently, to define additional switches in the inventory is sufficient to allow the system to configure an entire network in one automated run. The validation scripts follow a similar methodology, retrieving configurations independently from each device and identifying discrepancies without disrupting the network.

Beyond scalability, multi-vendor support is a fundamental aspect of the design of the system. Our tests were conducted exclusively on ExtremeXOS switches, but the framework was intentionally structured to handle different network operating systems. A crucial feature in playbooks of Ansible is the vendor detection step, which dynamically determines the operating system of the switch before executing any configurations: if the right commands are defined for another vendor, the automation process will work without requiring structural modifications.

Since the framework has demonstrated that it correctly applies commands for this switch brand, the same logic applies to any other ones. By simply guaranteeing that the correct vendor-specific commands are used within the playbooks and validation scripts, the system is naturally extensible to Cisco IOS, Arista EOS, Juniper Junos, and other platforms. Additionally, since Ansible natively supports multiple networking vendors through dedicated modules, to integrate a new vendor would only require adding the corresponding command mappings, rather than redesigning the entire system.

From a performance perspective, increasing the number of configured devices introduce new challenges, particularly in terms of execution time and network latency. The architecture is designed to be lightweight, with AWS Lambda dynamically generating inventory files and Ansible executing configurations in parallel, reducing the overhead associated with scaling. The transition from configuring a few devices to managing a full-scale network requires no changes to the logic, proving the efficiency and modularity of the approach.

The system has not yet been deployed in a true large-scale environment, but the successful validation of multi-switch automation and the modularity of vendor detection indicate that the framework is well-suited for expansion. With minimal additional effort, it can support larger and more diverse network infrastructures, maintaining the same level of reliability and automation efficiency.

6.5 System Validation and Error Handling

6.5.1 Error Handling in the Chatbot

The chatbot is the entry point for user interactions, guaranteeing that only valid requests proceed to the automation process. Two key validation mechanisms avoid misconfigurations at this stage.

First, if a user tries to configure a service without establishing an SSH connection, the chatbot blocks the request and prompts the user to authenticate before proceeding. This prevents executing commands on a switch without the necessary credentials. The following screenshot demonstrates this scenario:

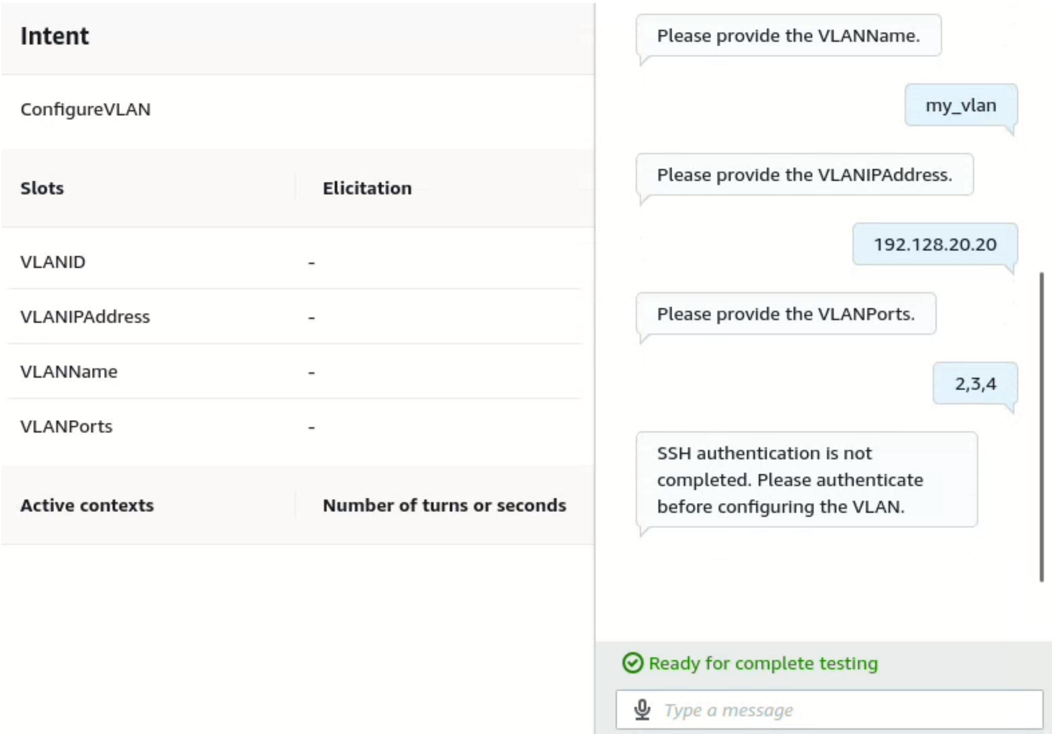

Figure 6.16. Chatbot error: SSH connection required before configuring services

Second, the chatbot performs IP address validation before accepting user input. If an invalid IP format is detected, the chatbot rejects the value and asks the user to enter a valid address. Below is an example of an invalid IP submission and the response of the chatbot:

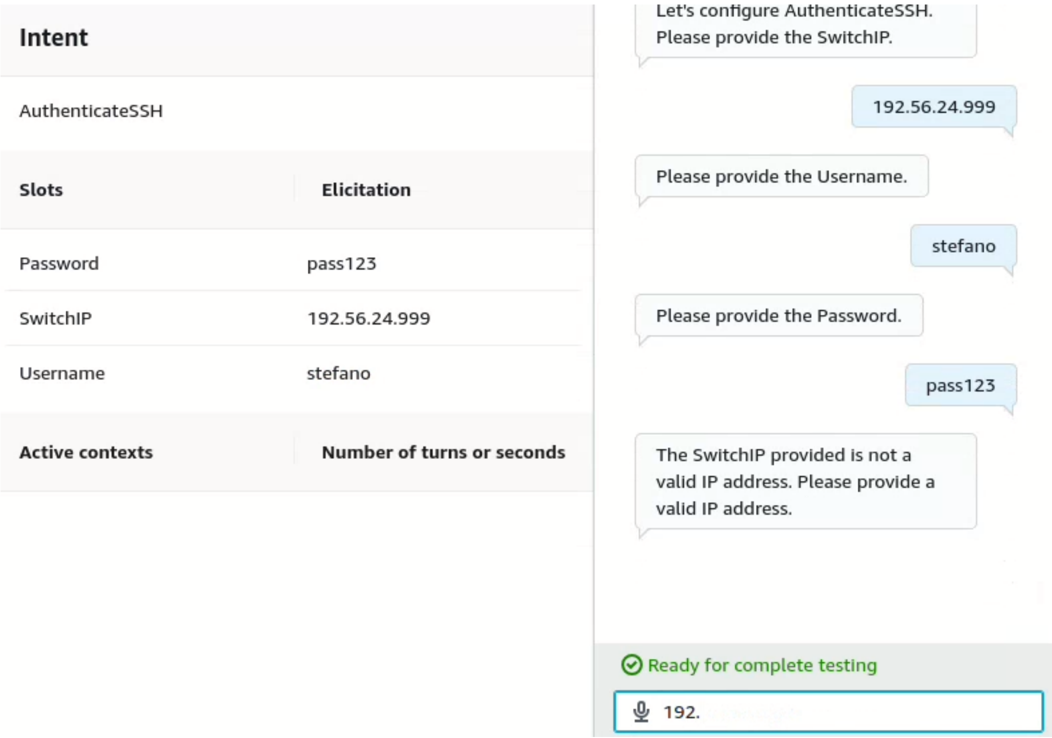

Figure 6.17. Chatbot error: Invalid IP address format

By enforcing these validations at the chatbot level, the system prevents misconfigurations before they even reach the automation framework.

6.5.2 Error Handling in Ansible

If a user requests the creation of VLAN 10 while it is already present on the switch, Ansible detects the conflict and returns an error. In this case, the rollback mechanism is triggered, restoring the last known working configuration and avoiding any undesired changes.

The following screenshot captures the output of Ansible when it tries to configure a duplicate VLAN:

Figure 6.18. Ansible error: Attempt to configure an existing VLAN

Immediately after detecting the failure, Ansible automatically initiates the rollback process, restoring the previously saved configuration from backup.

6.5.3 Validation and Error Detection with Python Scripts

In this test, we configure both VLAN and DHCP services, and all expected parameters match the applied settings. However, when checking the enabled ports, the validation script detects an extra port (Port 9) that was not originally specified in the user request. Despite the correct VLAN and DHCP settings, the script detects the unexpected port configuration and notifies it:

```
==================================================
🔍 PORT CHECK: Enabled Port Details
  Expected enabled ports: [1, 2, 3, 4, 5]
==================================================

NOT CORRECT: The port configuration is incorrect.
  Extra ports (enabled unexpectedly): [9]

Current enabled port configuration:
--------------------------------------------------
  Currently enabled ports: [1, 2, 3, 4, 5, 9]
--------------------------------------------------

✘ CHECK_PORTS encountered discrepancies.

============================================================
                   STARTING CHECK_DHCP
============================================================


==================================================
🔍 DHCP CHECK: DHCP Configuration
  Expected VLAN Name      : dhcp_vlan
  Expected Start IP       : 192.168.20.2
  Expected End IP         : 192.168.20.10
  Expected Gateway        : 192.168.20.1
  Expected Lease Time     : 10
  Expected DHCP Ports     : 2,3,4,5
==================================================

CORRECT: DHCP configuration matches the expected configuration.

Current DHCP Configuration:
--------------------------------------------------
  Vlan name: dhcp_vlan
  Start ip: 192.168.20.2
  End ip: 192.168.20.10
  Gateway: 192.168.20.1
  Lease time: 10
  Ports: 2-5
--------------------------------------------------

✔ CHECK_DHCP completed without discrepancies.

============================================================
                    FINAL TEST RESULT
============================================================

✔ CHECK_VLAN: Passed without discrepancies.
✘ CHECK_PORTS: Discrepancies found.
✔ CHECK_DHCP: Passed without discrepancies.

✘ Some tests failed. Please refer to the details above.
```

Figure 6.19. Validation script detecting an additional enabled port (Port 9)

This shows how the verification script provides a detailed analysis of network configurations, ensuring that even minor inconsistencies are identified and reported.

These tests prove the ability of system to detect and handle errors at multiple levels. The chatbot prevents invalid inputs, guaranteeing only correct requests proceed. The error handling and the rescue mode of Ansible maintain network stability, reverting changes if a misconfiguration occurs. Finally, the Python validation script detects discrepancies even when Ansible executes successfully, providing detailed reports for corrections. These combined mechanisms ensure *reliability*, *accuracy*, and a *robust automation framework*.

Chapter 7

Limitations and Future Works

The proposed framework successfully *automates* network configuration tasks, but several aspects can be further improved to enhance its usability, performance, and adaptability to various networking environments. The modular design of the system already allows easy extensions, but addressing certain limitations will be crucial.

This chapter introduces possible key areas for future development, focusing on multi-device support in the chatbot, reducing dependency on Ansible, implementing configuration versioning and additional enhancements such as role-based access control (RBAC) and graphical user interfaces (GUI). These improvements will contribute to make the system more robust, user-friendly, and adaptable to large-scale deployments.

7.1 Chatbot Multi-Device Support

The previous chapters have explained how the chatbot processes network configurations one device at a time. In this way the users require to start separate interactions for each switch: this approach is sufficient for small-scale configurations, but it becomes inefficient when dealing with multiple devices in large network infrastructures.

The chatbot could be extended to support multi-device configuration within a single interaction. Below there are possibile solutions.

- *More structured interface*: instead of asking the user to configure one device at a time, the chatbot could ask initially if the operation should be performed on one or multiple devices, collecting all the necessary information in a single session.

- *Batch configuration*: the user could provide a list of devices (e.g., with names or IPs) and then input the parameters just once for the operation (e.g., creating the same VLAN on multiple switches).

- *Group configuration*: instead of managing each switch individually, the system could allow the creation of groups of devices with similar characteristics and configure them all at once in a single operation.

7.2 Reducing Dependency on Ansible

The current system relies entirely on Ansible for applying network configurations. It makes automation very efficient but also introduces a strong dependency on a single technology. If Ansible does not support a specific vendor or if its performance becomes an issue at a large scale, the system may struggle to adapt.

Possible improvement to address this limitation are:

- *Hybrid Approach*: instead of using only Ansible, the system could combine different configuration methods (e.g., network device REST APIs, direct SSH commands). This would allow greater flexibility, especially for devices without dedicated Ansible modules.

- *Fallback options*: if an Ansible playbook fails, the system could have an alternative mechanism to attempt the configuration in a different way, reducing the risk of blocking errors.

7.3 Configuration Versioning

Currently, the system allows only a rollback to the last saved configuration, without a detailed history of changes. If a user want to revert to a specific configuration, this would not be possible.

To improve this functionality, future versions could implement:

- *Configuration History*: the system could save every applied configuration with a timestamp, allowing users to revert to a specific version when needed.

- *Configuration Comparison*: the system could provide a comparison between different versions, highlighting changes over time. One approach could be to integrate a Git-based repository where each configuration change is logged, allowing users to view historical modifications and restore specific versions as needed.

- *Distributed Backup*: in addition to local storage, configurations could be archived in a centralized repository, ensuring security and remote access.

7.4 Additional Enhancements

Beyond the major architectural improvements discussed, several additional enhancements could further improve the system's usability and security.

One key improvement is the introduction of *Role-Based Access Control* (RBAC): currently all users that interact with the chatbot and automation system have equal permissions, and this creates security risks in multi-user environments. Future implementations could define distinct roles (e.g., administrators, standard users, and read-only operators), restricting access based on authorization levels. This would prevent unauthorized users from executing high-impact configuration changes.

Another important enhancement is the development of a Graphical User Interface (GUI). The current system relies on command-line interactions and chatbot-driven inputs, while a web-based dashboard could significantly improve usability. The GUI could provide visual feedback on network configurations, real-time monitoring, and simplified troubleshooting, making the system more accessible to non-technical users.

This project has successfully demonstrated the feasibility of AI-driven network automation. Its future development will depend on its ability to adapt to emerging network technologies, expand support for multiple vendors, and enhance user experience through advanced interactivity and visualization tools. By implementing these improvements, the system can evolve into a fully-fledged, enterprise-grade automation framework capable of managing complex, multi-vendor network infrastructures.

Chapter 8

Conclusions

The increasing complexity of modern network infrastructures demands scalable, automated, and vendor-agnostic solutions for configuration management. To address this challenge, this thesis illustrated the development of a comprehensive *network automation framework* that integrates an AI-driven chatbot, AWS cloud services, Ansible playbooks, and Python validation scripts. The system was designed to bridge the *gap* between human intent and machine execution, enabling seamless network configuration through natural language interaction while ensuring accuracy and reliability.

Building on this approach, the implemented framework automates *essential network services*, maintaining a *modular and extensible architecture*. The design allows easy adaptation to new vendors and functionalities, providing *scalability* and *robustness* in real-world scenarios. Extensive testing in simulated environments demonstrated the system efficacy in applying configurations, detecting discrepancies and verifying alignment with user intent, enhancing validation beyond Ansible's standard feedback mechanisms.

Although the framework has demonstrated its effectiveness, certain limitations still need to be addressed. While Ansible allows parallel configuration, the chatbot currently processes requests one device at a time, requiring separate interactions for multiple switches. Additionally, the system currently supports *ExtremeXOS* and *Cisco*, but integration with other vendors requires further testing. The lack of a graphical user interface (GUI) and the dependency on Ansible for execution present opportunities for future enhancements. Introducing granular access levels could improve security and multi-user access management.

This research establishes the foundation for a scalable and intelligent network automation framework, with multiple possibilities for future enhancements. The framework has been validated in controlled environments, but *real-world deployment is essential* to evaluate its effectiveness in dynamic production networks.

This work demonstrates the potential of AI-assisted automation in networking, paving the way for *fully autonomous, self-optimizing network infrastructures*.

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

[2] Jacob Eisenstein. *Introduction to Natural Language Processing*. MIT Press, 2019.

[3] Wolfgang Ertel. *Introduction to Artificial Intelligence*. Springer, 2nd edition, 2018.

[4] Extreme Networks. *ExtremeXOS Quick Guide*, 2021.

[5] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating Configuration Management and Deployment*. O'Reilly Media, 2nd edition, 2017.

[6] Zied Ben Houidi and Dario Rossi. Neural language models for network configuration: Opportunities and reality check. *Computer Communications*, 2022.

[7] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Pearson, 2023.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[9] Jason C. Neumann. *The Book of GNS3: Build Virtual Network Labs Using Cisco, Juniper, and More*. No Starch Press, 2019.

[10] Ansible Project. *Ansible Documentation: Inventory Guide*, 2023.

[11] GNS3 Project. *GNS3 Official Documentation*, 2023.

[12] IEEE Computer Society. Ieee recommended practice for software requirements specifications, 1998.

[13] Alan Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 30, 2017.

[15] Andreas Wittig and Michael Wittig. *Amazon Web Services in Action, Third Edition: An In-depth Guide to AWS*. Manning Publications, 2023.

[16] Jonathan Womack and Scott Gray. *Introduction to Ansible Network Automation: A Practical Primer*. Apress, 2023.