POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Instructing network devices via large language models

Supervisors

Prof. Alessio SACCO

Prof. Guido MARCHETTO

Ph.D. Student Antonino ANGI

Candidate

Francesco DALL'AGATA

April 2024

Summary

In recent years, there have been significant advancements in Artificial Intelligence (AI), particularly in Natural Language Processing (NLP). These advancements have introduced a groundbreaking approach to effectively manage the escalating complexity of network environments, simplifying network operations by allowing operators to use declarative interfaces instead of imperative ones. This paradigm shift, known as "Intent-Driven Networking," facilitates the conversion of human language into network configurations through the use of NLP techniques, eliminating the necessity for manual coding or execution. Furthermore, the advent of state-of-the-art tools such as GPT, LLama, and PaLM, which possess remarkable capabilities in understanding and generating complex natural language, opens up exciting possibilities for Intent-Driven Networking. This thesis aims to exploit these innovative tools, collectively referred to as Large Language Models (LLMs), by developing a prototype capable of translating high-level policies into actionable network configurations within the realm of Software Defined Networks (SDN). To achieve this objective, we propose a pipeline where the LLM's role is to extract relevant information from human intent. Subsequently, the structured policy extracted is mapped to the Application Programming Interfaces (APIs) of network applications. Through this approach, we want to reduce the gap between high-level policy formulation and network configuration implementation. This model's implementation involves analyzing the few-shot learning capabilities of LLMs to translate a human intent into a machine-readable policies for network devices, utilizing tools such as Langchain to simplify the development of LLMs-based applications and integrating network tools like Ryu-SDN and P4-eBPF. Additionally, the model has been evaluated across various use cases such as firewall, rate-limiting, and load profiling. This evaluation ensures the accurate translation of intent into the appropriate machine-readable format and its effective implementation within the network infrastructure.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Guido Marchetto, Alessio Sacco and Antonino Angi. Their guidance, encouragement, and invaluable insights throughout this project were instrumental in its completion.

I am also deeply indebted to my family for their unwavering support throughout my academic journey. To my parents, Antonio and Antonella, thank you for your constant love and encouragement. You instilled in me a love of learning and the perseverance to see things through. To my sister, Carola, thank you for your support, especially during the more challenging moments. You embody the resilience during tough moments that inspired me to reach my objectives. And to my nephew, Anna, thank you for reminding me of the joy of a family.

Finally, a heartfelt thank you to my friends for their unwavering support and understanding. Their encouragement and sense of humor kept me going during the long nights and helped me maintain a healthy perspective throughout this process. To all of you, thank you from the bottom of my heart. This thesis would not have been possible without your love, support, and guidance.

“ When positivity seems hard to reach. I keep my head down and my mouth shut
’Cause if you’re going through hell. Well, just keep going”
by King Krule in the song Easy Easy.

Table of Contents

1 Introduction 1
1.0.1 Thesis Structure . . . 2

2 Literature Review 3
2.1 Overview on Large Language Models . . . 3
2.1.1 Researches on Prompt Engineering . . . 4
2.1.2 Research about Semantic Parsing using LLMs . . . 7
2.1.3 PALM: Pathways Language Model . . . 7
2.2 Network tools based on NLP algorithms . . . 8
2.2.1 Lumi ChatBot . . . 8
2.2.2 NLP4 and NAIL . . . 9
2.3 Our Contribution . . . 11

3 Background 12
3.1 The Network Device structure . . . 12
3.2 SDN: Software Defined Networks . . . 13
3.2.1 SDN's Architecture . . . 14
3.2.2 OpenFlow . . . 15
3.3 Tools for Building SDN Applications . . . 16
3.3.1 Ryu-SDN . . . 16
3.4 P4-eBPF . . . 16
3.4.1 P4 . . . 16
3.4.2 P4's core processing pipeline . . . 17
3.4.3 P4's Architecture . . . 18
3.4.4 eBPF . . . 19
3.4.5 From P4 to eBPF . . . 21
3.4.6 NIKSS-vSwitch . . . 21
3.5 Natural Language Processing . . . 24
3.6 Large Language Models . . . 25
3.6.1 An overview of LLM's Architecture . . . 26
3.6.2 Using LLM's APIs . . . 29

3.6.3 Prompt Engineering 30
3.7 Gemini 30
3.8 Langchain 31

4 System's Architecture **33**
4.0.1 System's Architecture 33
4.1 A deep dive into the architecture 34
4.1.1 Converter 34
4.1.2 Application Program Interface 35
4.1.3 Network Applications 35

5 System's Implementation **37**
5.1 Converter 37
5.1.1 JSON Objects 37
5.1.2 Prompts 39
5.1.3 Converter's Program 44
5.2 Application Program Interface 48
5.2.1 OpenFlow's API 48
5.2.2 P4-eBPF's API 48
5.3 Network Application: Ryu-SDN 48
5.3.1 Ryu Controller's implementation 49
5.3.2 Network Functions 51
5.4 Network Application: P4-eBPF 52
5.4.1 P4 program 53
5.4.2 NKISS-CTL 55
5.5 Network Infrastructure 55

6 Evaluation **56**
6.0.1 Testbeds 56
6.1 Network Application: Ryu-SDN 58
6.1.1 Adaptive multi-paths routing 58
6.1.2 Firewall ACLs 59
6.1.3 Rate Limiter 59
6.1.4 Load Profiling 62
6.2 Network Application: P4-eBPF 66
6.2.1 Firewall ACLs 66
6.3 Rate Limiter 66
6.3.1 Load Profiling 68
6.4 Large Language Model (LLM) Correctness 71
6.4.1 Testbed 72
6.4.2 Results 73

6.5 Conclusion 74

7 Conclusion 76

7.1 Final Considerations 76

7.2 Limitations 76

7.3 Future works 77

Bibliography 78

Chapter 1

Introduction

The ever-increasing complexity of network environments demands smarter and more intuitive management solutions. This thesis explores the potential of Intent-Driven Networking (IDN), a revolutionary approach powered by Natural Language Processing (NLP) and Large Language Models (LLMs), to simplify network operations.

Traditionally, network configurations require manual coding and execution, posing a significant barrier for non-technical users. IDN disrupts this paradigm by enabling declarative interfaces, where users express their intent in natural language, instead of writing complex code. This shift empowers both network experts and non-technical stakeholders to participate in network management effectively.

The emergence of powerful LLMs such as GPT, LLama, and PaLM marks a significant leap in NLP capabilities. This thesis leverages these tools to develop a prototype that bridges the gap between high-level policies and actionable network configurations within Software Defined Networks (SDN).

Our proposed pipeline relies on the LLM's ability to extract key information from human intent. This extracted data is then mapped to the Application Programming Interfaces (APIs) of network applications, enabling seamless translation from policy to configuration.
This research aims to:

- Reduce the complexity of network management by leveraging the power of natural language.
- Democratize network control by making it accessible to users with varying technical expertise.
- Explore the real-world potential of LLMs in network management, exploiting

their few-shot learning capabilities.

By successfully implementing this prototype, we pave the way for a future where network management becomes more intuitive, efficient, and accessible to all.

1.0.1 Thesis Structure

The thesis is divided into seven chapters:

- **Chapter 2 (Literature Review)**: Describes the research related to the thesis topics. In particular, it reviews existing research on large language models (LLMs), prompt engineering, semantic parsing using LLMs, and network applications based on NLP algorithms.
- **Chapter 3 (Background)**: This chapter provides background knowledge on network device structure, SDN architecture, OpenFlow, tools for building SDN applications (including Ryu-SDN and P4-eBPF), Natural Language Processing (NLP), and LLMs. It also details the architectures of LLMs and how to use their APIs.
- **Chapter 4 (System's Architecture)**: This chapter dives deep into the system's overall architecture, explaining its components and illustrating how they work and are connected.
- **Chapter 5 (System's Implementation)**: This chapter details the implementation of the system's components, including the converter, API, network applications (focusing on Ryu-SDN and P4-eBPF), and network infrastructure.
- **Chapter 6 (Evaluation)**: This chapter explains how the system is evaluated. It describes the testbeds used and the evaluation results for network applications (like Ryu-SDN and P4-eBPF) and functionalities (like firewall ACLs, rate limiter, and load profiling). It also evaluates the correctness of the LLM.
- **Chapter 7 (Conclusion)**: This chapter summarizes the thesis, discusses its limitations, and proposes future works.

Chapter 2

Literature Review

In this chapter, we embark on a comprehensive literature review related to the conducted research. The first section delves into the emergence and potential of Large Language Models (LLMs) and their applications within the domain of Natural Language Processing (NLP). Additionally, we investigate the potential of LLMs in facilitating few-shot learning, enabling models to generalize to new tasks with limited training data. Moving on to the second section, we provide detailed descriptions of different tools utilizing NLP techniques or LLMs to streamline network management processes.

2.1 Overview on Large Language Models

The advent of Large Language Models (LLMs) represents a significant milestone in the realm of natural language understanding. Developed over recent years, these deep neural network models have garnered widespread attention for their remarkable capacity to comprehend human language, marking a substantial leap forward in Natural Language Processing (NLP). Notably, LLMs exhibit exceptional adaptability compared to their predecessors in the NLP domain, boasting proficiency across a diverse range of NLP tasks rather than being confined to specific applications.

In recent years, tech giants like Google, OpenAI, and Facebook have unveiled their own implementations of large language models (LLMs), including GPT-3 [1], GPT-4 [2], PALM [3], Gemini [4] and LLama [5].

The vast versatility of Large Language Models (LLMs) has been explored in numerous research studies. Many of these studies concentrate on evaluating the prompting technique, which involves structuring input prompts to LLMs to attain

desired outputs. Guidance provided to Large Language Models (LLMs) via prompts specifies the task to be accomplished and the desired output format. These prompts encompass directives, contextual cues, input data, and output cues. Leveraging prompt engineering enables the utilization of LLMs across a spectrum of tasks, ranging from basic question answering to intricate creative text synthesis. This methodology relies on an emerging capability known as in-context learning [6], enabling LLMs to perform a task from a small of amount of examples.

2.1.1 Researches on Prompt Engineering

In this section, we will present the primary research endeavors undertaken in the domain of prompt engineering. To assess the efficacy of these models in conjunction with these techniques, these studies rely on various benchmarks and datasets whose aim is to evaluate the LLM's performance in various fields.

One of the pioneering studies in this field was conducted by Tom B. Brown in the paper titled "Language Models are Few-Shot Learners" [1]. Focused on GPT-3, a 175-billion-parameter autoregressive language model, the paper underscores the potential of Large Language Models (LLMs) to excel in few-shot settings. This implies that GPT-3 can carry out different Natural Language Processing (NLP) tasks with just a few examples provided in the input prompt (Fig. 2.1). This approach diverges from the traditional fine-tuning method, where models typically necessitate extensive supervised datasets to perform specific tasks. For example, GPT-3 gains great results on Reading Comprehension using as dataset CoQA (Conversational Question Answering), which is a dataset containing 127.000 question-answer pairs from different topics. In particular, it achieves 85 F1 in a few-shot setting. Another interesting result has been achieved on the SuperGLUE benchmark, which is a dataset to evaluate the LLMs on different NLP tasks. In this case, GPT-3 achieves in average a 71.8% Brown's study demonstrates GPT-3's impressive performance on various NLP tasks in zero-shot, one-shot, and few-shot settings. In some cases, the model even achieved performance approaching the state-of-the-art for fine-tuned systems.

Another interesting study was conducted by Google Research Team [8] where the field of study was a prompting technique called Chain of Thoughts(CoT). The idea is to guide the model's thinking through a series of connected reasoning steps. This approach notably enhances the ability of LLM to engage in complex reasoning tasks Fig. 2.2.

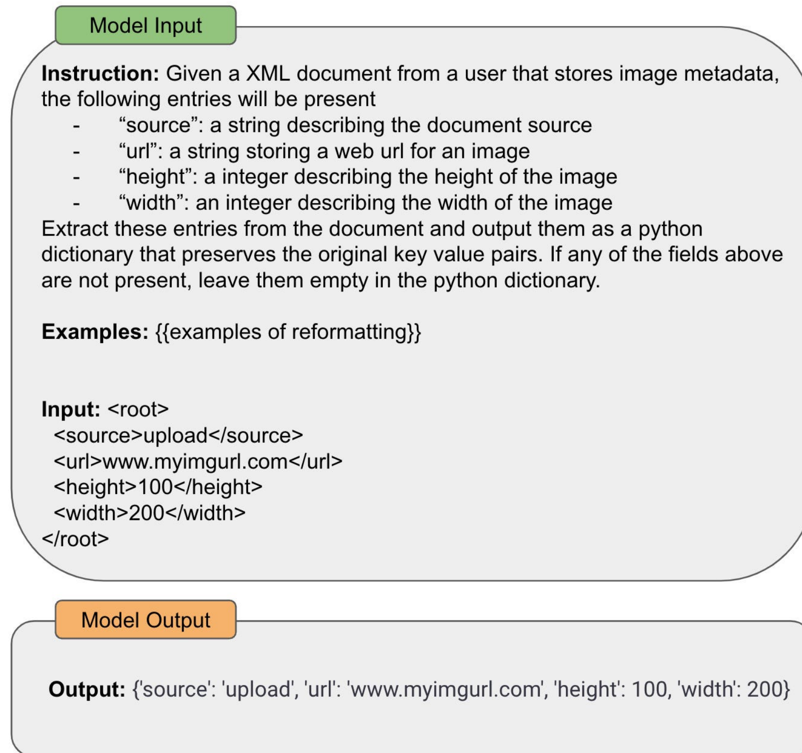

Figure 2.1: An examples of few shots prompting [7]

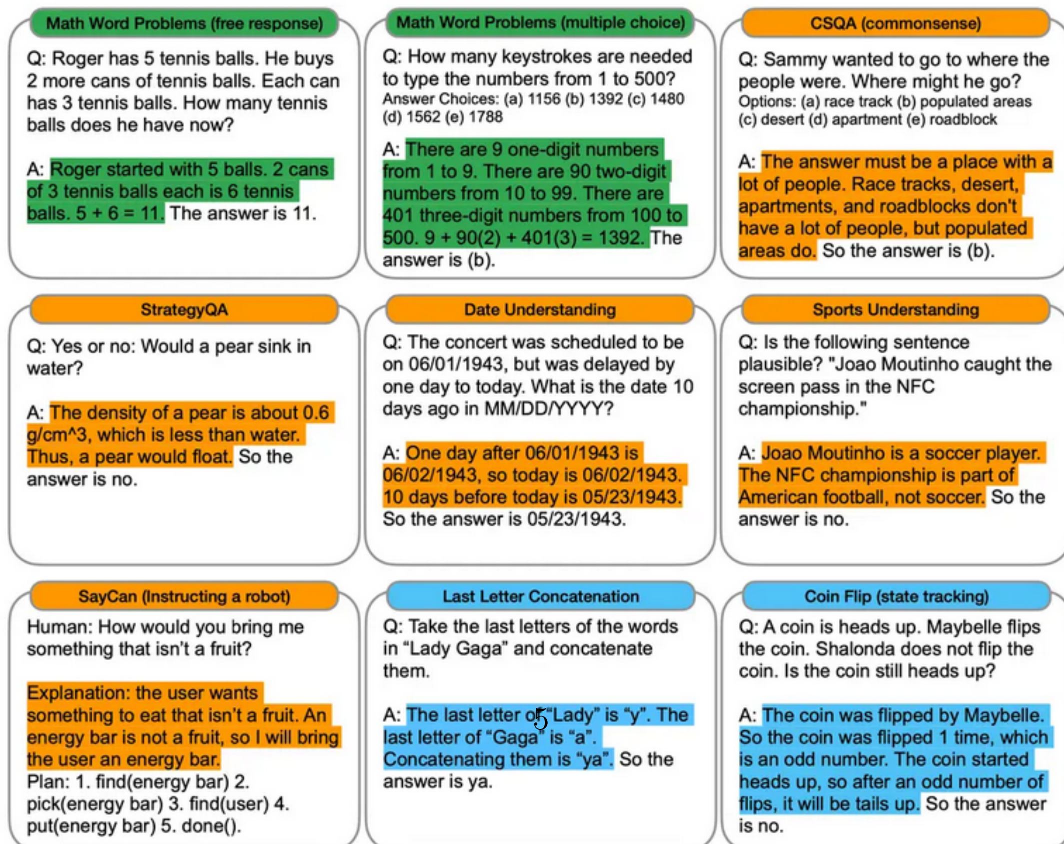

Figure 2.2: A series of examples with CoT prompt technique [8]

The results in the field of Arithmetic and Common reasoning are impressive, especially for Google's LLM model (Fig. 2.3 and Fig. 2.4).

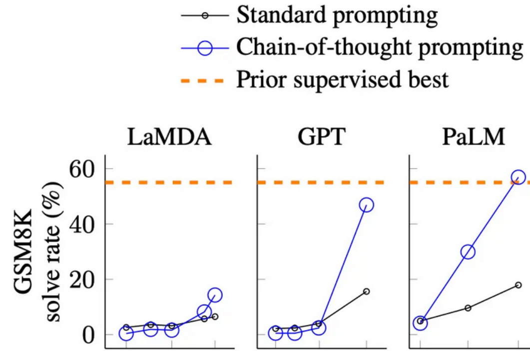

Figure 2.3: Scores of CoT prompt on math problems [8]

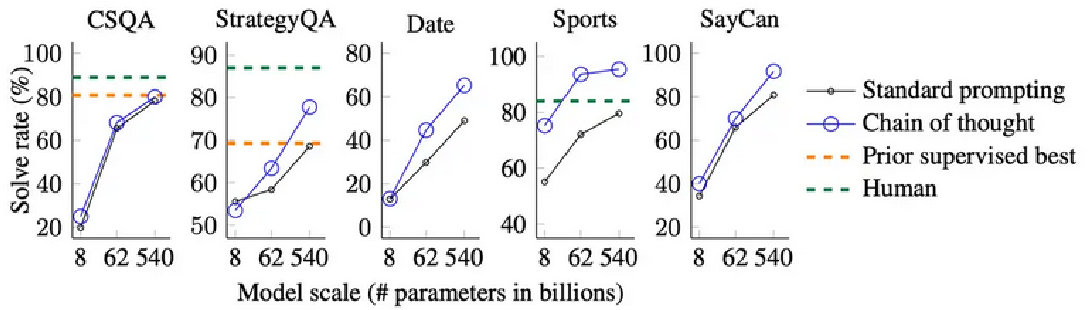

Figure 2.4: Scores of CoT prompt on commonsense reasoning [8]

From these two key prompting techniques have been developed other techniques are promising: Zero-Shot CoT [9], Automatic CoT [10], Self-Consistency [11]

2.1.2 Research about Semantic Parsing using LLMs

One key aspect related to our work is the ability of Large Language Models (LLMs) to perform semantic parsing, the task of converting natural language into a machine-readable format. In the realm of LLMs, an interesting study by Y. Wu in the paper "Semantic Parsing by Large Language Models for Intricate Updating Strategies of Zero-Shot Dialogue State Tracking" [12] investigates this capability.

The study shows that LLM models like GPT-3 can effectively translate a dialogue into a JSON representation, even in situations where no prior training examples are provided (zero-shot setting). This is particularly beneficial for Dialogue State Tracking (DST), which aims to estimate user goals based on the context of a conversation.

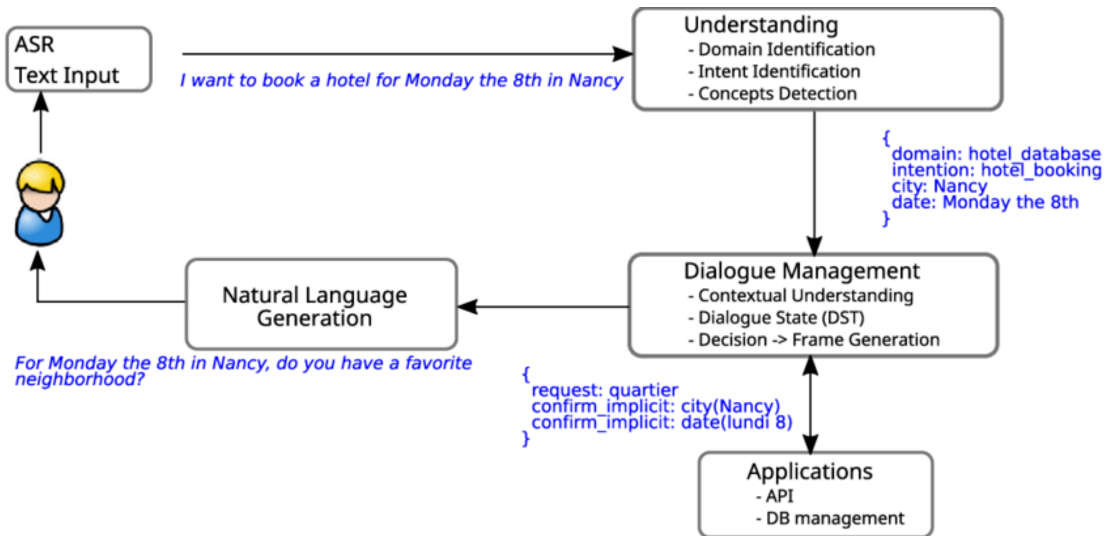

Figure 2.5: Overview of Task-Oriented Dialogue [13]

DST serves as a crucial step in Task-Oriented Dialogue (TOD) systems. These systems are designed to assist users in achieving specific goals by tracking dialogue states and generating appropriate responses [14].

2.1.3 PALM: Pathways Language Model

This thesis employs PaLM as its main model, a large language model (LLM) developed by Google AI with 540 billion parameters and strong performance on

various NLP tasks. This chapter analyzes the key characteristics of this model.

- **Revolutionary Architecture**: PaLM utilizes a novel decoder-only transformer architecture, achieving training efficiency through the Pathways system. This system primarily involves a different formulation of transformer modules and parallel attention and feedforward layers, enabling parallel processing.
- **Strong Few-Shot Performance**: Compared to other state-of-the-art models like GPT-3 and LaMDA, PaLM excels in few-shot settings across various tasks.
- **Reasoning and CoT Prompting**: PaLM exhibits impressive performance in reasoning and chain-of-thought (CoT) prompting tasks. Notably, it surpassed the previous state-of-the-art score of 55% held by GPT-3, achieving 58% accuracy on math questions in the GSM8K benchmark.

2.2 Network tools based on NLP algorithms

In this paragraph, we are going to illustrate some examples of tools that leverage NLP techniques to process human intent into network configuration. In particular, we are going to analyze: Lumi [15], NLP4 [16], and NAIL [17].

These applications are related to a core paradigm which is Intent-Based Networking. Traditional network configuration involves manually specifying commands and configurations for each device. IBN takes a different approach. It focuses on understanding the desired outcome (intent) expressed by the network administrator, rather than the specific steps to achieve it. IBN leverages technologies like Artificial Intelligence (AI) and Machine Learning (ML) to translate your intent into the necessary configuration changes across the network. This automation reduces errors and simplifies management.

2.2.1 Lumi ChatBot

Lumi can be seen as an attempt to reach the goal of IBN processing an intent written in natural language by an operator, so a system that allows "to talk to the network". The LUMI system is structured into four main blocks:

- **Information Extraction**: This block utilizes established machine learning algorithms, particularly Named Entity Recognition, to extract labeled entities from the input provided by the operator. The goal here is to identify and extract relevant information from the written input.

- **Intent Assembly**: Acting as an intermediary layer between natural language intents and network configuration demands, this block abstracts the extracted information. It bridges the gap between the language used by the operator and the technical requirements of network configuration.
- **Intent Confirmation**: Once the intent is assembled into the abstraction layer, this block verifies the correctness of the extracted configuration. It prompts the operator to confirm the accuracy of the configuration generated based on the input provided.
- **Intent Deployment**: The final step involves translating the abstract configuration into a real network configuration language. This block projects the configuration derived from the abstraction layer into a format suitable for implementation in an actual network.

As already said, LUMI uses an abstraction layer to structure the tagged entity in a schema that can be used sequentially to generate the related configuration. The abstraction layer used is Nile, which is useful for writing high-level abstraction for structured intents and is able with its syntax to cover many real-world intents. Also, a feature of Nile is to project his structured schema to a number of different existing network configurations. In order to use this abstraction layer, the tagged phase of the entities becomes crucial, because the entity and the tag must be selected precisely in order to be applied to the abstraction model proposed by Nile. Lumi's creator tries to mitigate this problem, using a series of generally tagged entities to let the operator express freely and a mechanism of feedback in order to correct some errors between the operator's expectation and the resulting generated intent by Lumi. This mechanism applied as a normal chatbot, is used also to correct some missing or unrecognized parameters in the operator's request. Also, Lumi seems to have some difficulties identifying some complex or unusual phrases and has problems tagging some terms. This is also due to the fact that it is difficult to find large datasets for training the model and Named Entity Recognition could lead to some misunderstandings of the input text or it's difficult to tag some words that are placed in the text in an unusual way respecting the standard pattern.

2.2.2 NLP4 and NAIL

NLP4 [16] and NAIL [17] are innovative approaches developed by the researchers of Politecnico di Torino that aim to simplify network requirement specification and make network programming accessible to users, especially those with limited experience in networking.

NLP4 utilizes Natural Language Processing (NLP) techniques to translate user

intents expressed in human language into P4 program code, which is used to customize network behavior based on user preferences. The architecture involves preprocessing user intents, tokenizing and encoding them using a dictionary, and applying a MultiLayer Perceptron (MLP) model to map the intents to specific network elements.

One key aspect of NLP4 is its ability to automatically program P4 switches to meet desired traffic profiles specified by users. By converting user intents into P4-enabled switches' configuration files, NLP4 enables users to customize network behavior without the need for extensive programming knowledge.

Furthermore, NLP4 has been improved in more sophisticated architecture called NAIL.
NAIL leverages the same NLP4 approach of translating human intents into P4 configurations for data-plane switches and it is composed mainly of four blocks:

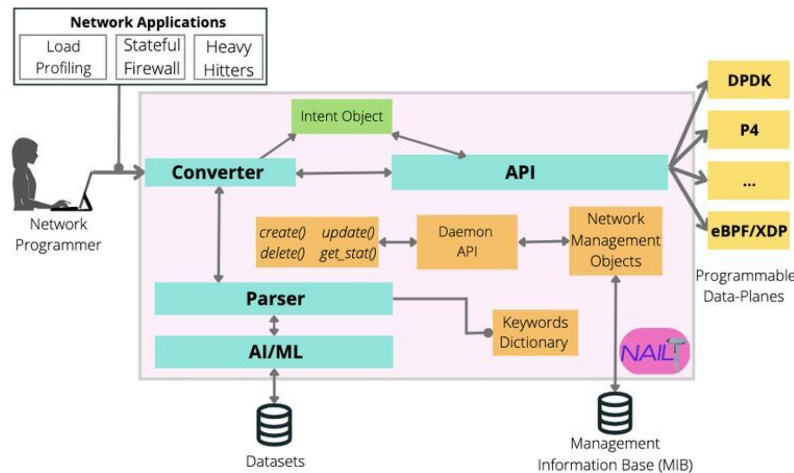

Figure 2.6: NAIL's Architecture [17]

- **Converter**: This block utilizes NLP techniques and a parser to translate human intent into a machine-readable object called an "intent object."
- **API**: This interface implements all possible methods for operating on the intent, such as creating, deleting, updating, or retrieving statistical information about the network status.
- **Management Information Base**: A database that stores information about the network infrastructure and the applied intents.

- **NLP Dataset**: A dataset used by the NLP algorithms to translate human intent into an intent object.

Actually, NAIL offers three network services: load profiling, stateful firewall, and heavy hitters. The user only needs to specify the desired configurations offered. NAIL will then create a new intent object by translating the human intent into a machine-readable configuration using NLP libraries. Subsequently, the intent will be applied to the network infrastructure using API calls.

2.3 Our Contribution

Our work draws inspiration from the mentioned applications, with a particular emphasis on NAIL's structure for processing intents. Specifically, we focus on translating human intent into network configurations, utilizing the few-shot capabilities of large language models to transform human intent into a machine-readable format. These configurations are then applied to the network infrastructure using API calls.

Chapter 3

Background

In this chapter, we will delve into the key tools and concepts central to our thesis. We will begin by exploring the Software Defined Network (SDN) paradigm, a revolutionary approach to networking that decouples the control plane from the data plane, offering dynamic network programmability and flexibility. We will discuss notable SDN tools such as Ryu-SDN, an open-source controller platform that facilitates the implementation and management of SDN networks, and P4-eBPF, a compiler to translate P4 programs into eBPF code, enabling programmable data plane processing. Furthermore, we'll examine the emerging field of Large Language Models (LLMs), which represent a significant advancement in natural language processing (NLP). These models are trained on vast amounts of text data and demonstrate remarkable capabilities in understanding and generating human-like text. We'll explore the architecture, capabilities, and applications of LLMs. Finally, we will describe Langchain, a tool that is used to simplify the development and the usage of LLM-based application.

3.1 The Network Device structure

Every device in networking can be seen as composed of three main logical levels:

- **Data Plane:** This is the lower level of the entire stack. The main duties of this level are the forwarding of traffic, executing few and simple instructions such as receiving, forwarding, and transmitting.
- **Control Plane:** This is the intermediate level of the stack. The aim of this level is to be responsible for managing and controlling the overall behavior of the network, including routing, signaling, configuration, and management of network devices.

- **Management Plane:** This is the higher level of the stack. It deals mainly with administrative tasks, monitoring, and maintaining the network infrastructure.

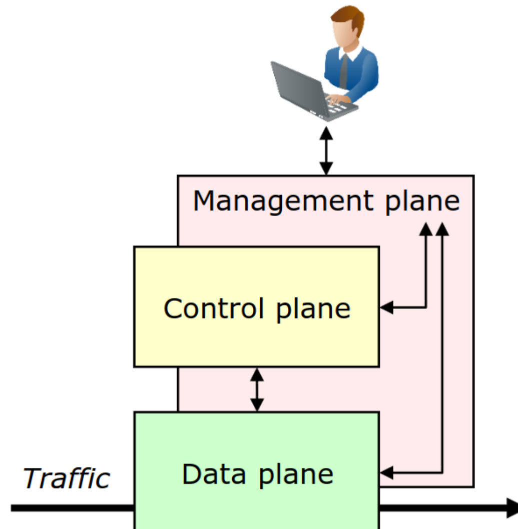

Figure 3.1: Network device's architecture [18]

3.2 SDN: Software Defined Networks

For many years, the IT industry has favored a market topology that can be described as "vertical." This entailed offering vertically integrated solutions comprising hardware, operating systems, and application systems. However, Software Defined Networks (SDN) represent a paradigm shift aimed at restructuring network architectures and transitioning from a closed, proprietary market to a more flexible, horizontally oriented one. SDN seeks to revolutionize network infrastructure by promoting programmability, allowing developers to exercise greater control over network implementation through software rather than being constrained by proprietary systems. The primary objective of SDN is to enhance network agility and customization. Key characteristics of Software Defined Networks include:

- **Separation of Control Plane and Data Plane**: In the case of SDN, the Control Plane and Data Plane are physically separated. This characteristic provides two benefits: the flexibility of choosing the Data Plane independently of the Control Plane and the possibility to change the Control Plane without affecting the programming of the Data Plane. Specifically, in an SDN scenario, the Control Plane and Data Plane are called the SDN Controller and SDN Switch, respectively.

- **Simplified Data Plane**: The Data Plane is composed of very simple components. Their tasks are mainly forwarding packets. They must be dumb, without any heavy logic, and equipped with a small firmware that allows the data plane to communicate with the Control Plane. The hardware must be simple as well in order to manage only simple data structures.

- **Centralized Controller**: The centralized controller refers to a network structure where we have a single SDN Controller for the entire network infrastructure. The SDN controller is an important component that acts as the brain of the network. It mainly manages the flows of network traffic and provides a centralized view and control of the entire network infrastructure, separating the control plane (logic that determines where traffic is sent) from the data plane (actual forwarding of network packets).

- **Programmability**: Programmability in Software Defined Networks (SDN) facilitates automation, flexibility, and customization of network management through software applications and scripts. By leveraging programmable interfaces, organizations can automate tasks, customize network behavior, and integrate with orchestration platforms to streamline operations and gain valuable insights into network performance and security. This approach empowers organizations to adapt quickly to changing business needs, optimize resource utilization, and innovate with tailored network solutions.

3.2.1 SDN's Architecture

The SDN architecture consists of three primary tiers interconnected through southbound and northbound APIs:

- **Application tier**: This encompasses the control program engineered to execute the network control logic, capable of operation either locally or on a remote system.

- **Northbound interface**: This component defines a clear and structured API.

- **Control tier**: This serves as the central intelligence of the network. It interprets commands from the application layer, communicates with the underlying infrastructure, and retrieves pertinent information, offering a comprehensive network overview to the higher tiers.

- **Southbound interface**: This establishes a standardized open interface to interact with physical devices.

- **Data tier**: This segment identifies the distributed forwarding components.

There are several protocols that can implement the Southbound interface. One of the most used is OpenFlow.

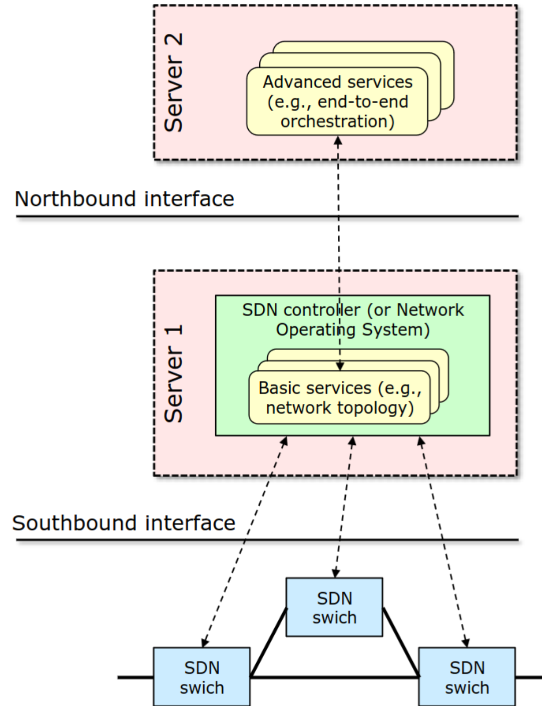

Figure 3.2: SDN's architecture [18]

3.2.2 OpenFlow

OpenFlow [19] is the protocol used to transmit control commands from the SDN Controller to the Switch, commonly referred to as an OpenFlow Switch. In the realm of SDN networks, OpenFlow supports the concept of fully programmable network devices and facilitates the development of various services within the network. With the capability of housing multiple programmable services within a single device, OpenFlow employs a table-based protocol. Each OpenFlow Switch contains one or more tables designed to identify types of traffic and apply one or more actions accordingly. OpenFlow 1.2 primarily defines three types of rules: flow, group and meter. The flow-table is a fundamental component of the OpenFlow Switch, serving as the primary mechanism for traffic management. It comprises three key fields: rule determines the criteria for selecting specific types of traffic (also called match), action specifies how packets are to be handled, and statistics tracks various counters. This table plays a central role in orchestrating the routing and processing of network data within OpenFlow-enabled environments. They can

be used to enforce bandwidth limits, prioritize certain types of traffic, or implement other traffic management policies. The group-table is similar to the flow-tables but it allows the administrator to define groups of actions to the packets selected. For example, load-balancing packets among different ports. The meter-table is used for Quality of Service(QoS) and rate limiting. They can define the policy of rate limiting for specific flows. They can also be used to enforce bandwidth limits, prioritize certain types of traffic, or implement other traffic management policies.

3.3 Tools for Building SDN Applications

There are several tools that allow the building of SDN applications. In this thesis, we have used different:

- **Ryu-SDN**: an open-source software-defined networking (SDN) controller
- **P4-eBPF**: a tool to translate P4 network program into eBPF code

3.3.1 Ryu-SDN

Ryu-SDN [20] is an open-source network operating system designed to provide programmable control and management of network devices. Its modular and flexible architecture features a controller core that coordinates network operations, along with a component-based design for handling specific functionalities like packet processing and flow management. With support for protocols such as OpenFlow, Netconf, and Of-config, Ryu enables developers to create custom SDN applications tailored to diverse networking requirements. Its event-driven architecture ensures responsiveness and scalability, while RESTful APIs facilitate integration with external systems. Additionally, Ryu supports network virtualization and incorporates optimizations for scalability and performance, making it a versatile platform for implementing software-defined networking solutions in modern infrastructures.

3.4 P4-eBPF

This section describes P4-eBPF, beginning by defining what P4 and eBPF are.

3.4.1 P4

P4 [21] is a domain-specific programming language designed for configuring network devices like switches and routers. It allows network engineers and programmers to define how packets are processed and forwarded. P4 provides a high level of flexibility and programmability, enabling customization of network behavior to suit

specific requirements. It is often used in software-defined networking environments to implement innovative network architectures and protocols. Traditionally, network devices are built using fixed-function chips, following a "bottom-up" approach. P4, however, prefers a "top-down" approach where programmers define features in a P4 program, which is compiled and deployed onto the device, utilizing programmable blocks within compatible chips. The main three points of P4 are :

- **Reconfigurability**: in the field: P4 aims to enable programmers to change the way switches process packets even after deployment. This flexibility allows for adjustments and optimizations based on changing network conditions or requirements
- **Protocol Independence**: P4 seeks to decouple switches from specific network protocols. By doing so, switches can be programmed to handle various protocols without being tied down to any one standard, promoting interoperability and adaptability
- **Target underpinned**: P4 aims to enable programmers to describe packet-processing functionality independently of the underlying hardware specifics. This means that the same packet-processing logic can be applied across different hardware platforms, providing a higher level of abstraction and simplifying development efforts

3.4.2 P4's core processing pipeline

The abstract framework depicted and visible in Fig. 3.3 employs switches to route packets through a configurable parser and multiple stages of match+action processing, which can be arranged in series, parallel, or hybrid configuration. Unlike OpenFlow's rigid assumptions, the P4's model introduces three key innovations. Firstly, it accommodates a programmable parser, allowing for the definition of new headers. Secondly, it allows match-action stages to operate in parallel or series, contrary to OpenFlow's sequential design. Thirdly, the model adopts protocol-independent primitives for composing actions within switches.By establishing a common language (P4), programmers can develop device-agnostic programs that compilers can adapt to different devices. The P4 model has three main blocks:

- **Parser**: Responsible for extracting and parsing headers from incoming packets. It converts raw packet data into a structured format that can be processed by the match-action pipeline.
- **Match-Action Tables**: These tables are where packet processing logic is implemented. They match parsed headers against predefined rules and execute corresponding actions based on the match results. Actions could include forwarding the packet, modifying its headers, or dropping it.

- **Deparser**: The deparser is responsible for reconstructing packet headers after processing. It takes modified packet headers from the match-action pipeline and assembles them back into a raw packet format for transmission

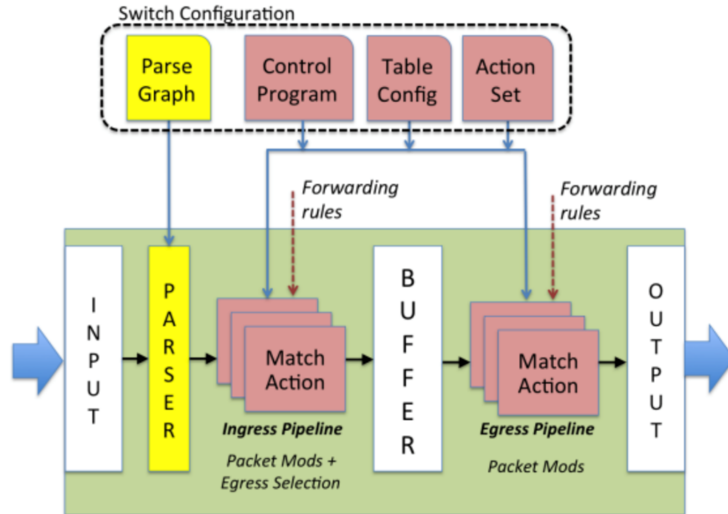

Figure 3.3: P4's forwarding model [21]

These three blocks together form the core processing pipeline in a P4-based network device. The abstract framework of P4 allows for flexible configuration and composition of these blocks to support various network protocols and applications.

3.4.3 P4's Architecture

The possible architectures for P4 have evolved with the language itself, offering more flexibility and targeting diverse hardware/software platforms.

P4v14

Released in 2014, P4v14 represents an earlier generation of the P4 programming language. It defined a single pipeline forwarding architecture specifically designed for devices following the Programmable Internet Switch Architecture (PISA). While this approach laid the groundwork for programmable network devices, limitations arose over time. Firstly, the single pipeline architecture offered limited flexibility. Unlike modern, modular architectures, it constrained packet processing capabilities. Secondly, its dependence on PISA devices restricted its applicability to other hardware platforms. These factors contributed to P4v14 becoming superseded by

newer, more versatile, and portable versions of the language.

P4v16

P4v16, released in 2016, marked a significant improvement in the P4 programming language. Unlike P4v14, this version targets devices that support the PSA [22] (Portable Switching Architecture). Compared to the previous version, there are several changes: architectural independence decoupling the programming model from the underlying architecture, modularity offered by standardized building blocks, and additional features for control plane interaction, such as enhanced table management and telemetry support. The PSA architecture is illustrated in Fig. 3.4.

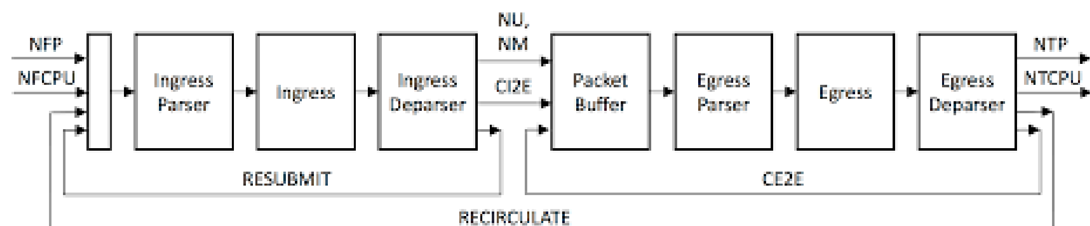

Figure 3.4: Portable Switch Pipeline [22]

3.4.4 eBPF

eBPF (extended Berkeley Packet Filter) [23] is a revolutionary technology embedded within the Linux kernel, enabling you to write sandboxed programs that run directly in kernel space. This grants you unprecedented access and control over the system, unlocking many possibilities. Think of it as a powerful toolkit that lets you inspect, modify, and influence various aspects of your system's inner workings, all while maintaining security and performance. With eBPF, you can:

- **Monitor and analyze network traffic**: Gain deep insights into data flows, identify anomalies, and optimize network performance.
- **Secure your system**: Implement custom security filters, detect malicious activity, and enforce fine-grained access control.
- **Optimize application performance**: Profile and trace application execution, pinpoint bottlenecks, and dynamically adjust system resources.
- **Extend kernel functionality**: Develop custom tools and utilities tailored to your specific needs without modifying the kernel itself.

All the eBPF programs are written in a subset of C and then compiled into a BPF instruction bytecode. Then, thanks to the BPF's virtual CPU single thread, the BPF instructions are translated into machine code. One of the key future of eBPF is its robust program safety. This means developers can confidently write eBPF programs, knowing they'll execute within a secure, controlled environment. This assurance stems from several strict restrictions placed on program behavior:

- **Limited memory access**: eBPF programs can only access specific, pre-defined memory regions, preventing unauthorized interaction with sensitive system areas.
- **Time-bound execution**: Programs operate within defined time constraints, ensuring they don't consume resources or indefinitely block system operations.
- **Resource-constrained environment**: Access to kernel resources is carefully controlled, typically through pre-defined kernel services, further restricting potential harm.

All the actions listed before that eBPF can perform can be executed into two different points inside the Network Linux stack:

- **TC hook point**: Traffic Control (TC) hook point allows eBPF programs to be attached to various packet processing stages within the kernel's Traffic Control subsystem. This enables fine-grained control over packet handling, including traffic shaping, filtering, and classification.
- **XDP hook point**: The Express Data Path (XDP) hook point is located at the earliest possible packet reception stage in the network stack, providing eBPF programs with low-latency access to incoming packets. This allows for ultra-fast packet processing and decision-making, making XDP ideal for tasks such as DoS mitigation, packet filtering, and load balancing.

By leveraging these hook points, eBPF programs can efficiently manipulate network traffic at different packet processing stages, offering unprecedented flexibility and performance in network programmability.

At the heart of eBPF lies a powerful communication mechanism connecting the kernel and user space called tables, sometimes called maps. These tables act as key-value stores, where both keys and values are fixed-size sequences of bits. The width of these elements and the overall capacity of the table are determined at creation, ensuring efficient data management. Both user-space programs and kernel-space code can manipulate tables by inserting, removing, searching for, modifying, or iterating over their entries. However, a key difference exists between how keys and values are exposed in each space. Within the kernel, direct pointers lead to the

raw data stored in the table. In contrast, user-space programs access copies of the data, maintaining security and data integrity.

3.4.5 From P4 to eBPF

Considering the preceding exposition, it becomes evident that the P4 and eBPF coding languages exhibit distinct levels of expressiveness. Nevertheless, a considerable convergence exists in their functionalities, notably within the realm of processing network packets.

The P4 backend for eBPF [24] represents an initial effort towards enabling convergence between the two languages, focusing primarily on translating the shared functionality of packet filtering.

However, this initial effort primarily focused on packet filtering. To address a wider range of use cases, PSA-eBPF [25] emerged. This project aims to translate P4 programs into eBPF programs with additional features and support for the PSA architecture.

This P4 to eBPF compiler presently converts code authored in P4v16 into a specific subset of C in order to be used by tools like clang and/or bcc (the BPF Compiler Collection)for managing eBPF programs. In order to use this tool, we used a particular software switch called NIKSS-vSwitch.

3.4.6 NIKSS-vSwitch

In the age of Network Functions Virtualization (NFV) and advanced networks like 5G, software switches have emerged as crucial building blocks. Acting as virtualized counterparts to hardware switches, they handle data communication between Virtual Machines (VMs) and containers within a network.

NIKSS [26] stands out as a unique P4 software switch embedded directly within the kernel. This tight integration complements the P4-eBPF compiler, allowing for seamless translation of P4 programs into eBPF code for efficient packet processing. NIKSS leverages the Portable Switch Architecture (PSA) as a foundation for forwarding decisions and utilizes an extended Berkeley Packet Filter (eBPF) for advanced packet manipulation.

The PSA-to-eBPF compiler offers two distinct code generation flavors: TC-based and XDP-based. Each caters to different use cases and performance requirements:

- **TC-based**: This approach leverages the existing eBPF TC (Traffic Control)

hook, enabling the implementation of any PSA program. Its advantage lies in universal compatibility, ensuring program functionality across various environments (Fig. 3.5).

- **XDP-based**: This option offloads packet processing to the faster eBPF XDP (eXpress Data Path) hook, delivering superior performance compared to the TC-based design. However, this performance boost comes with certain trade-offs. The XDP-based design currently lacks support for some features like packet recirculation and quality of service (Fig. 3.6).

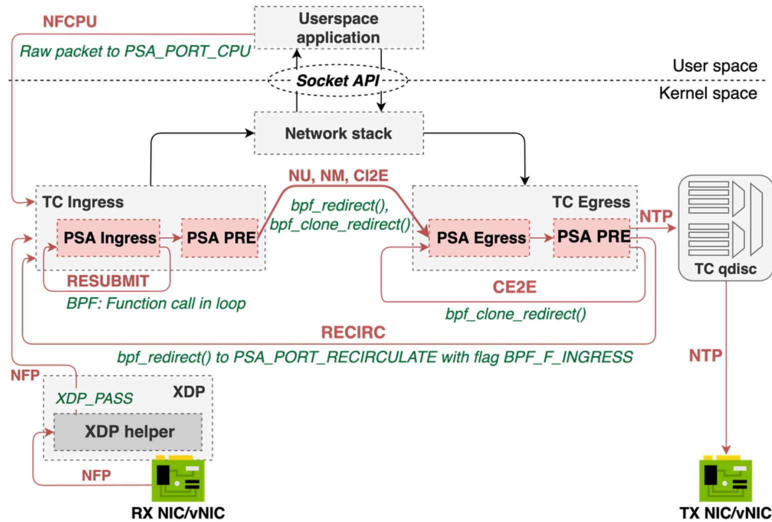

Figure 3.5: PSA-eBPF-TC's design [26]

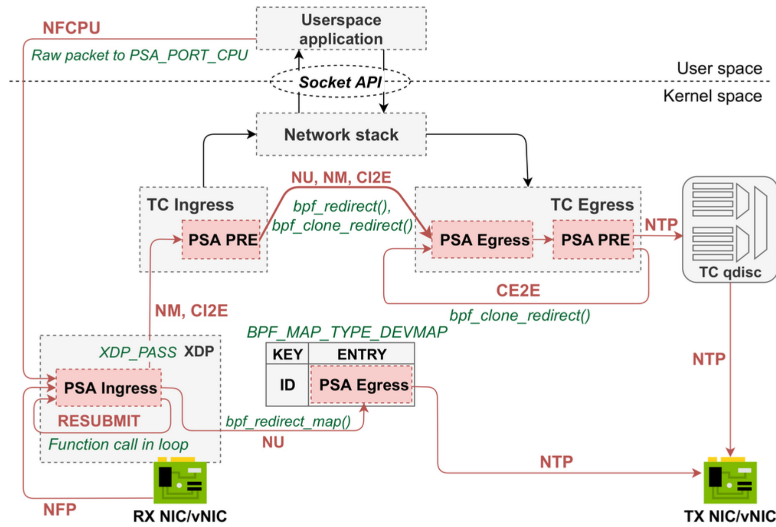

Figure 3.6: PSA-eBPF-XDP's design [26]

While NIKSS demonstrates its effectiveness as a P4 software switch, achieving impressive performance, it's worth noting a slight performance difference compared to native eBPF programming in specific scenarios. This trade-off is often justified by the benefits of in-kernel integration and portability across varied hardware platforms.

It is important to mention that P4-eBPF programs are actually managed only using the NIKSS API without integrating P4Runtime as the controller. This simplifies deployment within the NIKKS environment but restricts integration with broader P4 ecosystem tools. It is also possible to emulate a network infrastructure that leverages NIKSS switches on Mininet but to interact with the eBPF program, it is possible to use only the NIKSS CLI tool. This limitation will be resolved in the future version of the P4-eBPF project.

The figure illustrates the main execution flow of a P4 program translated into eBPF code and injected into the NIKSS switch using specific APIs or CLI tools (Fig. 3.7).

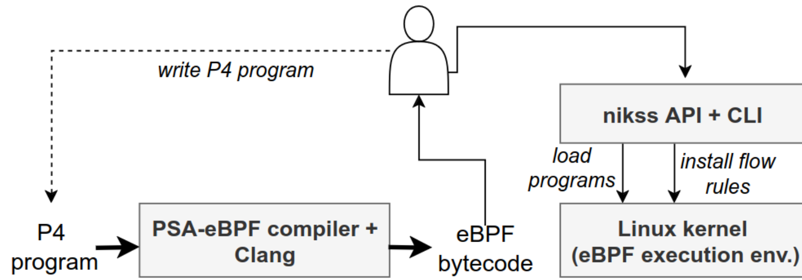

Figure 3.7: NIKSS workflow [26]

NIKSS-vSwitch is a promising solution for constructing high-performance and programmable network infrastructure by leveraging P4-eBPF. Its portability and performance characteristics make it particularly well-suited for cloud and container environments.

An example of a possible implementation of this tool is illustrated in Fig. 3.8. The figure depicts a controller that can manage all the configurations in a network of both physical and virtualized switches that leverage on the same P4 programming code.

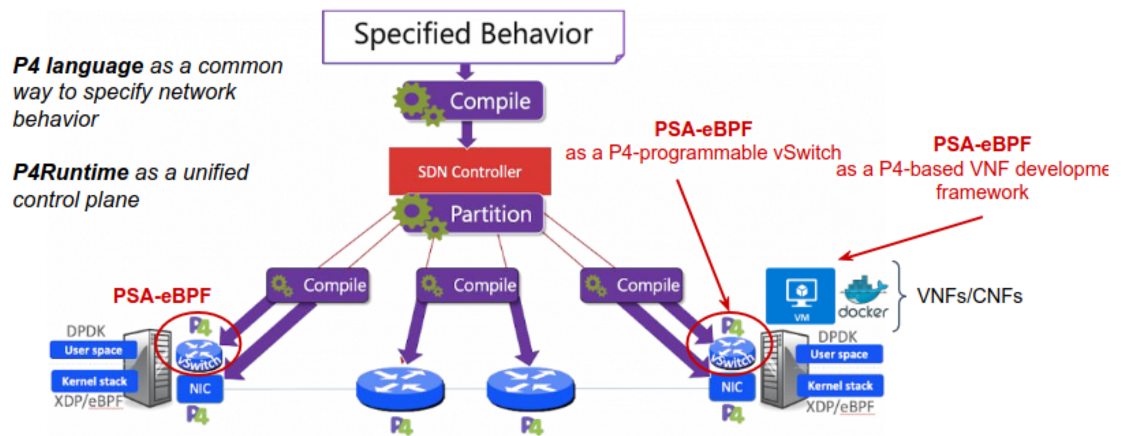

Figure 3.8: PSA-eBPF in the E2E programmable platform [27]

3.5 Natural Language Processing

Natural Language Processing (NLP), a subfield of Artificial Intelligence, bridges the gap between human language and computers. NLP's core objective is to equip computers with the ability to comprehend human language. This involves extracting

and analyzing information from text, allowing computers to process it and perform various tasks. These tasks include text analysis to extract insights and patterns, sentiment analysis to understand opinions, machine translation to convert languages automatically, named entity recognition to identify key figures, text summarization to condense lengthy pieces, question answering to respond to natural language inquiries, text generation to create relevant content, language modeling to predict and produce text, information retrieval to find relevant information within vast collections, and information extraction to automatically gather structured data from unstructured or semi-structured text sources.

To perform this task, the NLP is based on two fundamental components that are Natural Language Understanding (NLU) and Natural Language Generation (NLG):

- **Natural Language Understanding**: The goal of NLU algorithms and models is to comprehend the meaning and intent behind human language, translating human input into machine-readable data.
- **Natural Language Generation**: The objective of NLG algorithms and models is to generate human-like text, translating machine-readable data into human-understandable language.

3.6 Large Language Models

Large Language Models (LLMs) are a subset of deep learning, a field of Machine Learning that studies the use of multi-layer networks to process more complex patterns than traditional machine learning. They are neural networks trained on huge quantities of text data designed to process and generate human language. Due to the ability to generate text, LLMs intersect another branch of Artificial Intelligence: the Generative AI.
The term "large" in LLMs signifies both the expansive size of the training corpus and the extensive parameter count within the model architecture. These models undergo training phases initially on copious amounts of unlabeled text data. This phase, known as pre-training, furnishes the model with a general-purpose understanding of language devoid of specialization to any specific task. To further optimize LLMs for specific applications, two principal approaches are commonly employed:

- **Fine-Tuning**: This method involves tailoring a pre-trained model to a particular task by training it on a smaller, task-specific dataset. Fine-tuning can be executed in two primary ways:
  - **Instruction Finetuning**: This approach entails training the model using

  question and answer pairs, enabling it to comprehend and respond to specific queries effectively.
  - **Classification Finetuning**: In this variation, the model is trained on a labeled dataset, enhancing its ability to classify inputs according to predefined categories.

- **Prompt Engineering**: This strategy leverages various techniques to optimize the input prompt to steer the LLM toward the desired output. Unlike fine-tuning, prompt engineering offers greater flexibility and versatility, facilitating easier and swifter adaptation of LLMs to diverse tasks. This method has been adopted in this thesis in order to adapt the model to our task. In the previous chapter, this paradigm will be analyzed.

3.6.1 An overview of LLM's Architecture

Contemporary Large Language Models (LLMs) predominantly utilize the transformer framework, a sophisticated neural network architecture first introduced in 2017 [28]. The transformer architecture revolutionized natural language processing tasks by effectively capturing long-range dependencies and contextual information without relying on recurrent neural networks (RNNs). LLMs employ self-attention mechanisms to weigh the importance of different words in the input sequence, enabling efficient parallelization and improved performance on various language understanding tasks.

For each element in the input sequence, the self-attention mechanism creates three distinct representations:

- **Query(Q)**: This vector embodies the current element's "question" about its relationship to other elements within the sequence.
- **Key(K)**: This vector functions as a "label" for other elements to compare themselves against, aiding the model in determining their relevance.
- **Value(V)**: This vector encapsulates the actual information or content associated with the specific element.

Within a sequence, a word's internal representation (query vector) is compared to the representations (key vectors) of all other words, including itself. This comparison generates "attention weights," which reflect the level of relevance each word holds in relation to the current word. These attention weights are then used to emphasize specific information. Each word's corresponding context vector (value vector) is weighted based on its attention score. Finally, the weighted context vectors are combined, resulting in a new representation of the word that captures

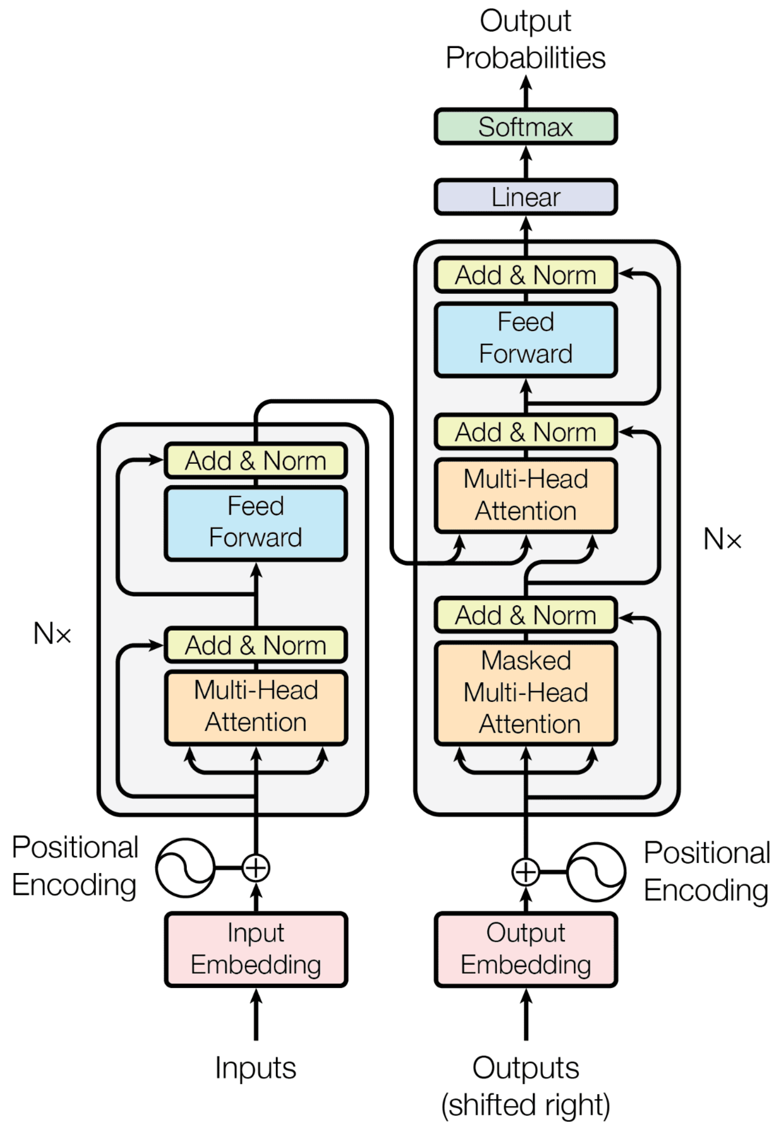

Figure 3.9: Transformer-model architecture[28]

its meaning in the context of the entire sequence. Let's delve deeper into the architecture of transformers by exploring their core building blocks:

- **Embedding Layer**: This layer acts as a bridge between the textual input and the model's internal numerical representation. It transforms individual words into vectors, capturing their semantic meaning and relationships within the context. These vectors, called embeddings, encode crucial information for

the model to process and understand the language.

- **Encoder**: This section plays a pivotal role in processing the embedded input. It aims to extract meaning by analyzing the relationships between words within the sequence. It achieves this through two key sub-components:
  - **Self-attention mechanism**: This mechanism empowers the model to analyze how each word relates to others within the entire sequence, taking context into account. It allows the model to focus on relevant parts of the input, similar to how humans pay attention to specific elements while comprehending language.
  - **Feed-forward network**: This component adds non-linearity to the model, increasing its ability to learn complex relationships and patterns within the data. It enhances the model's capacity to go beyond simple linear connections and capture the intricate nature of language.
- **Decoder**: This section, responsible for generating the output, leverages the encoded information obtained from the encoder. It employs multiple "decoder layers," each containing:
  - **Masked self-attention**: Unlike the self-attention mechanism in the encoder, this one incorporates a mask to prevent the model from "looking ahead" at words that haven't been generated yet. This ensures the model generates the next word solely based on the previously generated context, mimicking the natural flow of language production.
  - **Attention to the encoder output**: This mechanism allows the decoder to "pay attention" to the encoded representation of the input. This crucial step bridges the gap between the input and output sequences, enabling the model to generate text that adheres to grammatical rules and accurately reflects the core message and context conveyed in the original input.
  - **Output layer**: This final layer transforms the internal representation generated by the decoder into the human-readable text output. It essentially translates the model's internal processing into the language we understand.

During sequence generation, the decoder considers two crucial elements: masked self-attention and attention to the encoder's output. Masked self-attention allows the decoder to analyze the relationships between previously generated words while paying special attention to their order. Additionally, the decoder focuses on relevant parts of the encoded input by paying attention to the encoder's output. Combining these insights, the decoder predicts the likelihood of each possible next word in the sequence. The decoder selects the most probable word based on these probabilities,

expanding the generated sequence. This process iterates, with each newly generated word contributing to the context for predicting the next one, ultimately leading to the complete sequence.

Before the embedding layer, the words are split into **tokens** during the text pre-processing stage. The tokenization process is crucial for reducing the gap between human language and the computational area of LLMs. In particular, the pre-processing stage is divided into three main stages;

- **Text Cleaning**: This phase cleans the raw text, removing unnecessary characters like punctuation or special symbols.
- **Normalization**: The text is normalized, converting it into lowercase or lemmatized
- **Tokenization**: This is the phase where the text is split into separate tokens; there are different techniques like word-based, character and sub-word tokenization. In particular, the last one is the most used by most common-shelf LLMs. It consists of splitting the words into smaller units like prefixes, suffixes, or morphemes.

While the Transformer model is a common reference point for modern large language models (LLMs), different models adapt it in various ways. For instance, PaLM [3] utilizes a decoder-only architecture. This means it departs from the traditional encoder-decoder structure and solely relies on the decoder to process the input text and generate the output. This approach offers computational efficiency compared to the traditional design.

3.6.2 Using LLM's APIs

Nowadays, most of the LLMs can be used by utilizing specific APIs offered by big-tech companies like OpenAI and Google.

All these APIs provide tools for users to interact with the LLMs and set some configurations for generating the output. Also, some key concepts are important for correctly using these tools.

- **Prompt**: This is simply the text that the user provides as input to the LLM
- **Model parameters**: These parameters describe the complexity and size of the models, for example, the size of the training data used to train the model.
- **Token**: This is the unit of text the LLM processes. It's an important parameter to consider because the number of tokens can affect the LLM cost. Most LLMs' APIs also specify the allowed number of input and output tokens.

- **Response**: This is the generated response of the LLM.
- **Temperature**: This parameter ranges from 0 to 1. Especially this parameter specifies the "creativity" of the generated output. Generally, a high-temperature value corresponds to a more predictable output.

3.6.3 Prompt Engineering

In line with the discussion in the preceding chapter, various techniques have been developed to structure the input prompt of Large Language Models (LLMs) to achieve the desired output. This is possible thanks to the unique few-shots capabilities in learning a new task. This thesis primarily employs two key prompting techniques:

- **Few-Shots Prompting**: Leveraging the few-shot capabilities of pre-trained LLMs, this method supplies the model with a few examples (shots) to specialize it for a specific task. This approach enhances flexibility and efficiency in handling diverse applications by quickly adapting the model to new tasks with minimal training data.
- **Chain of Thoughts**: This method provides explicit instructions or steps to guide the LLM through the desired task or sequence of actions.

3.7 Gemini

Gemini [4] represents a collection of generative AI models developed by Google. This means that through Gemini, it is possible to process different types of prompts, like images and text, using the relented models and receive them as text output. The families of models are:

- **Gemini**: A list of models that can accept as input text and images and retrieve output text
- **Palm**: A series of models that can accept only text and retrieve as output text
- **Embeddings**: These models are used to convert a text into a numerical vector in high-dimensional space, capturing the semantic relationship and similarities between words
- **Retrieval**: This model aims to answer a question based on provided sources and attribute the answers to those sources.

In this thesis, we have mainly used the models offered by Palm. Palm offers different models based on the task to perform, we mainly focus on two of them:

- **text-bison-001**: It is a model that takes as input a text and generates in output a text. This model is optimized for tasks such as Code Generation, Text Generation, Problem Solving, and Information Extraction.
- **embedding-gecko-001**: This model performs an embedding technique to the input text.

3.8 Langchain

Langchain [29] is an open-source framework that helps to develop applications based on LLMs, which was released in October 2022 as an open-source project. It became very popular among developers, and in early 2024, the first stable version was released. LangChain supports various LLM providers like OpenAi, Hugging Face, and Gemini. LangChain helps to create easier intricate prompts and orchestrate a sequence of steps, guiding LLMs toward your ideal output. The main modules of Langchain are:

- **Modules I/O**: These tools help to structure the prompt using various templates and also to parse the output to a precise format (for example, to a JSON format).
- **Retrieval**: This tool helps to load documents (HTML, PDF, code) from external sources or websites into the prompt of the LLMs. This is a very powerful tool that improves the capabilities of LLMs and their knowledge of something that is not present in the training set. This technique is commonly called **Retrieval Augmented Generation** (RAG).
- **Chains**: Thanks to the chains, you can easily define a step-by-step process to handle any input for LLMs. The chains are like building blocks you can combine to create a clear workflow, leading you to the desired output.
- **Agents**: Agents represent an evolution of chains. While chains follow a predefined sequence, agents leverage the reasoning capabilities of LLMs to determine the best course of action and order of steps dynamically.

In this thesis, we primarily relied on Langchain's Module I/O to effectively structure prompts and parse LLM responses into well-defined JSON schemas. Additionally, we used the Chains to meticulously orchestrate the sequence of steps necessary to achieve the desired outcome.

LangSmith is a companion platform to LangChain, specifically designed to aid in developing and deploying large language model (LLM) applications built on LangChain or any other LLM framework. One of the features that Langsmith offers is the evaluation and monitoring. LangSmith allows you to assess the performance of your LLM applications and monitor their behavior in real-time.

Chapter 4

System's Architecture

In this section, we will illustrate the main components that compose our project, describing their functionalities and tasks.

4.0.1 System's Architecture

The project's architecture is composed mainly of four core blocks:

- **Converter**: This block translates and parses human intent into a machine-readable JSON object for processing by the system's API. The translation and parsing are done with the help of the Large Language Model (LLM).
- **Use Cases**: Here are presented the network functionalities offered to the user:
  - **Load profiling**: Allows selecting weights on switch ports to distribute traffic across the network infrastructure based on user-specified weights.
  - **Firewall ACLs**: Allows specifying IP addresses to be blocked on the specified switch.
  - **Rate limiter**: Allows limiting the bandwidth of all incoming traffic on the specified switch.
- **Application Programming Interfaces**: This block selects the appropriate function based on the goal specified in the JSON object. It then calls the corresponding network function to interact with the underlying network application and fulfill the user's intent.
- **Network Application**: This block represents the implementation of a network application the user interacts with. Our project utilized two different network tools: Ryu-SDN and P4-eBPF.

- **Network Infrastructure**:This layer consists of all data-plane switches. Using the before mentioned network tools, the human intent is applied by setting rules on switches.

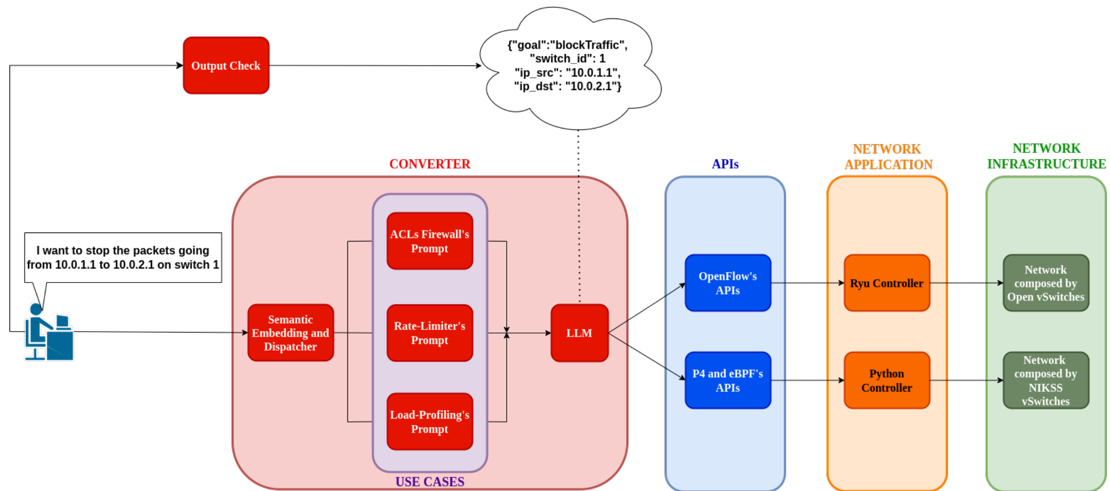

Figure 4.1: Overview of the System's Architecture

4.1 A deep dive into the architecture

This section describes the key features and characteristics of each previously mentioned block:

4.1.1 Converter

The converter is designed to be scalable handling future possible implementation of new network functionalities. It efficiently translates and parses human intent into a well-defined JSON schema, ensuring proper data representation with the usage of LLMs. Also, The converter operates independently of the specific network tools employed. Also the converter has been implanted in order to receive human feedback about the output generated by the LLM, in order to fix eventually misrepresenting of the human intent processed by the LLM. This block is composed of different modules that cooperate to achieve the goal of translating human intent:

- **Semantic Embedding and Dispatcher**: This dispatcher directs the user input to the most suitable prompt template based on semantic similarity. It achieves this by comparing the user's input with the available prompt templates using semantic similarity techniques.

- **Prompt Templates**: These templates provide a structure for feeding the LLM with the user's input. They act as guides, ensuring the input is presented in a way the LLM can understand and generate a meaningful response. They are designed to leverage few-shot learning capabilities by incorporating:
  - **Examples**: Each prompt template includes illustrative examples that demonstrate how an example of user intent is translated into a JSON object.
  - **Steps**: Some complex prompts might benefit from step-by-step instructions. These steps break down the user's intent into smaller, more manageable chunks, further aiding the LLM in understanding the overall goal and translating correctly.
  - **Format instruction**: These instructions specify the expected structure and data types for the user input within the prompt.
- **LLM**: It is the module that processes the input prompt, which contains the user's input. The generated output is in the form of a JSON object which will be passed then to the APIs of the network application.

The system will present the generated LLM output to the user for verification of intent accuracy. Users can then make any required adjustments to ensure the output aligns with their needs.

4.1.2 Application Program Interface

The APIs are custom functions designed to interact with the underlying network applications. Based on the user's intent (parsed by the converter), the appropriate API is dynamically chosen to interact with the relevant network application.

4.1.3 Network Applications

The network applications are the heart of network infrastructure management. They are programmed to execute the network functionalities offered by the system and to interact with the network infrastructure. Depending by the tools utilized this blocks is implemented differently:

Ryu-SDN's scenario

In this scenario, we leverage Ryu-SDN to define the behavior of a centralized controller managing the network infrastructure. This controller sets rules on the network's constituent switches using REST APIs defined within the aforementioned APIs. These REST API calls interact directly with the controller, which processes

the parameters specified within the JSON object and executes the necessary actions to implement the network functionalities intended by the user interacting with the switches using the OpenFlow protocol. Given Ryu's simplicity in defining SDN networks, we aimed to implement a topology-independent controller, meaning it can operate on diverse network topologies. This is achieved by defining a specific routing algorithm that utilizes the concept of adaptive multi-path routing. This approach dynamically determines all possible paths between any two hosts in the network to facilitate communication between those endpoints.

P4-eBPF's scenario

In this scenario, the P4-eBPF compiler is employed to translate P4 code into an eBPF program that is subsequently loaded onto all NIKSS switches within the infrastructure. This approach shares some similarities with the Ryu-SDN scenario in the use of a controller and programmable switches, but with key distinctions. Firstly, the controller is not a traditional one that communicates via the OpenFlow protocol (as in Ryu-SDN). Due to the aforementioned limitations of this still-under-development tool, the controller for simplicity will be primarily Python code, with a series of NKISS CLI calls for each network functionality to directly interact with the switches for managing tables, rules, and meters. Furthermore, this P4-eBPF implementation is not topology-independent. The inherent complexity of the P4 programming language, along with the lack of a standardized controller within P4-eBPF, presents challenges in creating a solution that can adapt to arbitrary network topologies.

Chapter 5

System's Implementation

In this chapter, we will analyze how each module of the system has been implemented, covering various aspects like implementation choices, utilized libraries, how the aforementioned tools [4] have been used and the system's code structure.

5.1 Converter

This paragraph presents the implementation of the converter. The converter's role is to translate human intent into a machine-readable object using the large language model (LLM).

The LLM's primary function is to perform semantic parsing of the intent and convert it into a JSON object.

5.1.1 JSON Objects

To process human intent, the system first converts it into a machine-readable object. This allows the system to interpret the intent and select the appropriate network function from the API system.
Each network function has a well-defined JSON object that summarizes the high-level policies written by the user. All JSON objects share a common structure, specifying the user-defined goal of the network function, the switch id where the intent must be applied, and the parameters needed to implement it.

In particular, the JSON schema of the network functions are:

Firewall ACLs

```
{
  "type": "object",
  "properties": {
    "goal": {
      "type": "string",
      "enum": ["blockTraffic"],
      "description": "the goal of the network function"
    },
    "switch_id": {
      "type": "integer",
      "description": "the id that identifies the switch"
    },
    "ip_source": {
      "type": "string",
      "description": "the ip source address of the traffic to
    block "
    },
    "ip_dest": {
      "type": "string",
      "description": "the ip destination address of the traffic to
     block "
    }
  },
  "required": ["goal", "switch_id", "ip_source", "ip_dest"]
}
```

Load Profiling

```
{
  "type": "object",
  "properties": {
    "goal": {
      "type": "string",
      "enum": ["setWeights"],
      "description": "the goal of the network function"
    },
    "switch_id": {
      "type": "integer",
      "description": "the id that identifies the switch"
    },
    "weights": {
      "type": "object",
```

```
      "additionalProperties": {
        "type": "integer"
      },
      "description": "the weights associated to the releted port
   as key "
    }
  },
  "required": ["goal", "switch_id", "weights"]
}
```

Rate Limiter

```
{
  "type": "object",
  "properties": {
    "goal": {
      "type": "string",
      "enum": ["setRate"],
      "description": "the goal of the network function"
    },
    "switch_id": {
      "type": "integer",
      "description": "the id that identifies the switch"
    },
    "rate": {
      "type": "integer",
      "description": "the rate to apply on the switch"
    }
  },
  "required": ["goal", "switch_id", "rate"]
}
```

These schemes will be passed to the LLM to parse the intent into a well-defined, machine-readable structure. This structure will then be processed by the API system to dynamically choose the relevant function for interacting with the network application and applying the desired functionality.

5.1.2 Prompts

Prompts are essential for the LLM, defining the specific input it processes. In our system, each network function has a dedicated prompt. This approach limits token usage per LLM call, as each request has a token limit. Our case study shows each prompt consumes roughly 550 tokens. Additionally, this approach enhances system scalability. When adding new network functions in the future, only the

appropriate prompt for the specific implementation needs to be defined. This eliminates the need for a comprehensive prompt with all the various possibilities and configurations, mitigating concerns about input token limitations.

All prompts are structured to leverage prompting techniques discussed in previous chapters. Specifically, the prompt follows this structure:

- **Introduction**: This initial section explains the context and desired function of the LLM.
- **Constraints**: Following the CoT prompting technique, this section lists how the human intent should be processed, identifying its goal using specific parameters. Importantly, if the intent remains unrecognized by the LLM, the goal defaults to "noGoal" to prevent generating unsupported functions or misleading intents.
- **Examples**: Following the Few-Shot learning approach, this section lists examples of questions related to the generated output.
- **Schema**: This section specifies the JSON schema to be followed for generating the output.
- **Fixes**: Fixes are added to the prompt if the previous configuration was generated incorrectly. This section includes the last generated output and user-written fix specifications.
- **Query**: This is the user-written human intent, specifying the network functions to be applied to the network.

The prompt used for each network function are:

Firewall ACls

```
1 You are a very good and performant system to parse human intent to
     JSON object. Your duty is to parse a human intent into JSON
    object to block network traffic. \
2
3 [CONSTRAINTS]
4
5 1. If the intent is not really realted to block traffic, set the
    goal to "noGoal"
6 2. If the intent is really realted to block traffic, set the goal
    to "blockTraffic"
7 3. Identify the switch id on which the intent will be applied
```

```
4. Identify in the intent an ip source address and an ip
   destination address
5. If ip_source or ip_dest is not defined, set it to "any"
6: Check if the user specify some fixes to the previous genereted
   output
7. The network configuration should be returned in json format
   based on the following schema definition without additional
   comments:

[SCHEMA]

\n{format_instructions}\n

[EXAMPLES]:

\n{examples}\n

\n{fix}\n


Here is the intent:
{query}"""
```

Load Profiling

```
You are a very good and performant system to parse human intent to
    JSON object. Your duty is to parse a human intent into JSON
   object to set weights on ports in order to distribute traffic
   over the network \
[CONSTRAINTS]

1. If the intent is not really related to setting weights, set the
    goal to "noGoal"
2. If the intent is really realted to set weights, set the goal to
    "setWeights"
3. Identify the switch id on which the intent will be applied
4. Identify in the intent the weights and the releted associated
   port
5: Check if the user specify some fixes to the previous genereted
   output
6. The network configuration should be returned in json format
   based on the following schema definition without additional
   comments:

[SCHEMA]

```

```
\n{format_instructions}\n

[EXAMPLES]:

\n{examples}\n

\n{fix}\n


Here is the intent:
{query}
```

Rate-Limiter

```
You are a very good and performant system to parse human intent to JSON object. Your duty is to parse a human intent into JSON object to set a rate limiter on a specified switch in order to rate limit all the incoming traffic on that switch \
[CONSTRAINTS]

1. If the intent is not really realted to set a rate limiter, set the goal to "noGoal"
2. If the intent is really realted to set a rate limiter, set the goal to "setRate"
3. Identify the switch id on which the intent will be applied
4. Identify in the intent the rate that will be applied to the switch
5. Converts the rate's value to Kbit/s
6. Check if the user specify some fixes to the previous genereted output
7. The network configuration should be returned in json format based on the following schema definition without additional comments:

[SCHEMA]

\n{format_instructions}\n

[EXAMPLES]:

\n{examples}\n

\n{fix}\n


Here is the intent:
```

```
24 {query}
```

An extra prompt will be used only to delete the blocking rule installed in the network. This prompt uses the JSON schema of the blocking traffic, but it has a different goal.

Delete Block Traffic Rule

```
1  You are a very good and performant system to parse human intent to
     JSON object. Your duty is to parse a human intent into JSON
     object to delete a rule that block traffic. \
2
3  [CONSTRAINTS]
4
5  1. If the intent is not really realted to delete a rule, set the
     goal to "noGoal"
6  2. If the intent is really realted to delete a rule that block
     traffic, set the goal to "deleteFlow"
7  3. Identify the switch id on which the intent will be applied
8  4. Identify in the intent an ip source address and an ip
     destination address
9  5. If ip_source or ip_dest is not defined, set it to "any"
10 6: Check if the user specify some fixes to the previous genereted
     output
11 7. The network configuration should be returned in json format
     based on the following schema definition without additional
     comments:
12
13 [SCHEMA]
14
15 \n{format_instructions}\n
16
17 \n{fix}\n
18
19 [EXAMPLES]:
20
21 \n{examples}\n
22
23
24 Here is the intent:
25 {query}
```

The examples serve as question-and-answer pairs that assist the LLM in generating the desired output. These examples provide the LLM with concrete scenarios and expected responses, helping it understand the desired format and intent of the generated output.

Here is an example:

```
"question": "block traffic from 10.1.1.1 to 10.1.1.2 on switch 4",
 "answer": """{
        "goal": "blockTraffic",
        "switch_id": 4,
        "ip_source": "10.1.1.1",
        "ip_dest": "10.1.1.2"
    }"""
```

5.1.3 Converter's Program

This paragraph will illustrate how the program for the converter was implemented using the LangChain framework.

LangChain proves valuable in this case for two key functionalities: structuring the prompt and defining a series of actions for the LLM to follow and ensuring accurate output generation. The program utilizes the following LangChain modules:

- **Prompt Templates**: These templates facilitate building the prompt by defining the variables to be included. In our example, these variables involve the user's intent, JSON schema, examples, and fixes.
- **Chains**: Chains specify the precise order of steps for the LLM to process the input prompt

Using Prompt Templates is simple: you pass the input text, defining the internal variables and their corresponding values.

Chains define the actions and prompts sent to the LLM. In our case, the chain is structured in the following way:

```
chain = (
    {"query": RunnablePassthrough()}
    | prompt
    | ChatGoogleGenerativeAI(model="llm-model")
    | parser
    )
```

This chain works as follows:

1. **"query": RunnablePassthrough()**: This step passes the user's intent (query) without modification.
2. **prompt**: This step concatenates the user's intent with the pre-defined prompt template.

3. **ChatGoogleGenerativeAI(model="llm-model")**: This step sends the combined prompt to the LLM for output generation.

4. **parser**: This step parses the LLM's response according to the predefined parsing logic.

For each intent written by the user, this intent must be inserted in the right prompt that will be given in input to the LLM. This is achieved by routing the input to the correct prompt by semantic similarity. This implies that all the prompts and the user's intent are embedded into a common vector space. This allows the system to calculate the semantic similarity between the user's intent and each available prompt, and route the input to the most semantically similar prompt for specialized processing by the LLM. The key idea is to embed both the user's intent and the pre-defined prompts into a shared vector space. Vectors are mathematical representations of text that capture meaning. By calculating the similarity (often using cosine similarity) between the user's intent vector and the prompt vectors, you can find which prompt is semantically closest to the intent the user expressed. Once the closest matching prompt is identified, the system sends the user's intent to that specific prompt. Each prompt likely has its own structure and output parser to process the user's intent in the best way for that specific configuration type.

Algorithm 1 Converter's Program

```
1: Set up semantic similarity embedding templates for routing prompts
2: Define the JSON schema of each network configuration
3: Define examples of input-output pairs for each type of network configuration
   task
4:
5: Function RoutePrompt(userInput):
6: Embed the user input using semantic embedding
7: Calculate similarity between user input and prompt templates
8: Select the most suitable prompt template to fulfill the user intent based on
   the semantic similarity
9: Set up the chain composed by the user intent, the selected template, the LLM
   to process the intent, and the parser
10: return The generated chain
11:
12: Function Main():
13: Display available network configuration options to the user
14: Prompt the user to specify their intent
15: Initialize a variable for storing possibly fixes
16: repeat
17:     Invoke the RoutePrompt function to generate the chain
18:     Invoke the chain to generate the network configuration
19:     Display the generated configuration to the user
20:     Prompt the user to confirm if the generated configuration is correct
21:     if the user confirms then
22:         Break the loop
23:     else
24:         Prompt the user for fixes and store them
25:     end if
26: until the user confirms the generated configuration
27: Output the generated network configuration to the APIs
```

As it's possible to see from the pseudocode below, the user is always requested to check if the generated output is correct. If not, the user is asked to insert the fixes. At this point, a new chain is generated, including the fixes specified by the user in the prompt.

5.2 Application Program Interface

In this section, the implementation of the Application Programming Interface (API) is presented. The purpose of this layer is to read the desired network functionalities parsed from the JSON object and call the appropriate function to interact with the underlying network application. The implementation of this layer depends on the underlying network application used.

5.2.1 OpenFlow's API

Using Ryu-SDN as the underlying tool, the network configurations are communicated to the Ryu Controller using REST API calls. This is possible thanks to a REST API interface implemented on the Ryu Controller, which allows the controller to receive, process, and respond to HTTP requests. Specifically, the parameters of the parsed JSON object are sent to the controller via a REST call that starts a precise function within a Ryu application to implement the network function. Ryu permits this function calling through the use of decorators and a built-in web server. Ryu has a built-in web server (based on Web Server Gateway Interface(WSGI)) that acts as the interface for REST API calls. This web server listens for HTTP requests (GET, POST, PUT, DELETE) on specific endpoints.

After the LLM generates the JSON object, the system will analyze the 'goal' field to identify the network functions requested by the user. It will then execute the appropriate REST API calls to transmit the necessary parameters to the Ryu Controller, fulfilling the user's request.

5.2.2 P4-eBPF's API

Due to the limitations of emulating P4-eBPF within a Mininet environment, the NIKSS switch 'controller' is implemented as Python code that interacts with the switches by calling a series of CLI instructions. This approach resembles Ryu's methodology: after generating the JSON object and determining the desired network functions, the system executes a routine of CLI commands to interact with the switches and implement those functions.

5.3 Network Application: Ryu-SDN

In this section, the implementation of the Ryu Controller is presented, describing how the controller applies the concept of adaptive multi-path routing and how it manages incoming requests to realize the network function defined by the user.

5.3.1 Ryu Controller's implementation

This controller is inspired by the work of Wildan Maulana Syahidillah [30]. Its purpose is to discover all available paths between two hosts and distribute network traffic across these paths. The distribution is determined by weights assigned to the outgoing ports of the switch from which the paths originate. We have customized this solution for our project, where the weights on the ports are utilized to distribute traffic according to the user-specified weights. These weights are defined using the load profiling function.

This controller utilizes event handlers provided by the Ryu framework to retrieve information about the network topology. This information is essential for calculating the different paths in the topology. These handlers are functions specifically designed to respond to various network events, including:

- **Network Switches**:Events related to switch connection or disconnection.
- **Packets**: Events triggered by incoming or outgoing packets.
- **Topology changes**: Events that signal changes in the network topology, like link addition or removal.

Within this specific implementation, event handlers play a crucial role in dynamically discovering and managing the network topology:

- **EventOFPPacketIn**: This handler is triggered whenever a switch receives a packet that it cannot process due to a lack of matching flow rules.
- **EventSwitchEnter**: This handler is invoked when a new switch connects to the controller. Upon receiving this event, the controller retrieves information about the switch's ports and updates its internal switch database. This information is essential for path calculation.
- **EventLinkAdd**: This critical handler is triggered whenever a new link is established between two switches. When this event occurs, the controller extracts information about the involved switches from the event data. It retrieves the specific ports used for the newly established link and stores this information for future reference. The controller updates its internal representation of the network topology to reflect the newly added link. This updated topology is then used for path calculations when needed.

The Depth-First Search (DFS) algorithm is used to find all connections between two devices on the network. It starts at the source device and explores every available path one at a time, going as far as possible on each branch before going back and trying another direction. Having identified all potential paths between two network

endpoints, only the shortest path is selected for each possible path. However, a mechanism is still required to distribute traffic across these shortest paths in accordance with a pre-defined weight scheme. OpenFlow's group tables provide a suitable solution for this task. Group tables function as centralized entities within OpenFlow switches, allowing the specification of multiple forwarding options (buckets) for incoming packets. Each bucket can define a specific action, such as directing the packet to a particular outgoing port. In this specific implementation, group tables are utilized for the following purpose:

- A single group is established, encompassing numerous buckets. Each bucket represents a unique path identified between the source and destination hosts.
- An associated weight is assigned to each bucket, reflecting the desired preference for utilizing that particular path for traffic distribution.
- Upon receiving a packet destined for the target host, the switch consults the pre-configured group table. Based on the assigned weights, the switch probabilistically selects a bucket and executes the corresponding action

This approach can be likened to a weighted random selection process. Each path (represented by a bucket) possesses a weight that determines its relative likelihood of being chosen. The selection process effectively distributes traffic across the available paths in a manner proportional to their assigned weights. The main functions can resumed as follows:

Algorithm 2 List Available Paths

```
1: Initialize an empty list to store available paths
2: for each switch containing a path do
3:     Initialize an empty list to store ports with a path
4:     for each port in the switch do
5:         if the port contains a path then
6:             Add the port to the list of ports with a path
7:         end if
8:     end for
9:     if there are multiple ports with a path in the switch then
10:        Create a group table flow
11:        for each port with a path in the switch do
12:            Create a bucket
13:        end for
14:    else
15:        Install a normal flow for the switch
16:    end if
17: end for
```

In particular, each bucket is defined by mainly:

- **Bucket Weight**: An integer value that influences the probability of a bucket being chosen when a packet matches the group's match criteria. Higher weights increase the likelihood of selection.
- **Actions**: A list of OpenFlow actions that define what should be done with the packet if this bucket is chosen. In our case, the action is typically to forward the packets on the port associated with the selected path.

Initially, the bucket's weights are all equal for the same group table.

Each flow rule defined in OpenFlow is typically associated with a match field. The match field defines which types of traffic the rule applies to. In this implementation, two match fields are defined:

- **Match IP**: This defines the IP traffic flow between the two hosts for which all paths have been calculated, specifying the matched IP fields (e.g., source IP, destination IP)
- **Match ARP**: Similar to the previous match, but targeting ARP packets.

In this way, all the IP packets that match the defined criteria are redirected to follow precise pre-calculated paths. ARP packets are used to create an association between IP addresses and the corresponding MAC addresses of hosts. In the preliminary stage of communication, these packets are typically sent in broadcast mode. These packets are also selected and forced to follow the calculated paths to avoid saturating the network and creating loops.

At the beginning, when the switch and host start communicating, the event handler of EventOFPPacketIn is triggered to manage packets without matching rules. These packets are sent to the controller, which starts calculating all the necessary paths to make the packets reach their destination by installing the corresponding flow rules and group tables.

5.3.2 Network Functions

Ryu controller can receive REST API calls thanks to its built-in web server. This call will be served by a precise function to manage and fulfill the request.

There is a function for each request network function that we want to implement.

- **Block Traffic**: This feature is implemented by defining a function that serves a POST API call. The request body specifies the following parameters:

  - **IP source**: The IP address of the traffic source to be blocked.
  - **IP destination**: The IP address of the traffic destination to be blocked.
  - **Switch ID**: The switch's identifier on which to block the traffic.

  The blocking mechanism involves installing a flow rule on the specified switch. This rule targets packets with the specified IP addresses using a match variable and defines an empty action field to prevent their forwarding.

- **Load Profiling**: This feature is implemented by a function designed to handle a PUT API call. The request body includes the following parameters:
  - **Switch ID**: Unique identifier of the target switch.
  - **Weights**: A list of port-weight pairs specifying a weight for each port.

  The weights specified by the user are then updated on the switch with that related port, updating the bucket's weight of each group table.

- **Rate Limiter**: Rate limiting on an SDN switch is achieved by defining a meter for each switch using a meter mod. An OpenFlow meter mod is a message sent from the controller to the switch specifically to configure a meter. This meter acts as a traffic regulator. It monitors the data rate (kilobits per second or kbps) of packets flowing through the switch and can be configured to take action if the rate exceeds a predefined threshold. The meter mod allows you to specify the desired rate limit in kbps. This value controls the maximum allowed traffic rate on the switch. An API call (PUT request) can dynamically update the rate limit. The request body typically includes two parameters:
  - **Switch ID**: This uniquely identifies the target switch where the meter needs to be updated.
  - **Rate**: This is an integer value specifying the new desired bandwidth limit in kbps.

  Upon receiving a valid PUT request, the controller sends a meter mod to the corresponding switch, updating the configured rate limit.

5.4 Network Application: P4-eBPF

This paragraph explains how a P4 program is structured. The program will be translated into an eBPF program using the P4-eBPF compiler. We will also specify how to interact with the tables of the translated P4 program. This implementation is designed for a network topology with four upper switches and five lower switches, where each lower switch connects to two hosts.

5.4.1 P4 program

This P4 program controls incoming traffic (ingress) using forwarding tables, access control lists, and meters. It first applies a forwarding table based on the destination Ethernet address to decide how to route the packet. The table can choose a deterministic port or probabilistically select a port among the port options with given weights. Then, an ACL table based on the source or destination IP address can drop packets if needed. Finally, a meter limits the overall traffic rate. The tables can be populated using the NIKSS CLI tools, specifying which parameters must be added to the table entries to trigger the related action to the table.

In the case of leaf and spine topology, the tables are configured to forward the packets correctly to reach all the destinations. In particular, if the packets arrive at a lower switch and the packet's destination is for one of the hosts directly connected to the switch, the packets will be forwarded directly to the host. Instead, the packets that arrive on a lower switch have as their destination another host not directly connected; the packets are forwarded on one of the possible ports to the upper switches. The section of the port is made using a weight random choice algorithm implemented in the P4 code. Resuming the tables and related actions are:

- **Forwarding table**:
  - **Key**: This table uses the destination MAC address for exact matching.
  - **Actions**:
    - **Forwarding**: Sends the packet to the specified output port.
    - **Random selection**: This action performs a weighted random forwarding between four ports. Each port has an associated weight, and the packet is sent to a port based on a random value and these weights. Counters are maintained for each port to track the number of packets sent.
  - **Access Control List table**:
    - **Key**: This table uses the source/destination MAC address for exact matching.
    - **Actions**:
      - **Drop**: Drops the packet.

These tables are applied in the ingress control block in the following order:

1. **Access Control List table**: here, the source/destination MAC address is checked. The packet is dropped if the source MAC address is found in the table with a drop action. Otherwise, it's allowed to continue processing.

2. **Forwarding table**: if a matching entry is found based on the destination MAC address, the corresponding action is taken (forwarding, random forwarding).

The random choice selection of the port is based on the algorithm of the weighted random choice. Here is how it works:

Algorithm 3 Weighted Random Choice

1: Define the weights for each port: w_1, w_2, w_3, w_4 (32-bit values representing the weight of each port)
2: Generate a random value ($rand$) between 1 and 16 (inclusive) using a platform-specific random number generator function
3: Calculate the total weight: $total_weight = w_1 + w_2 + w_3 + w_4$
4: Define probability ranges for each port:
   - port1: 0 to w_1 (exclusive)
   - port2: w_1 (inclusive) to $w_1 + w_2$ (exclusive)
   - port3: $w_1 + w_2$ (inclusive) to $w_1 + w_2 + w_3$ (exclusive)
   - port4: $w_1 + w_2 + w_3$ (inclusive) to 16 (inclusive)
5: Determine the output port based on the generated random value ($rand$):
6: **if** $rand \leq w_1$ **then**
7:     output port1
8: **else if** $rand \leq w_1 + w_2$ **then**
9:     output port2
10: **else if** $rand \leq w_1 + w_2 + w_3$ **then**
11:     output port3
12: **else**
13:     output port4
14: **end if**

Once a port is chosen, the meter is consulted to check if the port allows forwarding for the current packet.

In order to implement the rate limiter function, the code defines a single meter named **meter** within the ingress control block. The properties are:

- **Meter Type**: specifies that the meter measures the number of bytes processed.
- **Meter Function**: the code uses the meter within two actions: forwarding action and random choice forwarding. Packets exceeding a certain byte quota are dropped. The meter quota can be settled through the NIKSS CLI tool.

5.4.2 NKISS-CTL

NIKSS-CTL is a command-line tool likely used to interact with NIKSS (Native In-Kernel P4-programmable Software Switch). The main commands are to manage the tables and the meter's quota.

- **Tables**: These define how packets are forwarded within the switch. NIKSS-ctl provides commands to manage tables, including setting the default pipeline ID, table name, action to be taken on packets, and additional action parameters to specify mainly on which ports the packet must be sent.
  - **Example command**: *nikss-ctl table default set pipe ID TABLE_NAME action ACTION [data ACTION_PARAMS]*
- **Meter**: These control the rate of packets flowing through the switch. NIKSS-ctl allows updating meters by specifying the pipeline ID, meter name, index, Peak Information Rate (PIR), Peak Burst Size (PBS), Committed Information Rate (CIR), and Committed Burst Size (CBS).
  - **Example command**: *nikss-ctl meter update pipe ID METER_NAME index INDEX PIR:PBS CIR:CBS*

5.5 Network Infrastructure

The network infrastructure is emulated using Mininet. Mininet acts as a valuable tool for crafting and exploring Software-Defined Networking (SDN) solutions. This stems from its ability to construct virtual network environments that support OpenFlow, along with its interactive features that facilitate testing and demonstration of network functions and applications. Additionally, Mininet's extensible framework allows for defining custom switches, enabling the creation of specialized network switches like the NIKSS switch, which can be further enhanced by loading eBPF programs.

Chapter 6

Evaluation

This chapter describes the experiments that were conducted to evaluate the system's performance and the resulting data. It focuses on two key areas:

- **Network Applications**: This section analyzes the performance of various network functions (ACLs, Load Profiling, Rate Limiters). In the case of the Ryu Application, the performance of the adaptive multi path routing is also evaluated.

- **Large Language Model (LLM) Effectiveness**: This section examines the effectiveness of the LLM in translating user intent into a well-structured JSON format.

6.0.1 Testbeds

The performance of the network applications is analyzed using three different network topologies. These topologies are emulated using Mininet, a popular network emulator. In the case of the application developed with Ryu, the switches are emulated through the usage of Open vSwitch (OVS) switches. OVS are software-based switches that implement the OpenFlow protocol. This protocol allows Ryu, acting as an SDN (Software-Defined Networking) controller, to communicate with and program the virtual switches in your Mininet network. This creates a controllable environment for testing and development of Ryu applications. Regarding the P4-eBPF application, the switches are emulated using the NIKSS vSwitches[3.4.6].

- **First topology**: This is a simple network with two hosts, four switches, and two primary paths for host communication [Fig. 6.1].

- **Second topology**: Similar to the first topology, but with four potential paths between the hosts [Fig. 6.2].

- **Third topology**: A leaf-and-spine architecture with four upper switches, five lower switches, and two hosts connected to each lower switch Fig. 6.3.

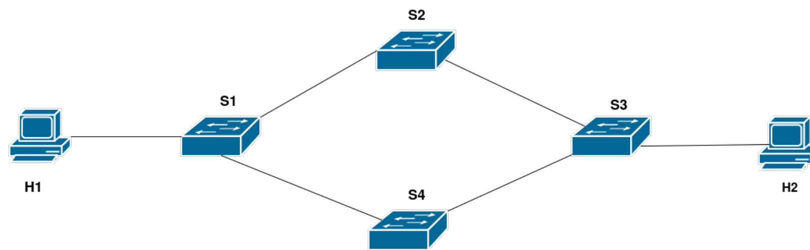

Figure 6.1: First network topology: Two multi-paths topology

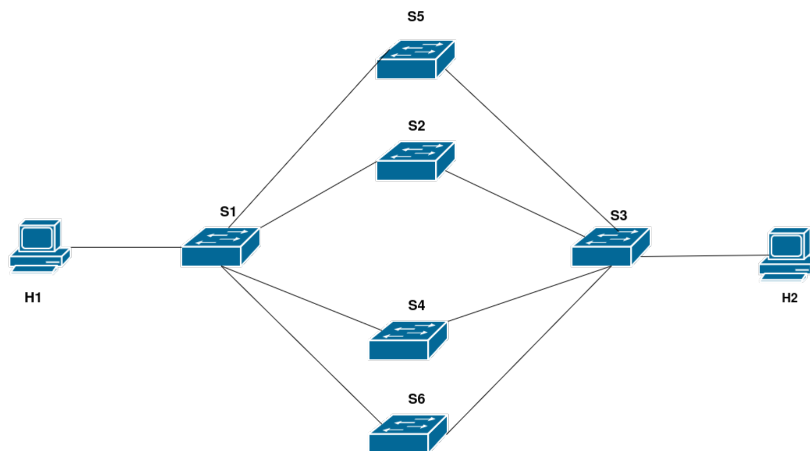

Figure 6.2: Second network topology: Four multi-paths topology

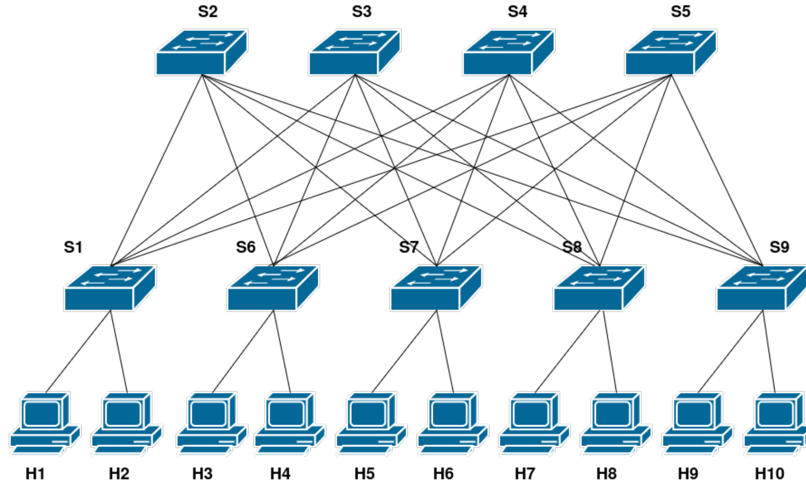

Figure 6.3: Third topology: Leaf and spine DC

6.1 Network Application: Ryu-SDN

This section evaluates the performance of the adaptive multi-path routing and the performance of network functions (ACLs, Load Profiling, Rate Limiters) within the context of network applications developed using the Ryu-SDN framework.

6.1.1 Adaptive multi-paths routing

In this section, we assess the performance of the adaptive multi-path routing within the context of Ryu-SDN. The evaluation encompasses the determination of host reachability and the time required for path installation. The experiments are conducted across various network topologies[Fig. 6.1,6.2,6.3], with the "pingall" command executed within the Mininet environment to simulate network communication. The findings from these experiments are presented in the subsequent table:

Topology	Reachability	Average Time (ms)
Two multi-paths topology	100%	5.8 ms
Four multi-paths topology	100%	7.3 ms
Leaf and spine topology	100%	12.8 ms

Table 6.1: Description of topologies with reachability and average time

6.1.2 Firewall ACLs

In this section, we show the results of the network functionalities when blocking the IP traffic. The feature is evaluated in different use cases focusing only on the third topology (Fig. 6.3).

Assuming the correct translation of the human intent, the human intents applied to the network are:

- *"Can we block communication from 10.0.0.1 (H1) to any destination on switch 1?"*
- *"Block all outgoing traffic destined for 10.0.0.3 (H3) from 10.0.0.1 (H1) on switch 1"*
- *"Could you block all communication from 10.0.0.2 on switch 8?"*
- *"Prevent all incoming packets from 10.0.0.5 on switch 8."*

The experiments involve pinging other hosts and verifying packet drops. This emulation is conducted using the "ping" command. The table 6.2 presents the outcomes regarding packet drops and the duration required to apply rules on the Ryu controller upon the arrival of incoming intents:

Intent	Packets Dropped / Packets Sent
First Intent	50/50
Second Intent	50/50
Third Intent	50/50
Fourth Intent	50/50

Table 6.2: Results of Packet Drops and Rule Application Time on the Ryu Controller

6.1.3 Rate Limiter

This section analyzes the correct application of the rate limiter function in the network. Considering the third topology, the following intentions are applied to the network:

- *"Can you cap the bandwidth to 1 Gbit/s on switch 1?"*, shown in Fig. 6.4
- *"Set a maximum bandwidth of 0.5 Gbit/s on switch 2, please."*, shown in Fig. 6.5

- *"We need to limit the bandwidth to 7 Gbit/s on switch 3. How can we achieve this?"*, shown in Fig. 6.6

The tool "iperf3" has been used to evaluate whether the bandwidth limits have been respected on the switches, generating traffic among two hosts and checking the throughput. The following tables show the results:

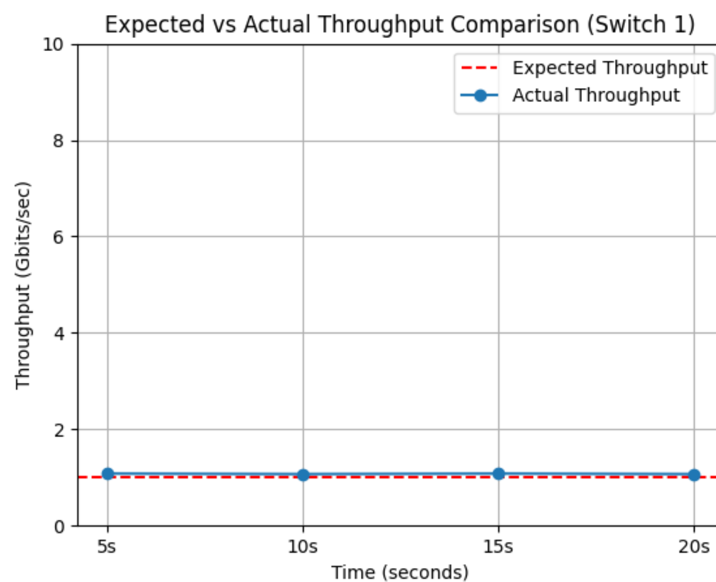

Figure 6.4: Throughput on switch 1

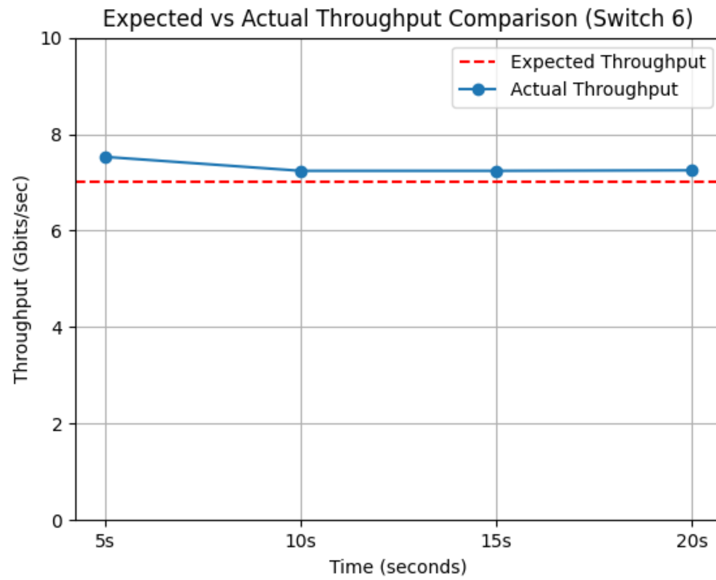

Figure 6.5: Throughput on switch 2

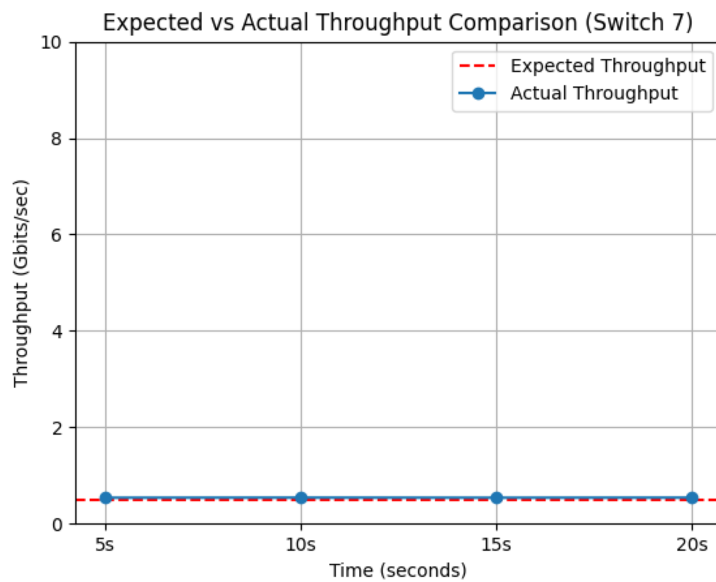

Figure 6.6: Throughput on switch 3

6.1.4 Load Profiling

In this paragraph, we provide the network function's evaluation of distributing traffic among different paths when different weights are specified for each port. The experiment mainly consists of setting the weights on the different switches' ports and examining if the amount of packets sent respects the weights. The traffic in the network is generated using the "iperf3" command that simulates realistic network traffic.

Assuming the correct translation of the intent by the converter, the human intents translated are:

- *"Modify traffic distribution on switch 1 by assigning weights 25, 25, 25, and 25 for ports 1,2,3 and 4, respectively."*, shown in Fig. 6.7
- *"On switch 6 distribute the traffic accordingly to the weights: 70 on port 1, 10 on port 2,10 on port 3, 10 on port 4"*, shown in Fig. 6.8
- *"On switch 7 set the weights 25 on port 1, 25 on port 2, 25 on port 3, 25 on port 4"*, shown in Fig. 6.9
- *"On switch 8 the weights are 30 on port 1, 30 on port 2, 30 on port 3, 10 on port 4"*, shown in Fig. 6.10
- *"On switch 9 the weights must be 50 on port 1, 20 on port 2, 20 on port 3, 10 on port 4"*, shown in Fig. 6.11

The experiments confirmed that the weight settings were successfully applied to the switches, modifying the values of all bucket weights in each group table. However, issues were observed with traffic distribution according to the weights. For instance, generating traffic between H1 and H3 did not result in a split that respected the configured port weights. In particular, the traffic was distributed only on one path respected the other.

From the documentation, it emerged that this behavior can be attributed to the specific way Open vSwitch (OVS) handles weights. OVS employs a hashing algorithm that considers both the bucket ID and its weight. It calculates a "score" for each live bucket by multiplying the hash of flow data with the bucket weight. The bucket with the highest score is then chosen to handle the packet.

This method might not perfectly match the weight configuration, especially in scenarios with limited traffic diversity. So, the next experiments were conducted to enhance traffic diversity using "iperf3" to introduce variations in the traffic flow. This can involve changing ports. This can be achieved using "iperf3" with

the -P flag followed by the number of parallel TCP streams to generate. In the context of iperf, the -P flag specifies the number of parallel TCP streams to generate. It allows you to simulate multiple concurrent connections between the sender and receiver, effectively increasing the diversity of your network traffic. The tables 6.7,6.8.6.9,6.10,6.11 show the results of the experiment setting the value of concurrent flows to 20.

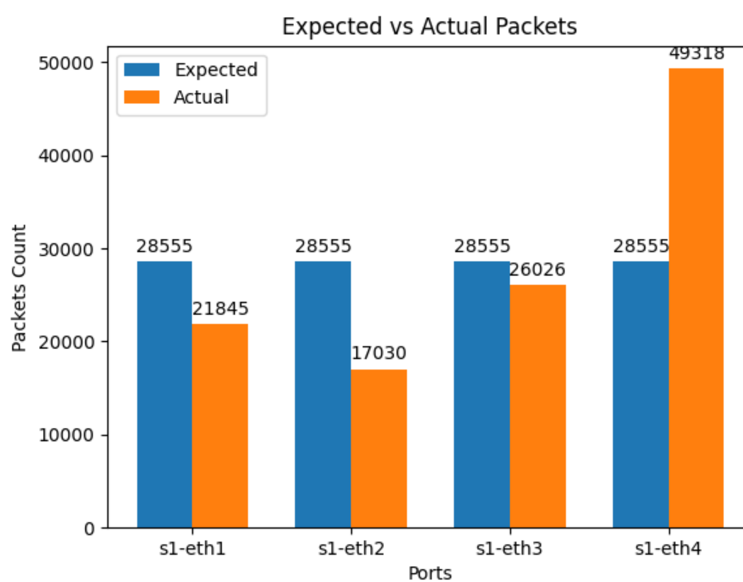

Figure 6.7: Load Profiling on switch 1

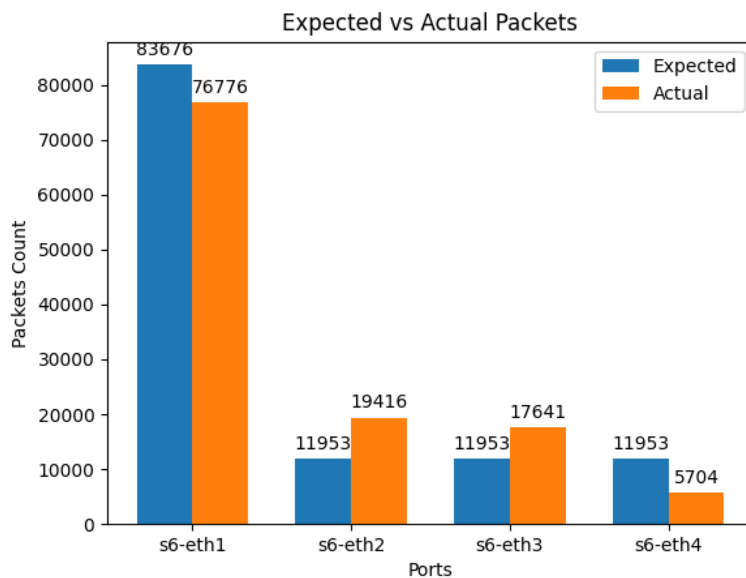

Figure 6.8: Load Profiling on switch 6

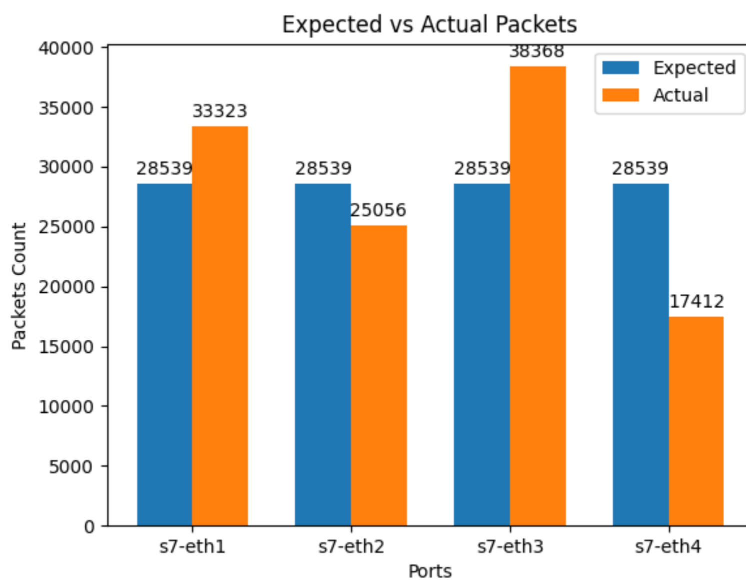

Figure 6.9: Load Profiling on switch 7

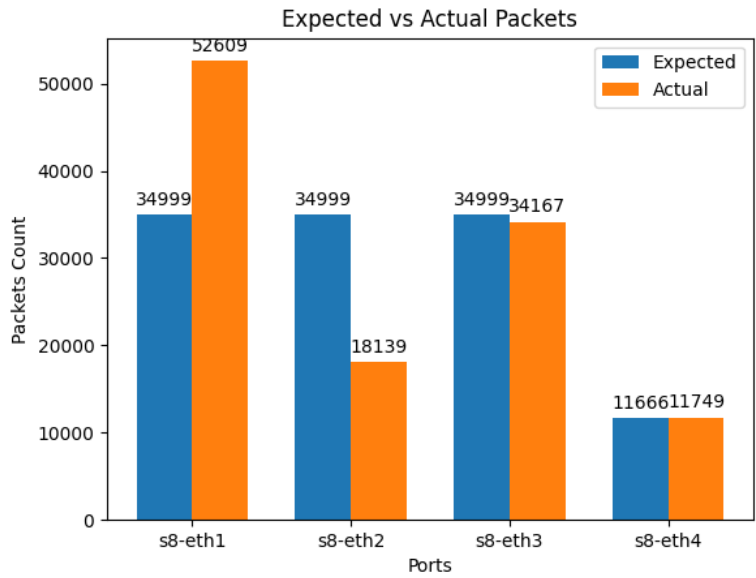

Figure 6.10: Load Profiling on switch 8

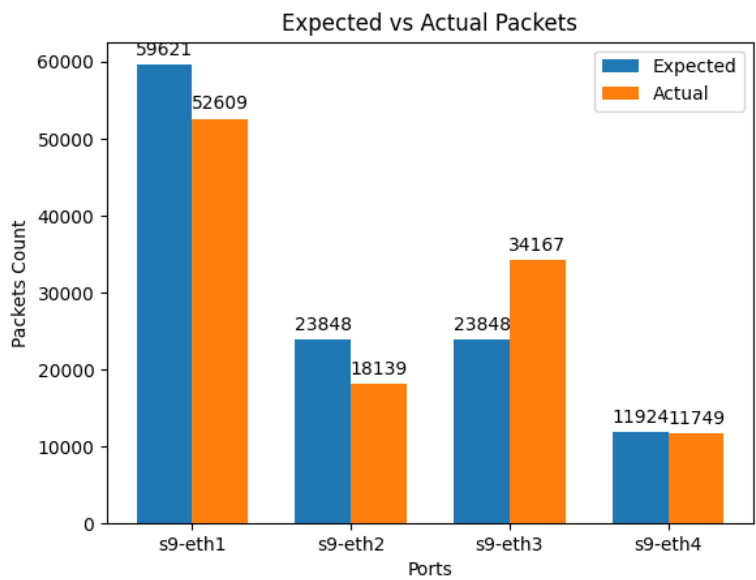

Figure 6.11: Load Profiling on switch 9

6.2 Network Application: P4-eBPF

All experiments are executed in a scenario identical to the leaf and spine topology [Fig. 6.3] used for evaluating the Ryu application.

6.2.1 Firewall ACLs

This section investigates the efficacy of a Ryu application's IP traffic blocking functionality. The evaluation methodology involves applying various intents to target switches and verifying the subsequent packet drops on those switches.

For this experiment, we focused on the following intents as input for our model:

- *"Can we block communication from 10.0.1.2(H2) to any destination on switch 1?"*
- *"Stop all data transfer between 10.0.1.1(H1) and 10.0.6.4(H4) on switch 1."*
- *"On switch 8 block all the traffic incoming from 10.0.1.1(H1)"*
- *"Stop all packets destined for 10.0.7.6 on switch 8."*

The results are reported in the table:

Intent	**Packets Dropped / Packets Sent**
First Intent	50/50
Second Intent	50/50
Third Intent	50/50
Fourth Intent	50/50

Table 6.3: Results of Packet Drops

6.3 Rate Limiter

In this section, we show the performed rate limiter evaluation function using P4-eBPF. The experiment has been conducted applying the following intents:

- *"Set a maximum traffic rate of 1 Gbit/s on switch 1."*,,shown in Fig. 6.12
- *"On switch 6 the traffic must be limited to 1 Mbit/s"*, shown in Fig. 6.13
- *"Is it possible to limit the traffic to 3 Gbit/s on switch 7?"*,shown in Fig. 6.14

To check the bandwidth, the "iperf3" command was used to create traffic flows among hosts. The results are displayed in the following tables:

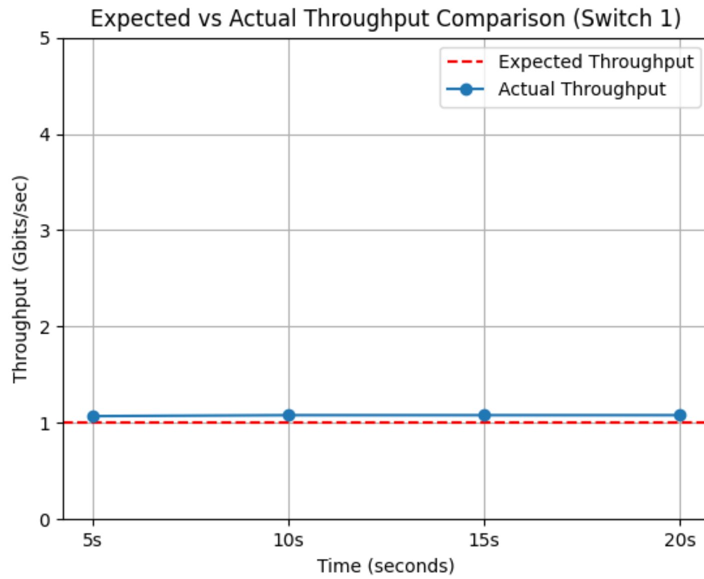

Figure 6.12: Throughput on switch 1

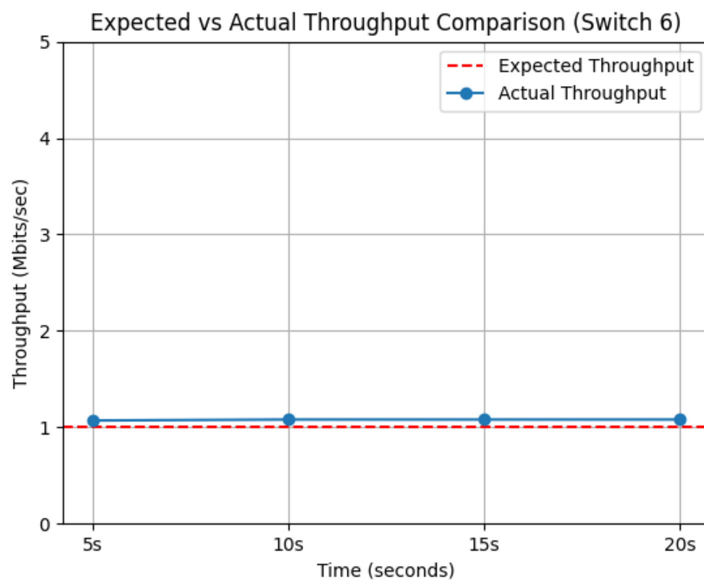

Figure 6.13: Throughput on switch 6

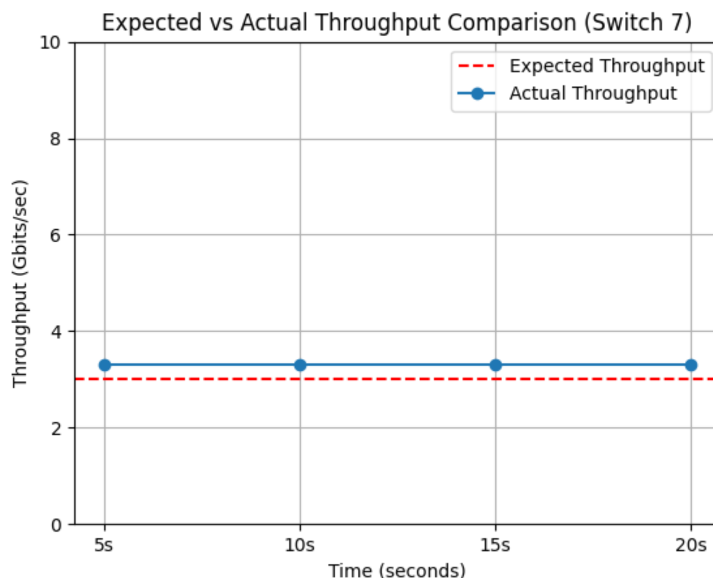

Figure 6.14: Throughput on switch 7

6.3.1 Load Profiling

In this paragraph, we analyze the load profiling functions by applying the following intents to the network:

- *"On switch 1 set the following weights : 3 on port 1, 3 on port 2, 3 on port 3 and 3 on port 4."*, shown in Fig. 6.15
- *"On switch 6 set the weights: 13 on port 1, 1 on port 1, 1 on port 2, 1 on port 3 and 1 on port 4."*, shown in Fig. 6.16
- *"On switch 7 the weights are: 6 on port 1, 6 on port 2, 2 on port 3 and 2 on port 4"*, shown in Fig. 6.17
- *"On switch 8 set the weights: 5 on port 1,6 on port 2, 4 on port 3,1 on port 4"*, shown in Fig. 6.18
- *"On switch 9 the weights must be: 5 on port 1, 4 on port 2, 2 on port 3 and 5 on port 4 "*, shown in Fig. 6.19

Unlike Ryu's evaluation using the iperf command, this experiment is conducted using the ping command. We decided to use *ping* instead of *iperf* because even a small amount of packets is considered sufficient for accurate load profiling. In this

experiment, we send 1,000 ICMP packets to hosts across different subnets. The results are listed in the following tables:

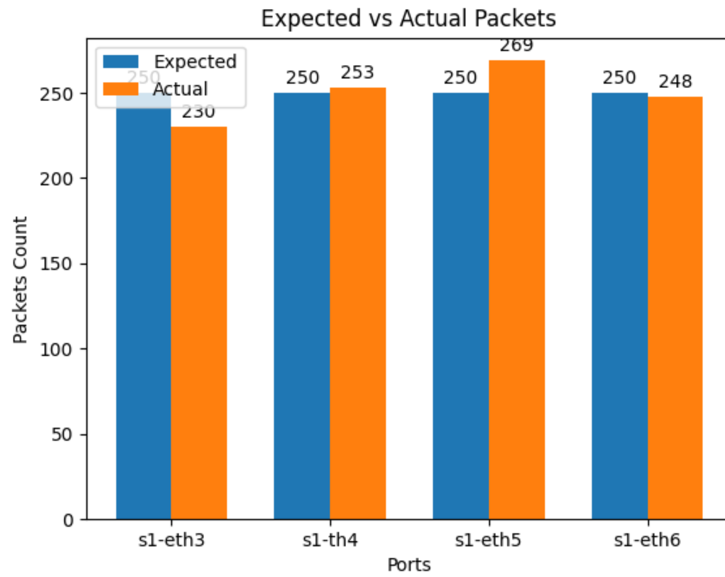

Figure 6.15: Packets counted and expected on switch 1

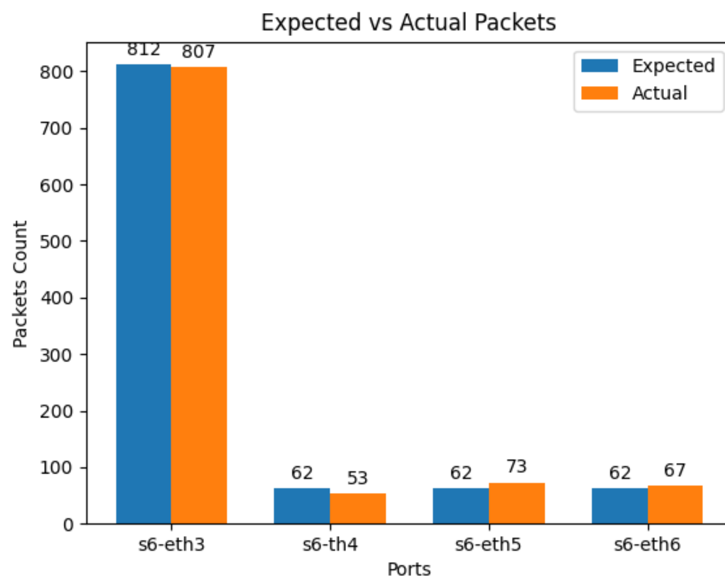

Figure 6.16: Packets counted and expected on switch 6

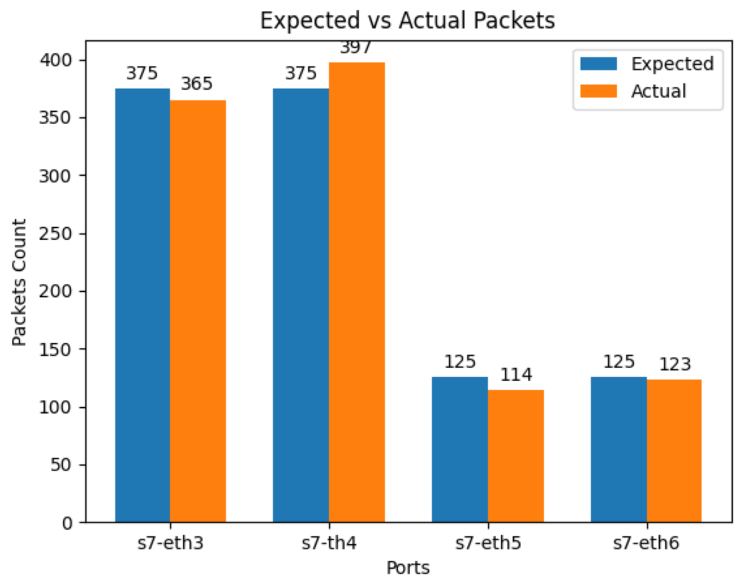

Figure 6.17: Packets counted and expected on switch 7

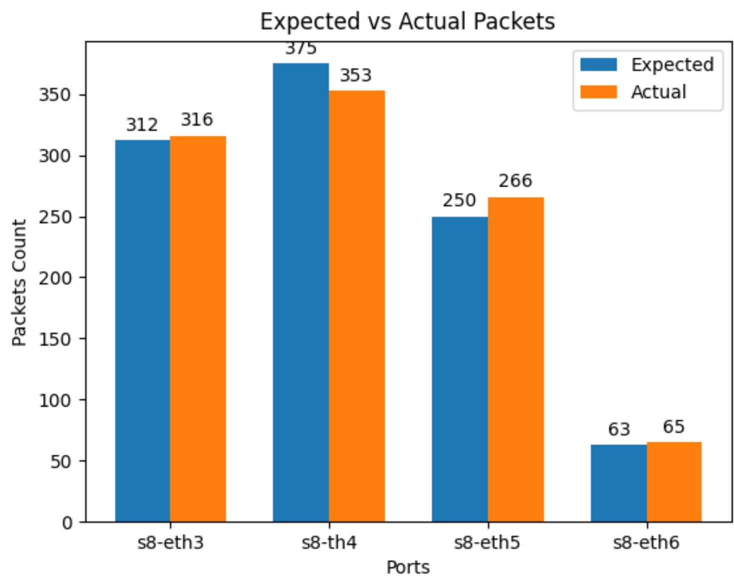

Figure 6.18: Packets counted and expected on switch 8

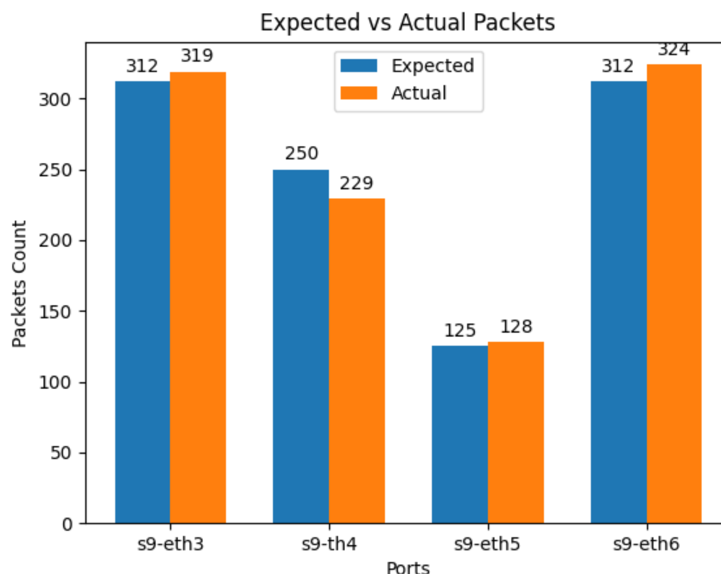

Figure 6.19: Packets counted and expected on switch 9

6.4 Large Language Model (LLM) Correctness

This section aims to analyze the correctness of the LLM to translate the human intent into a structured JSON format. This is challenging for different reasons. Firstly, the conventional paradigm of employing separate training, validation, and testing sets for performance evaluation becomes inapplicable in this context. Since LLMs are often trained on massive datasets with potentially opaque training procedures, we lack access to the specific distribution of data the model was exposed to during training. Also, as stated in other similar work [31], while established techniques exist for evaluating traditional machine translation tasks, assessing the correctness of translating natural language into formal specifications remains an open challenge. This difficulty arises from two primary factors: inherent ambiguity within natural language itself and the potential instability observed in Large Language Models (LLMs) [32]. In fact, one of the drawbacks of LLMs is their propensity for hallucination. Hallucination refers to the model's tendency to confidently fabricate information that may seem plausible but is entirely inconsistent. This can manifest in several ways during translation from natural language to formal specifications. For instance, the LLM might invent details not present in the original text, or it could generate nonsensical instructions that the system cannot execute. Hallucination impacts the accuracy of the resulting formal specification, potentially leading to unexpected or even malfunctioning systems. In our case, we

rely on a synthetic data set that includes several intent scenarios to evaluate the correctness of the LLM's translation.

6.4.1 Testbed

The hand-crafted dataset has been produced using a tool called Chatito [33]. Chatito is a tool specifically designed to facilitate the creation of high-quality datasets for training and validating Natural Language Processing (NLP) models, particularly those focused on conversational AI applications. Chatito allows developers to define examples simply and intuitively, streamlining the process of generating realistic and diverse conversations. Using Chatito, we could generate different human intents related to network functions. The correct translations of these intents to structured JSON were then manually annotated, creating a dataset of question-answer pairs. The dataset is composed of 70 blocking traffic intents, 70 rate limiter intents, and 70 load profiling intents for a total of 210 intents.

The experiment has been conducted on the platform of Langsmith, a tool offered by Langchain to evaluate and monitor the LLM based application. In particular, using the preceding generated dataset, we tested the converter to analyze the similarity between the generated output and the reference output in the dataset. The metrics used to evaluate the similarity between two JSON strings is the JSON edit distance [34]. This tool measures the similarity between two JSON strings by calculating their edit distance. Edit distance refers to the minimum number of edits (insertions, deletions, substitutions) needed to transform one string into another. The evaluator performs the following steps to ensure an accurate distance calculation:

- **Parsing**: It converts both JSON strings into corresponding data structures (like dictionaries or lists) for easier manipulation.

- **Normalization**: Whitespace and key order are disregarded to focus solely on the content and structure. Keys within dictionaries are sorted, and lists are arranged consistently. This ensures that semantically equivalent JSON strings with different formatting will have a zero distance.

- **Distance Calculation**: After normalization, a modified Damerau-Levenshtein distance algorithm is applied to determine the minimum edit operations required to make the two structures identical.

The distance formula is recursively defined as follows:

$$d_{a,b}(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1,j)+1 & \text{if } i > 0 \\ d_{a,b}(i,j-1)+1 & \text{if } j > 0 \\ d_{a,b}(i-1,j-1)+1 & \text{if } i,j > 0 \text{ and } a_i \neq b_j \\ d_{a,b}(i-2,j-2)+1 & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \end{cases}$$

- $d_{a,b}(i,j)$ denotes the measure of edit distance between the substrings $a[1..i]$ and $b[1..j]$.
- a and b refer to the strings being analyzed.
- a_i represents the ith character of string a.
- b_j represents the jth character of string b.
- The first scenario addresses the initial condition when both strings are devoid of characters.
- The second and third scenarios correspond to the cases of character deletion from a or b individually.
- The fourth scenario accounts for substitution when the characters at the respective positions differ.
- The fifth scenario covers transposition, which occurs when the last two characters of a and b are interchanged.
- $1_{(a_i \neq b_j)}$ serves as an indicator function, taking the value 0 when $a_i = b_j$ and 1 otherwise.

To achieve the most deterministic output possible, we set the LLM temperature to 0. We then analyze the correctness of the translation with and without prompting techniques to assess their efficacy.

6.4.2 Results

The results show the performance achieved in the experiments conducted. Table 6.4 shows the percentage of intent translated and the average edit distance score obtained when using prompting techniques or not for each network function. Table 6.5 shows the average usage of tokens per call for each network function and the related latency.

Network Function	Prompting Technique	% Intent Translated Correctly	Average Edit Distance Score
ACLs Firewall	Yes	100%	0.00
ACLs Firewall	No	71.4%	0.03
Load Profiling	Yes	100%	0.00
Load Profiling	No	71.4%	0.05
Rate Limiter	Yes	100%	0.00
Rate Limiter	No	84.2%	0.01

Table 6.4: Performance with and without Prompting Techniques

Network Function	Prompting Technique Used	Latency (s)	Average Tokens (Input)	Average Tokens (Output)
Firewall ACLs	Yes	2.25	630	53
Firewall ACLs	No	2.04	302	53
Load Profiling	Yes	2.15	596	59
Load Profiling	No	2.05	322	59
Rate Limiter	Yes	1.83	520	32
Rate Limiter	No	1.66	299	32

Table 6.5: Average Token Usage and Latency for Each Network Function

6.5 Conclusion

The results demonstrate the system's ability to accurately translate human intent into structured JSON format, which is then correctly mapped to network application APIs. Notably, the prompting technique significantly improves the translation accuracy, increasing the average percentage of correctly translated intents from around 75% to 100%. Without the prompting technique, the model sometimes makes different mistakes, for example, misrepresenting the values (weights, IP addresses, or rates) associated with the intent or wrongly defying the key value of the JSON object. Additionally, using the prompting technique, the average number of tokens used per call to the LLM is remarkably low, at around 580 tokens in input and 48 in output, with an average latency of 2 seconds to receive a generated output. Running this model on Vertex AI [35], which is Google Cloud's unified platform for building, deploying, and managing machine learning (ML) model, the prices [36] related to the usage of PaLM with the model text-bison are the following:

Item	**Description**	**Price (USD)**
Input	*Price per 1,000 characters*	*$0.00025*
Output	*Price per 1,000 characters*	*$0.0005*

Table 6.6: Input and Output Prices of the PALM Model

As visible from Table 6.6, each call in input costs around $ 0.00043 and in output $ 0.000024. This price is taken considering an estimation of 1 token for each 3 characters.

Furthermore, while Ryu applications generally adhered to user specifications, load profiling revealed deviations in traffic distribution from the intended weights. Conversely, P4-eBPF applications achieved near-perfect adherence to user intent, delivering satisfactory results.

Chapter 7

Conclusion

In this section, we describe the final considerations, the limitations, and the related future works.

7.1 Final Considerations

The proposed pipeline, which integrates LLMs' capabilities with network APIs and tools such as Langchain, Ryu-SDN, and P4-eBPF, effectively bridges the gap between human intent and network configuration implementation. Through rigorous evaluation across various use cases including firewall, rate-limiting, and load profiling, the model has demonstrated its ability to accurately translate intent into machine-readable formats and execute them within the network infrastructure. This work contributes to simplifying network operations, reducing manual coding efforts, and advancing the field of Intent-Driven Networking, thereby opening up new possibilities for efficient network management in complex environments.

7.2 Limitations

During the thesis, we encountered different limitations:

- Finding a free-to-use large language model (LLM) service proved challenging. The only readily available option was Google's PaLM, which has a 1024-token limit per call in its free tier. This restriction necessitated dividing prompts for each network function and directing user input to the appropriate prompt template.
- As a relatively new tool, P4-eBPF currently lacks some features present in more established options like P4runtime. This limited our ability to develop a controller in a traditional manner.

- Due to the absence of a standardized approach for evaluating LLM-based applications, we were forced to rely on a custom-built dataset for evaluating specific scenario. The examples of user requests are limited to a set of situations, and a more in-depth study should be conducted to explore the system's limitations.

7.3 Future works

Several improvements can be made to enhance the system's capabilities:

- The system could be extended to encompass a wider range of network functions. This would allow users to manage their network infrastructure more comprehensively by providing access to a broader set of services.
- Implementing an additional tier that facilitates the expression of complex user intent is a potential advancement. This tier could leverage a combination of various network functions, enabling users to define intricate network management scenarios through high-level policies. It might involve adding a task classification layer that identifies the necessary network functions to fulfill user-defined policies.
- The incorporation of a checker within the system to identify and prevent conflicting user intents would be beneficial. This would ensure the system's consistency and avoid unintended network configurations.
- Adding a monitoring feature to the system would enable it to periodically verify if the user-defined intents are being successfully implemented. This would provide valuable feedback for users and help maintain optimal network performance.

Bibliography

[1] Tom B. Brown et al. *Language Models are Few-Shot Learners.* 2020. arXiv: 2005.14165 [cs.CL] (cit. on pp. 3, 4).

[2] OpenAI et al. *GPT-4 Technical Report.* 2023. arXiv: 2303.08774 [cs.CL] (cit. on p. 3).

[3] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways.* 2022. arXiv: 2204.02311 [cs.CL] (cit. on pp. 3, 29).

[4] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models.* 2023. arXiv: 2312.11805 [cs.CL] (cit. on pp. 3, 30).

[5] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models.* 2023. arXiv: 2302.13971 [cs.CL] (cit. on p. 3).

[6] Qingxiu Dong et al. *A Survey on In-context Learning.* 2023. arXiv: 2301.00234 [cs.CL] (cit. on p. 4).

[7] *Advanced Prompt Engineering.* https://towardsdatascience.com/advanced-prompt-engineering-f07f9e55fe01 (cit. on p. 5).

[8] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* 2023. arXiv: 2201.11903 [cs.CL] (cit. on pp. 4–6).

[9] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. *Large Language Models are Zero-Shot Reasoners.* 2023. arXiv: 2205.11916 [cs.CL] (cit. on p. 7).

[10] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. *Automatic Chain of Thought Prompting in Large Language Models.* 2022. arXiv: 2210.03493 [cs.CL] (cit. on p. 7).

[11] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. *Self-Consistency Improves Chain of Thought Reasoning in Language Models.* 2023. arXiv: 2203.11171 [cs.CL] (cit. on p. 7).

[12] Yuxiang Wu, Guanting Dong, and Weiran Xu. *Semantic Parsing by Large Language Models for Intricate Updating Strategies of Zero-Shot Dialogue State Tracking*. 2023. arXiv: 2310.10520 [cs.CL] (cit. on p. 7).

[13] Jan Deriu, Alvaro Rodrigo, Arantxa Otegi, Guillermo Echegoyen, Sophie Rosset, Eneko Agirre, and Mark Cieliebak. «Survey on evaluation methods for dialogue systems». In: *Artificial Intelligence Review* 54.1 (June 2020), pp. 755–810. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09866-x. URL: http://dx.doi.org/10.1007/s10462-020-09866-x (cit. on p. 7).

[14] Namo Bang, Jeehyun Lee, and Myoung-Wan Koo. *Task-Optimized Adapters for an End-to-End Task-Oriented Dialogue System*. 2023. arXiv: 2305.02468 [cs.CL] (cit. on p. 7).

[15] Arthur Jacobs, Ricardo Pfitscher, Rafael Ribeiro, Ronaldo Ferreira, Lisandro Granville, and Sanjay Rao. «Deploying Natural Language Intents with Lumi». In: Aug. 2019, pp. 82–84. ISBN: 978-1-4503-6886-5. DOI: 10.1145/3342280.3342315 (cit. on p. 8).

[16] Antonino Angi, Alessio Sacco, Flavio Esposito, G. Marchetto, and Alexander Clemm. «NLP4: An Architecture for Intent-Driven Data Plane Programmability». In: June 2022, pp. 25–30. DOI: 10.1109/NetSoft54395.2022.9844035 (cit. on pp. 8, 9).

[17] Antonino Angi, Alessio Sacco, Flavio Esposito, G. Marchetto, and Alexander Clemm. «NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches». In: *IEEE Communications Magazine* PP (Jan. 2023), pp. 1–7. DOI: 10.1109/MCOM.001.2300313 (cit. on pp. 8–10).

[18] Fulvio Risso. *Architecture of network devices.* https://swnet.frisso.net/ (cit. on pp. 13, 15).

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. «OpenFlow: enabling innovation in campus networks». In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746 (cit. on p. 15).

[20] *Ryu-SDN framework.* https://ryu-sdn.org/ (cit. on p. 16).

[21] Pat Bosshart et al. *Programming Protocol-Independent Packet Processors*. 2014. arXiv: 1312.1719 [cs.NI] (cit. on pp. 16, 18).

[22] *P4v16 Portable Switch Architecture.* https://p4.org/p4-spec/docs/PSA.pdf (cit. on p. 19).

[23] *What is eBPF?* https://ebpf.io/what-is-ebpf (cit. on p. 19).

[24] *PSA implementation for eBPF backend.* https://github.com/p4lang/p4c/tree/main/backends/ebpf/psa (cit. on p. 21).

[25] *eBPF Backend.* https://github.com/p4lang/p4c/tree/main/backends/ebpf (cit. on p. 21).

[26] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. «A novel programmable software datapath for software-defined networking». In: *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies.* CoNEXT '22. New York, NY, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3555050.3569117. URL: https://doi.org/10.1145/3555050.3569117 (cit. on pp. 21–24).

[27] *PSA-eBPF: Portable Switch Architecture for eBPF.* https://opennetworking.org/wp-content/uploads/2022/05/PSA-eBPF-Tech-Brief-Final-Slide-Deck.pdf (cit. on p. 24).

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need.* 2023. arXiv: 1706.03762 [cs.CL] (cit. on pp. 26, 27).

[29] *Langchain's modules.* https://python.langchain.com/docs/modules/ (cit. on p. 31).

[30] *Multipath Routing with Load Balancing using RYU OpenFlow Controller.* https://github.com/wildan2711/multipath/blob/master/ryu_multipath.py (cit. on p. 49).

[31] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. *Making Network Configuration Human Friendly.* 2023. arXiv: 2309.06342 [cs.NI] (cit. on p. 71).

[32] Ziwei Ji et al. «Survey of Hallucination in Natural Language Generation». In: *ACM Computing Surveys* 55.12 (Mar. 2023), pp. 1–38. ISSN: 1557-7341. DOI: 10.1145/3571730. URL: http://dx.doi.org/10.1145/3571730 (cit. on p. 71).

[33] *Chatito.* https://github.com/rodrigopivi/Chatito/tree/master (cit. on p. 72).

[34] Leonid Boytsov. «Indexing methods for approximate dictionary searching: Comparative analysis». In: *ACM J. Exp. Algorithmics* 16 (May 2011). ISSN: 1084-6654. DOI: 10.1145/1963190.1963191. URL: https://doi.org/10.1145/1963190.1963191 (cit. on p. 72).

[35] *Vertex AI.* https://cloud.google.com/vertex-ai?hl=it (cit. on p. 74).

[36] *Vertex AI pricing.* https://cloud.google.com/vertex-ai/generative-ai/pricing (cit. on p. 74).