POLITECNICO DI TORINO
Repository ISTITUZIONALE

LLNet: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a Small Language Model

Original
LLNet: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a Small Language Model / Angi, A., Sacco, A., Marchetto, G.. - In: IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. - ISSN 1932-4537. - ELETTRONICO. - 22:4(2025), pp. 3403-3418. [10.1109/tnsm.2025.3570017]

Availability:
This version is available at: 11583/3001616 since: 2025-07-07T12:24:57Z

Publisher:
IEEE

Published
DOI:10.1109/tnsm.2025.3570017

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

18 October 2026

LLNet: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a *Small* Language Model

Antonino Angi, *Student Member, IEEE,* Alessio Sacco, *Member, IEEE,* and Guido Marchetto, *Senior Member, IEEE*

***Abstract*—Traditional network management requires manual coding and expertise, making it challenging for non-specialists and experts to handle increasing devices and applications. In response, Intent-Based Networking (IBN) has been proposed to simplify network operations by allowing users to express in natural language the program objective (or intent), which is then translated into device-specific configurations. The emergence of Large Language Models (LLMs) has boosted the capabilities to interpret human intents, with recent IBN solutions embracing LLMs for a more accurate translation. However, while these solutions excel at intent comprehension, they lack a complete pipeline that can receive user intents and deploy network programs across devices programmed in multiple languages. In this paper, we present LLNet, our IBN solution that, within the context of Software-Defined Networking (SDN), can translate seamlessly intent-to-program. First, leveraging LLMs, we convert network intents into an intermediate representation by extracting key information; then, using this output, the system can tailor the network code for any topology using the specific language calls. At the same time, we address the challenge of a more sustainable IBN approach to reduce its energy consumption, and we experience how even a *Small* Language Model (SLM) can efficiently help LLNet for input translation. Results across multiple use cases demonstrated how our solution can guarantee adequate translation accuracy while reducing operator expenses compared to other LLM-based approaches.**

***Index Terms*—user intents, LLM, SLM, network programmability, intent-based networking, SDN.**

I. Introduction

The rapid evolution of network technologies and the exponential growth in data traffic have made traditional network management techniques both inadequate and prone to human errors, increasing the demand for smarter and more intuitive management solutions. While the advent of Software-Defined Networking (SDN) has simplified the programming of network devices and enabled administrators to customize networks to meet specific requirements, writing code for these devices remains challenging due to the limitations imposed by hardware-level programming languages [1]–[5]. To overcome the limited abstraction offered by these languages, recent solutions have embraced the Intent-Based Networking (IBN) paradigm, a revolutionary approach powered by Natural Language Processing (NLP) to simplify network operations and make them accessible to a broader range of users, letting operators write intents (*i.e.,* declarative set of goals and outcomes that a network should meet) rather than working code [6]. As writing network configurations requires a high level of technical expertise, manual coding, and execution, IBN disrupts this paradigm by enabling declarative interfaces to democratize network management and enhance operational efficiency by allowing both network experts and non-technical stakeholders to collaborate seamlessly.

The emergence of powerful Large Language Models (LLMs) such as Meta's Llama [7], OpenAI's GPT [8] and PaLM [9], marks a significant leap in NLP capabilities [10]–[12]. These models have demonstrated remarkable proficiency in understanding and generating human language, making them ideal candidates for implementing IBN [13]–[16]. Current solutions advance the expressiveness of user intents, allowing operators to express the same intent using different phrasings. However, new challenges appear. For example, pre-trained open-source LLMs, like Code Llama [7], lack training on network-specific data, making them challenging to employ for network management purposes without further training [17], [18]. Moreover, using LLM to catch user needs is just the preliminary step for a more complete network program deployment that can help operators program the network without incurring in the typical "trial and error" process.

In this paper we propose LLNet which integrates LLM-powered NLP with IBN to create a more intuitive and accessible network management framework that can cover the full deployment cycle. By harnessing the capabilities of LLMs, our solution can understand and implement network policies expressed in natural language, thereby making them more inclusive and efficient. LLM extracts data and keywords from human intent, and LLNet maps this output to the deploying network infrastructure, enabling seamless translation from policy to configuration. This approach not only streamlines the configuration process but also reduces the likelihood of human error, leading to more reliable and efficient network management.

With our proposed system design, we simultaneously address two issues related to LLMs. First, we make the language model generalize the transpiled code towards diverse languages both at the data-plane and control-plane by utilizing an appropriate way of interacting with the LLM (referred to in the literature as prompting). This removes the need for fine-tuning (*i.e.,* deploy and re-train an LLM) and makes LLNet independent of the chosen LLM. Second, our solution

Antonino Angi, Alessio Sacco and Guido Marchetto are with DAUIN, Politecnico di Torino, 10129 Turin, Italy (e-mail: antonino.angi@polito.it, alessio_sacco@polito.it, guido.marchetto@polito.it).

works even with a *Small* Language Model (SLM) for intent interpretation, thus making IBN more sustainable and reducing the energy consumption and the economic expenses of using this type of model that is otherwise resource-hungry [19]–[21].

We prototyped LLNet over common data-plane technologies such as P4, OpenFlow, and eBPF, and control-plane frameworks such as Ryu, P4Runtime, and FloodLight. Results validate the accuracy of our small model when converting user intents over various network programs, showing it stably over 88%. At the same time, we experienced that the adoption of SLMs in LLNet can reduce costs by an average of 89.54% compared to other LLM options, while also reducing its carbon footprint.

In summary, the main contributions of this paper are:

- An SLM-integrated IBN framework (LLNet) that enables natural language-based policy expression and execution, facilitating network management throughout the deployment process;
- A language-agnostic translation mechanism leveraging prompt-based interaction with LLMs to generate configuration code across diverse network technologies, without requiring fine-tuning;
- Support for lightweight interpreters, showing that SLMs can effectively interpret network intents while significantly reducing energy consumption and computational costs;
- A full-stack prototype implementation, validated on multiple control-plane and data-plane technologies, demonstrating high accuracy (over 88%) and substantial cost savings (up to 89.54%) when adopting an SLM over LLM;
- Built-in feedback and monitoring support to evaluate and iteratively improve the effectiveness of deployed policies, enhancing the usability and reliability of intent-driven networking.

The rest of the paper is structured as follows. Section II presents the state-of-the-art methodologies that also focus on LLM-based solutions for IBN scenarios. Section III describes our architecture, giving a general overview of its design. In Section IV, we focus on the adopted SLM, providing details on the leveraged prompting techniques. Section V describes the network infrastructure deployed within LLNet. In Section VI, we illustrate a step-by-step example of our solution, starting from the network topology and intent provided by the user and concluding with the generated network configurations. Finally, results are shown in Section VII, and the conclusion in Section VIII.

II. Related Work

Network management is known to be a tedious task requiring manual coding and execution, thus posing a significant barrier for non-technical users. Recent advancements in Natural Language Processing (NLP) have favored the proliferation of solutions aiming to simplify the management of network infrastructure by making network management more human-friendly, reducing errors, and improving efficiency. A key innovation in this area is the use of network intents, introducing IBN, which allows administrators to specify high-level policies and goals that the network should achieve without specifying how to achieve those goals [22]. Some of these approaches rely on traditional NLP techniques, such as NAIL [18] and Lumi [23]. While the first uses an NLP-based transpiler to translate human intents into low-level data plane configurations (*i.e.,* P4 language) using a network management API to manipulate the involved network elements; the latter implements a chatbot interface with an intent definition language to match the input into an operation and identify the involved network elements and functions to apply (*e.g.,* bandwidth manipulation, middlebox). Another example of the intersection of NLP techniques with chatbot-like interfaces is [24], where the authors propose an IBN-based framework for expressing network slice intents, improving users' Quality of Experience and network automation. With the same goal, the authors in [25] present Chat-IBN-RASA, a RASA-based chatbot trained using Natural Language Understanding (NLU) models to deliver a conversation-like experience to the final user. By leveraging user feedback to constantly improve its translation accuracy, Chat-IBN-RASA is able to efficiently manage packet-optical networks through intents expressed in natural language.

The advent of Generative AI with LLMs (*e.g.,* Meta's Llama [7], OpenAI's GPT [8]) represents a significant milestone in 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 NLP. LLMs demonstrate exceptional versatility compared to their predecessors, excelling in a diverse domain of NLP tasks, generating logical, well-structured content, and facilitating in-depth conversations with users, thereby unlocking the vast potential for numerous scenarios (*e.g.,* personalized content recommendation [26], medical assistance [27]).

In recent years, researchers have also embraced the progress made on LLMs with network intents to enhance network automation by introducing deployment aspects such as the "zero-touch networks" paradigm. This integration leverages the advanced natural language understanding capabilities of LLMs to interpret and implement high-level network policies, significantly reducing the need for manual configuration and intervention. By doing so, networks become more adaptive, resilient, and easier to manage.

An example of this integration is shown in [13], a three-stage pipeline system in conjunction with the few-shot learning capabilities of LLMs that translates human intents into low-level executable policies. The first stage abstracts the intent by mapping it into specific types recognized by the system model (*e.g.,* utility, goal); the second stage converts the intent into policies and, finally, the third stage controls the policies before being mapped into application management APIs.

This integration is further shown in two other valuable works: GeNet [14] and LLM-NetCFG [28]. The first one, GeNet [14], develops a framework that leverages OpenAI's GPT-4 for two main tasks: (i) converting network topology images into textual descriptions with the LLM's vision capabilities and (ii) translating user intents into network device

configurations. The latter, LLM-NetCFG [28], introduces a network configuration generator by adopting a locally deployed LLM (*i.e.,* Zephyr-7b), fine-tuned on a specialized database network tasks, to provide faster response times when queried about identifying the network components and the objective of the given intent.

Another attempt to fully exploit the potential of LLMs to configure the network through intents is brought by [29], a framework that translates network intents to BGP routing configurations with different LLMs (*e.g.,* Llama2, Mistral, GPT-3.5). In [29], the network intent goes through a middleware pipeline that pre-processes it by classifying the intent's main task and extracting the associated network components. The recognized network elements will then be forwarded to the chosen LLM, which returns the possible network configurations, which will be finally validated by a specific module to check their correctness with the given network topology.

A more recent study [30] introduces an LLM-based pipeline designed to automate 5G network configurations and support network operators' deployment process. After pre-processing and tokenization, these tokens are analyzed by a Named-entity Recognition (NER) function and a fine-tuned LLM (*i.e.,* GPT-3.5, Llama2-7B) to extract relevant keywords, where a deep neural network filter is also employed to prevent input errors. Finally, a specialized API generates the required network configuration that will be human-validated and tested on a replica before being pushed into production.

With the advent of Software-Defined Networking (SDN) aiming to simplify the programming of network devices, studies have focused on enhancing SDN management and optimization by integrating LLMs to customize the network on a lower-level scenario (*e.g.,* at the data plane) and generate configurations to be directly installed into network devices. An example of this integration is brought by NETBUDDY [15], which translates high-level specified policies into low-level network programming configurations (*i.e.,* in P4 and BGP) with the support of LLMs. In this work, the model input (*i.e.,* the human intent) first goes through an LLM to generate the formal specification used for the embedding (*i.e.,* the data structure corresponding to the involved network element). It is then helped by an intermediate structure to translate the formal specification into a high-level structure used as routing information (*i.e.,* the connection between the network elements). Finally, NETBUDDY generates the low-level configurations installed into the destination device (*e.g.,* switch, router). The same authors also developed a benchmark called NetConfEval [32], which helps evaluate the effectiveness of different LLMs in automating network configuration.

Similarly, HLDPL [31] has been proposed as an innovative approach to simplifying network application development. It uses LLMs to translate natural language prompts into low-level programming code (*i.e.,* P4 and its extensions), which can then be installed and executed on physical devices, thereby overcoming the challenges of writing in low-level programming languages.

Our Contribution: Intersecting these contributions and leveraging such recent advances, in this article, we present LLNet that leverages an SLM to interpret user intents and enables an easy way to program network devices. As pointed out in Table I, although our work draws inspiration from the mentioned applications, we specifically focus on translating human intent into network configurations utilizing the few-shot capabilities of LLMs to transform human intent into a machine-readable format. These configurations are then applied to the network infrastructure using the appropriate functions. We also allow users to monitor the key metrics to validate the effectiveness of the deployed intent and, in case it fails to meet expectations, update it. Finally, we demonstrate how more lightweight interpreters are still acceptable in this regard and can drastically reduce energy consumption, a major issue of modern LLMs.

III. Overall Architecture Design

In this section, we describe the main components of LLNet, focusing on how the offered functionalities are achieved. An overview of LLNet is shown in Fig. 1. Four main components form our architecture:

- *User Interface*: This block is the entry point for the entire solution. It provides two main types of feedback: *(i)* a chatbot, similar to conventional SLM or LLM systems, allows the user to input their intent and interact multiple times to confirm the accuracy of the interpretation. This is essential for validating the correctness of the interpretation; *(ii)* real-time monitoring capability, which enables the user to confirm whether the network is indeed executing the intended program. This feature is crucial, as many existing solutions are limited to the interpretation phase, potentially leading to discrepancies later in the process. We argue that continuous monitoring of key network metrics is valuable to the user.
- *Intent Converter*: This block receives, translates and parses the human intent into a machine-readable JSON object later processed by the Network Manager. The translation and parsing are done with the help of LLMs. However, instead of directly inputting the user's text to the SLM, we use an intermediate structure that performs a pre-matching mechanism based on the demanded program. Our insight is that, rather than directly asking the SLM to generate the code (which has been observed to possibly generate diverging or buggy code [33]), it generates a configuration file that is then used by network applications. We share this procedure with other IBN approaches, *e.g.,* [15], [18], [23], [34], which observed how ChatGPT and other LLMs may lead to diverging or buggy code [33]. On the contrary, by working on use cases and placeholders, we insert rules (derived from the JSON object) into the working code, and it is easier for the user to focus only on the desired functionalities. Moreover, along with the user text, we add working examples that serve as a guide for the SLM. This action is done by the embedding and dispatcher components (detailed in Sec. IV-B), works by cosine-similarity, and better associates the intent to application scenarios known to LLNet. Specifically, cosine similarity measures the cosine of the angle between two vectors representing strings in a

	Advanced Prompting	Human Validation	Monitoring	Adaptability to more network planes	Sustainable Analysis & SLM Adoption
[13]	✘	✔	✔	✘	✘
GeNet [14]	✘	✔	✘	✘	✘
NETBUDDY [15]	✔	✔	✘	✔	✘
LLM-NetCFG [28]	✔	✔	✘	✘	✘
[29]	✔	✔	✘	✘	✘
HLDPL [31]	✔	✘	✘	✘	✘
LLNet	✔	✔	✔	✔	✔

TABLE I: Comparative analysis against the most recent works on LLMs for IBN. LLNet can jointly interpret user intents to program network devices with several languages and at several layers while accounting for a sustainable solution.

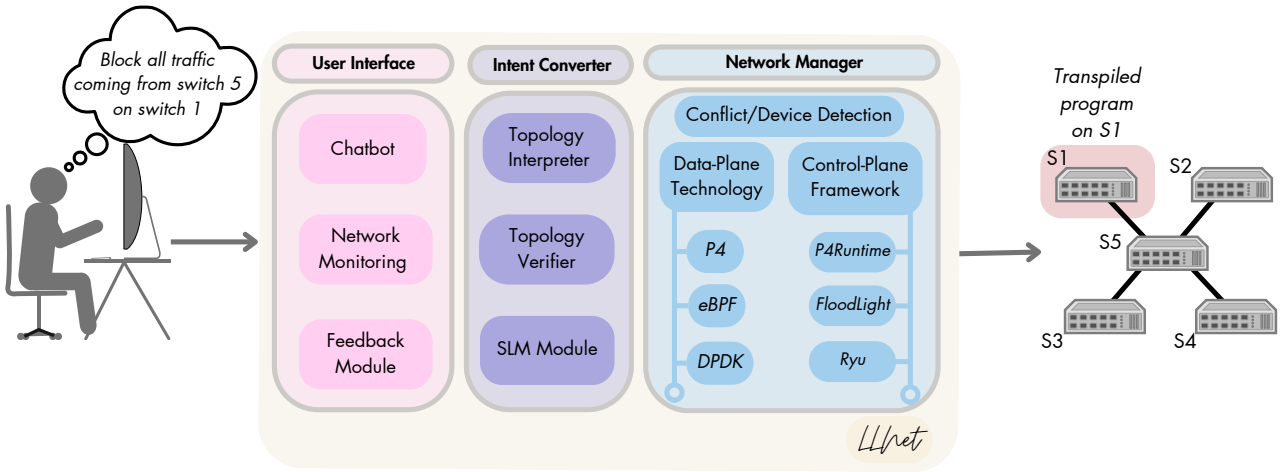


Fig. 1: Overview of the System's Architecture. LLNet provides a user-friendly interface enabling intent-to-network translation and immediate feedback to the user.

high-dimensional space, matching the overall scope of the introduced intent [35]. Some examples considered in our prototype are: *Firewall ACLs*, which allows specifying IP ranges/addresses to be blocked on the specified switch; *Rate limiter*, which allows limiting the bandwidth of all incoming traffic on the specified switch; *Load profiling*, that allows selecting weights on switch ports to distribute traffic across the network infrastructure based on user-specified weights. However, these applications only represent use cases for our tested scenarios, but LLNet possesses the versatility to be effectively implemented in a variety of other contexts, as long as a template is provided in the prompt.

- *Network Manager*: This block targets the appropriate data-plane or control-plane language 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. It has to deal with the implementation of a network application that the user interacts with. Our solution can work with different data-plane technologies, *i.e.,* OpenFlow, P4, eBPF, DPDK, and control-plane frameworks, *i.e.,* Ryu, P4Runtime, FloodLight. This module not only transpiles into the correct working code. LLNet can locate the node that best serves the intent, and, once located, it installs the specific rules. Consider for example, a network intent referring to a functionality that should be distributed among different devices, such as security rules in our firewall use case. Alternatively, in load profiling or just forwarding, LLNet is also equipped with capabilities of locating the interested devices and specializing the commands to them. To achieve this functionality, the topology in input is used to generate rules tailored to the specific infrastructure.
- *Network Infrastructure*: This layer consists of all data-plane switches and (if needed) control-plane applications. Human intent is applied by setting rules on switches using the network tools mentioned before.

While user intents can refer to any aspect of the network, *e.g.,* users, traffic, flow, LLNet is explicitly designed to translate such network intents into rules installed on individual nodes or devices. This design choice is justified by two main reasons: *(1)* it reflects the nature of the presented use cases (*e.g.,* firewall ACLs, load profiling), which require node-specific actions; *(2)* it enhances the LLNet's reliability and accuracy as it has to localize individual nodes rather than complex, distributed configurations. For example, firewall rules are applied to specific switches to block traffic, while load profiling involves configuring weights on specific switch ports to optimize traffic distribution. Locating the devices and installing the proper rules is part of LLNet operations. While the correctness of the syntax is verified by the Intent Converter, we also design continuous feedback for the user. This phase is crucial for ensuring the accuracy w.r.t. the actual goal of the network program. Users can confirm whether the intent is accurately interpreted and if the network is functioning as

intended. If not, adjustments can be made to align the program with expectations. Another important functionality offered by LLNet is the ability to satisfy the intent over time. While the intent is fixed, the generated rules can vary in time, for example, as network topology changes – *e.g.,* a switch gets added, routes change, new rules must be deployed, or when a specific link fails. Our solution can adapt the network program to environmental changes.

IV. Converting Human Intents with an SLM

In this section we focus on the core ability of LLNet, the interpretation and conversion of human intents by means of the SLM.

The *Intent converter*'s role is to translate human intents into a machine-readable object using an SLM. This class of models is a subset of deep learning, a field of ML 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, these language models intersect another branch of Artificial Intelligence: Generative AI.

A. Advantages of Language Models

Recent LLMs are built on the transformer architecture, a neural network framework introduced in 2017 [36]. Transformer revolutionized NLP tasks by capturing long-range dependencies without relying on recurrent networks, using self-attention mechanisms to weigh contextual relevance across input sequences. This capability enables LLMs to construct nuanced word representations that reflect full-sentence context, which is crucial for IBN. This ability of LLMs is key for IBN, enabling understanding of the complete structure of the sentence rather than each word individually, as in other NLP techniques. Before processing the input text, the text undergoes tokenization, bridging the gap between natural language and model input. This preprocessing phase typically includes: *(i)* text cleaning (removal of noise like punctuation or symbols), *(ii)* normalization (*e.g.,* lowercasing, lemmatization), and *(iii)* tokenization, often using sub-word units such as prefixes or suffixes for better generalization.

LLMs are first pre-trained on large text data to develop general language understanding. For task-specific adaptation, two main strategies are used: *Fine-Tuning*, involving additional training on task-relevant data, and *Prompt Engineering*, which modifies the input to elicit desired outputs without retraining. LLNet uses prompt engineering to efficiently tailor LLM behavior for diverse network scenarios (see Sec. VII-E), achieving high accuracy in intent translation while avoiding the overhead of fine-tuning.

B. Integrating Language Models in LLNet

The SLM in LLNet running in the *Intent Converter* block, processes the human intent by converting it into a machine-readable format through semantic parsing to effectively match the specified network functions and elements. This process begins with the SLM, which parses the intent and converts it into a JSON object with a well-defined structure that helps encapsulate high-level user policies. This file, which will then be passed to the network application, is comprised of an object specifying the user-defined goal, the switch ID where the intent is to be applied, and the parameters necessary for implementation. If the switch ID cannot be determined because it is not present in the user text, the *Network Manager* is able to set it by considering the network topology and IP addresses in it.

Each intent written by the user is thus inserted in the right prompt that will be given in input to the SLM. This is achieved by routing the input to the correct prompt based on semantic similarity. Our Intent Converter achieves this through two key modules: the *Semantic Embedding* and the *Dispatcher*, which direct the user input to the most suitable prompt template based on semantic similarity. First, all the pre-defined prompts and the user's intent are embedded into a common vector space, where this space is the mathematical representation of the meaning. Then, the dispatcher calculates the semantic similarity (*e.g.,* via cosine similarity) between the user's intent and each available prompt and routes the input to the most semantically similar prompt for specialized processing by the SLM. The prompt templates provide a structure for feeding the SLM with the user's input, acting as guides, ensuring the input is presented so the SLM can understand and generate a meaningful response.

In particular, this prompt is generated via two key prompting techniques: *Few-Shots Prompting* and *Chain of Thoughts*. The former 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 [37]. The latter method, presented in [38], provides explicit instructions or steps to guide the LLM through the desired task or sequence of actions. Prompting techniques play a crucial role in shaping the input processed by SLMs or LLMs, directly impacting their performance and output quality [39]. By clarifying the context, effective prompting helps those models generate more accurate responses, making them suitable across different scenarios.

We deployed LLNet by shaping a dedicated prompt for each analyzed network function. This strategy helps generate the appropriate responses for our Network Manager, which will then generate the network configurations. They are designed to leverage few-shot learning capabilities by incorporating several key elements. First, each prompt template includes examples that demonstrate how a user's intent can be translated into a JSON object. Additionally, for more complex prompts, step-by-step instructions are provided to break down the user's intent into smaller tokens, helping the SLM better understand and accurately translate the overall goal. Finally, format instructions are included to specify the required structure and data types for the input, ensuring consistency and clarity within the prompt. An example of a prompt template is shown in Listing 1.

By using well-designed prompting templates and clear rules, the number of output tokens generated by the SLM can be

minimized, helping reduce the overall token usage and decreasing associated costs [40]. For this reason, we shaped our adopted SLM to respond in a structured format (*i.e.,* in JSON), avoiding unnecessary comments, leading to a more efficient output generation, and reducing overall costs. We summarize in Fig. 2 all steps occurring in LLNet from the reception of user intents to the actual network program deployment. The flow-chart starts with a user defining a topology and introducing an intent. These entries are then analyzed by the adopted SLM, which converts them into JSON representations of network elements (*e.g.,* switch, IP address) and checks whether any intent's network element matches with one in the topology. If no element matches, an explanatory error is provided to the user, and the program ends. If at least one element matches, the SLM analyzes the intent to understand if its goal matches one of the application scenarios known to LLNet (*e.g.,* load profiling, rate limiter). If it does not, an explanatory error is provided to the user and the program ends; otherwise, the user is asked to confirm the JSON representation of the intent where both the involved network elements and the main goal are extrapolated. If not confirmed, LLNet starts over and the user is asked to re-introduce both the topology and the intent. If confirmed, LLNet locates the involved network devices and checks whether the targeted network elements have already been targeted by a previously added intent. If so, the user is prompted to confirm whether to proceed with the installation, ensuring that redundant or conflicting intents are not unintentionally deployed. After user confirmation or if no elements have been previously been interested by an intent, LLNet generates the necessary rules and deploys them via the appropriate API; otherwise, no action is taken. If the user does not confirm, LLNet terminates.

Listing 1: An example of a prompt template with specified output rules.

```
You are a good and performant system that parses
    human intent to JSON object following these
    rules:
[Rules]
1. Identify the involved network elements (switch id
    , ip source and ip destination address), on
    which the intent will be applied.
2. If the source IP is not defined, set it to "any".
3. If the destination IP is not defined, set it to
    "any".
4. The network configuration should be returned in
    json format based on the following schema
    definition without additional comments.
5. Follow the provided output_schema.
6. As reference, consider the provided examples.

[OUTPUT_SCHEMA]
\n{format_instructions}\n

[EXAMPLES]:
\n{examples}\n

The intent is the following:
{query}
```

As in other related works [23], [25], when LLNet receives a valid but unrecognized configuration parameter, the system tries to generate a JSON representation and asks the user for confirmation, rather than directly attempting an incorrect or incomplete rule installation. This minimizes errors and ensures the user is aware of potential issues with their input, avoiding incorrect network configurations or unintended network behaviors.

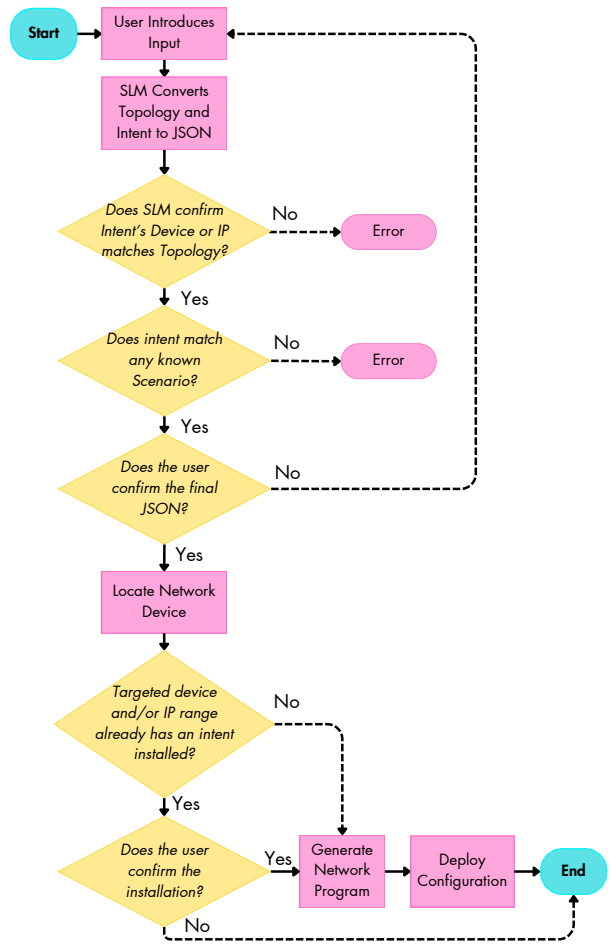


Fig. 2: Flow-chart describing the steps involved in LLNet to deploy network programs.

C. Evaluating Language Model generation

While it is still considered challenging to evaluate language models accurately due to their complexity, variability in outputs, and the possible misunderstanding of the human language, in LLNet we were able to evaluate the generated output to our expected responses since both elements are JSON files. To do so, we employed the edit-distance (*Damerau-Levenshtein*) similarity score [41], widely used in the literature [35], [42], [43] for measuring the minimum number of edits (*i.e.,* deletion, insertion, substitution and transposition) required to transform one string into another. The score is formally defined as:

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

where a and b are the strings to be compared; and i and j are the substrings' lengths of a and b, respectively.

In LLNet, we adopted this metric by merging it into three main steps: (i) we first parse both JSON files into corresponding data structures (*e.g.,* dictionaries, lists) for easier manipulation; (ii) then, we normalize the resulting data structure to ensure semantically equivalence between them; (iii) finally, we apply Eq.(1) and normalize the result to determine the minimum edit operations required to make the two structures identical. The resulting score will then range from $[0, 1]$ with 0 representing two identical structures and 1 representing completely different structures. Normalization is essential to ensure compatibility with the adopted Langchain tool, which operates on a similarity scale between 0 and 1 [44]. For example, with the intent used before: *"Block traffic from 10.0.0.5 on Switch 7."*, LLNet's Intent Converter module would translate it into a JSON representation, such as "goal": "blockTraffic", "switch_id": 7, "ip_source": "10.0.0.5", "ip_dest": "any", which completely aligns with the expected response, hence, because no modifications (*i.e.,* insertion, deletion, substitution, transposition) are needed, the *Damerau-Levenshtein* distance score is 0. However, assuming the Intent Converter module returned a different JSON representation, such as "goal": "blockTraffic", "switch_id": 3, "ip_source": "10.0.0.5", "ip_dest": "any", then it would not have matched the expected response due to the "switch_id" field. In this case, there is one substitution operation required to transform the incorrect response into the expected one, hence the *Damerau-Levenshtein* raw distance score is 1, which, when normalized by dividing by the maximum string length (*i.e.,* 81), results in a score of 0.012.

It is important to note that other metrics, such as BLEU [45] and METEOR [46], were not considered as they evaluate generated sequences by comparing them against reference texts, focusing on token or *n*-gram (*i.e., n*-consecutive items in the text) overlap. While effective for assessing linguistic structure and vocabulary usage, these metrics are less suitable for LLNet's objectives, which prioritize understanding semantic alignment and structural differences over lexical choice. Additionally, the *n*-granularity of these metrics does not align with the fine-grained evaluation required for JSON representations, where even small errors (*e.g.,* an IP address or switch ID) can affect the desired network behavior and overall performance.

V. Managing Softwarized Network Infrastructure

In this paper, we specifically consider the Software Defined Networking (SDN) scenario, which is built on the paradigm of decoupling the control-plane from the data plane, offering dynamic network programmability and flexibility. In particular, every device can be seen as composed of three main logical levels: *Data Plane*, the lower level of the entire stack, responsible for forwarding traffic, executing few and simple instructions such as receiving, forwarding, and transmitting. *Control-plane*, the intermediate level of the stack, responsible for managing and controlling the overall behavior of the network, including routing, signaling, configuration, and management of network devices. *Management Plane*, the higher level of the stack, which deals mainly with administrative tasks, monitoring, and maintaining the network infrastructure.

By logically and physically separating these planes, SDN facilitates automation, flexibility, and customization of network management through software applications and scripts. Through 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. This approach also favors our IBN solution.

The available protocols/languages share a "top-down" approach where programmers define features, which are then (compiled and) deployed onto the device, utilizing programmable blocks within compatible chips. However, they differ in several ways. P4, for example, is a popular domain-specific programming language [47], [48] 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. Similar to P4, eBPF (extended Berkeley Packet Filter) enables writing programs that run directly in the Linux kernel space [49]. This grants unprecedented access and control over the system, unlocking many possibilities, such as inspecting, modifying, and influencing various aspects of the system's inner workings, all while preserving security and performance. However, all the eBPF programs are written in a subset of C and then compiled into a BPF instruction bytecode. This process completely differs from the P4 building block (*i.e.,* Parser, Match-Action Tables, Deparser), posing an additional challenge to the network programmers.

Despite the distinct levels of expressiveness between P4 and eBPF, a considerable convergence exists in their functionalities, notably within the realm of processing network packets. The P4 backend for eBPF [50] 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 [51] 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 [52]. This type of switch is embedded directly within the kernel and allows 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 eBPF for advanced packet manipulation. 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 NIKSS environment but restricts integration with broader P4 ecosystem tools.

VI. Working Example

In this section, we illustrate in Fig. 3 a step-by-step example of LLNet, starting from the input provided by the user (*i.e.,* network topology and intent) and concluding with the generated network configurations. We designed our solution to be interactive with the user, incorporating a messaging-style interface in the form of a chatbot. The user is initially prompted to create the network environment by defining a topology where all intents will be installed (step **(1)**). Network topology can be defined as a JSON file or Mininet schema. After this first step, the user is asked to insert the intent, where a possible example is "Block all incoming packets from 10.0.0.5 on switch 7" **(2)**. These two entries, the provided topology and the corresponding intent, will then be forwarded to the *Intent Converter* **(3)**, which interprets the given topology by converting it into an extracted JSON file that shows the relationship between each network element. The presence of this *Verifier* ensures that the requests to the adopted SLM are done only if necessary, avoiding useless and expensive calls to the model. Secondly, the adopted SLM tokenizes the introduced intent and verifies that the recognized network elements (*i.e.,* "10.0.0.5" and "switch 7") are present in the provided topology. Finally, it checks whether the introduced intent matches one of the known scenarios, in this case "blockTraffic". After this step, the chatbot interface displays the structured JSON for user approval **(4)**. If not confirmed, the user is prompted to re-introduce the intent, which will follow the process as before (**5.1**). If the extracted elements are correct **(5.2)**, the response will go through a conflict detection structure within the *Network Manager* which verifies whether an intent has already been installed on the targeted switch (*i.e.,* "switch 7") or if the IP address (*i.e.,* "10.0.0.5") is already affected by another intent (*i.e.,* if a previous intent involved that IP address). Since the intent does not involve any previously introduced one, the *Network Manager* automatically generates the necessary rules matching the main goal, *e.g.,* P4 rules, and deploys them via the appropriate API, *e.g.,* P4 Runtime.

Aware of the importance of ensuring that each introduced intent successfully respects what is requested, despite the network conditions [13], [18]. In LLNet, we provide a monitoring feature that allows users to retrieve stats regarding each introduced intent by leveraging their control-plane frameworks (*e.g.,* Ryu, P4Runtime). These stats are automatically displayed after each introduced intent **(6)**, and can be easily adjusted through the corresponding API according to what needs to be shown (*e.g.,* # Matched Packets, # Packets for each port). It is important to note that the stats are shown on a separate thread running in the background, not limiting the possibility of introducing other intents.

An example of this feature is shown in Fig. 4, where, for simplicity, we plot the displayed stats for three different introduced intents over a 50-second period. In this case, we instructed LLNet to block packets coming from the same IP address (*i.e.,* 10.0.0.5) on three different switches (*i.e.,* 1, 3, 7) and randomly sent ICMP packets between our topology using the ping network tool. As visible from the figure, the user feedback provides a graphical representation of the corresponding number of matched (and blocked) packets for each introduced intent, allowing an efficient monitoring system throughout our network and a fast reaction if necessary. It is important to note that the monitoring feature in LLNet only tracks the progress of the implemented function (in this case, a firewall), and the evaluation of its correctness (whether it performs as intended) is assessed by the network administrator.

VII. Experimental Evaluation

In this section, we study the effectiveness of LLNet from three angles: the network performance, the accuracy of the adopted LLM, and the sustainable analysis with the SLM. Network application performance was examined by simulating various network environments, while LLM tasks mainly focus on specific use cases to better validate the accuracy.

A. Experimental Settings

In this section, we describe the experimental settings focusing on the adopted language models and network scenario.

Network Settings. We developed LLNet to be flexible and adaptable to different network planes; however, in our prototype, we mainly consider two scenarios: OpenFlow-based switches controlled with Ryu or FloodLight framework, and P4-based switches for custom packet processing determined by P4Runtime and eBPF for direct observability within the Linux kernel. All evaluations were conducted using the Mininet network emulator, which enables the emulation of network environments. In the case of P4 and eBPF, the switches were NIKSS vSwitches, and in the case of eBPF, all the intents were first translated into P4-structured rules that could have been injected into P4 programming codes or, with the P4-eBPF framework [50]. The performance of these network applications is then analyzed on different topologies, as shown in Fig. 5: a partial mesh topology (Fig. 5a), a star-mesh hybrid topology (Fig. 5b) and a leaf-spine topology (Fig. 5c). The input topology is provided in input to the LLM.

Language Model Settings. LLNet's generative model was deployed in Python adopting two main large models: a proprietary one, as Gemini version 1-Pro from Google, and an open-source model, Llama version 3-8B-instruct from Meta. The choice of adopting two LLMs is motivated by the complementary benefits brought by each model: while Gemini 1-Pro is known for its advanced reasoning capabilities and higher context lengths, Llama3-8B-instruct is known for its lower latency and compatibility for more limited hardware configurations [53]. As for the small model, we adopted the open-sourced Phi version 3-mini-instruct from Microsoft, which can limit the model size, as well as the usage costs and energy. All the open-sourced models (*i.e.,* Llama3-8B and Phi-3) are locally deployed via the *Ollama* API [54], while the experiment settings (*i.e.,* hyperparameters, prompting templates) and Gemini are called and configured via the *LangChain* framework, an open-source software library that facilitates the

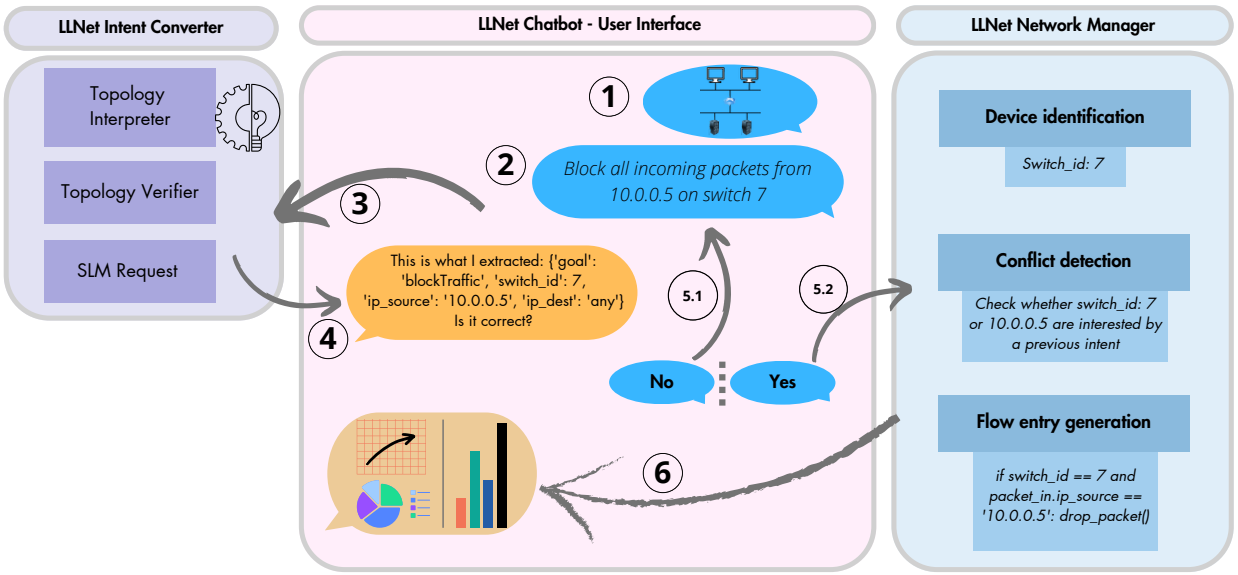


Fig. 3: A working example of LLNet when an intent is introduced. The chatbot interface gives an interactive demonstration of how the system responds to user inputs.

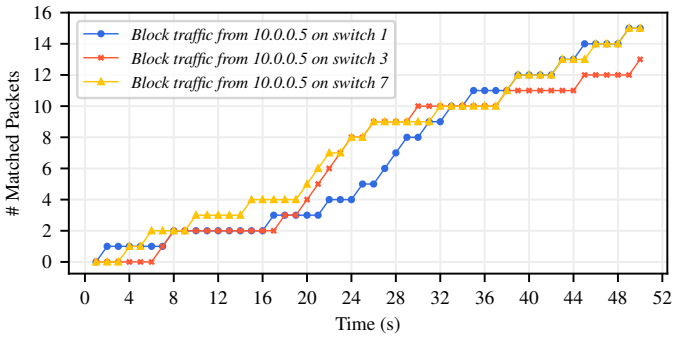


Fig. 4: LLNet allows to keep track of the introduced intents using a monitoring feature through API.

Transpiler	% Correctly Dropped Packets with FloodLight	% Correctly Dropped Packets with eBPF
LLNet	98.5 ± 0.05	98.7 ± 0.02
ChatGPT	94.9 ± 1.01	93.2 ± 3.02
Llama	92.4 ± 3.02	94.8 ± 4.03
Gemini	91.7 ± 5.02	97.1 ± 2.01

TABLE II: Results of Packet Dropped for diverse LLM solutions. Results validate our approach to adopt an intermediate representation used by network programs.

integration and management of LLM [55]. Finally, we did not re-train these models to favor portability, but (as explained in Sec. IV) we provide few-shot examples when prompting user intents. To achieve the most deterministic output possible, we set the LLM temperature to 0, but in the rest of the section, we also analyze the correctness of the translation at varying temperatures (as well as with and without prompting techniques).

User Input. The evaluations were performed using three main scenarios: (i) an access control list (ACL) firewall, where the main goal was to block packets coming/ going from/to specific IP addresses; (ii) a rate limiter rule for traffic flow, to ensure fair bandwidth distribution; and (iii) a load profiler mechanism, to have traffic prioritized according to specific weights. The hand-crafted dataset has been produced using a tool called Chatito [56]. 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. Using Chatito, we generated 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. We then asked *GPT-4o* to expand the created dataset by rephrasing the intents generated with Chatito and have a more complex dataset that could better identify possible specified intents. The final dataset is composed of $10,000$ blocking traffic intents (*e.g.,* "Block traffic from 10.0.0.3 to 10.0.0.4", "Prevent communication from 10.0.0.3 to any destination on switch 9"), $10,000$ rate limiter intents (*e.g.,* "Could you limit the network speed to 900 Kbit/s on all links of switch 2?", "Please set a maximum traffic rate of 550 Kbit/s in the network"), and $10,000$ load profiling intents (*e.g.,* "Modify traffic flow on switch 3 by enabling traffic in link 2 twice of link 1", "Configure switch 5 to prioritize traffic with weights 0.3, 0.6, and 0.1 on ports 1, 2, and 3 respectively") for a total of $30,000$ intents. The experiment has been conducted on the platform of Langsmith, a tool offered by Langchain to evaluate and monitor the LLM-based application.

B. Softwarized Firewall

For one of our use-case scenarios, we considered access control methods onboard a firewall as a way of pre-filtering the access using access control lists (ACLs). Our implementation combines matching rules with Bloom filters, a probabilistic data structure known for its fast computation time and efficient use of memory that allows the reduction of the overhead that a

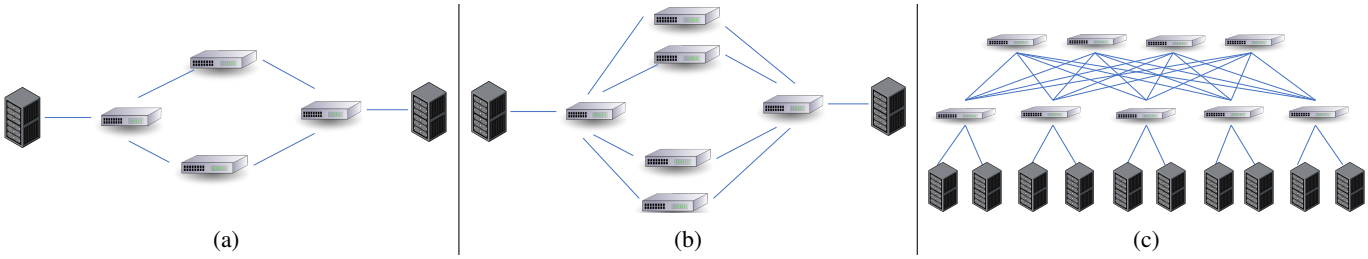


Fig. 5: Network topologies used throughout the experimental evaluation: (a) partial mesh, (b) star-mesh hybrid and (c) leaf-spine topology.

firewall can face when many policies are installed [57]. Despite the potential for false positives with the Bloom filter, this combination has been shown to be effective and widely adopted in various network scenarios. For both implementations, on the control-plane and data-plane, we identified the switch on which the rule had to be installed and dropped the packets coming from/going to the specified IP address. We tested this network function when sending ICMP packets across the network with the *ping* utility on both network planes: the control plane, with FloodLight, and the data plane, with eBPF. We compared LLNet's performance against other well-known transpilers (*i.e.,* ChatGPT, Llama, and Gemini) when used on their own, *i.e.,* without providing additional prompting or processing intermediate structure, and report the result in Table II. For the FloodLight version, we asked LLNet and other LLMs to generate a program in the control plane, which only implicitly reflects in a data-plane program that, in this case, is OpenFlow. Thus, results refer to the number of packets registered by the SDN controller. Conversely, in a data plane approach, the eBPF code is the output of the transpile process, and measurements occur directly in the network devices. A FloodLight-based approach, although less permanent, allows easier management as the SDN can actively monitor all switch ports.

As visible from the Table, LLNet was able to achieve the highest percentile of correctly dropped packets on both network scenarios, FloodLight and eBPF, while also reducing the variability of the generated output. Using LLMs to directly transpile user intents to network programs can lead to runs where the code seems accurate but diverges from the original intents, as empirically measured. On the contrary, LLNet, with appropriate prompting and by converting the intent with an intermediate structure, is able to first correctly translate the received intent, then generate the needed network configurations, leading to consistent accuracy over both a control-plane and data-plane transpiled programs.

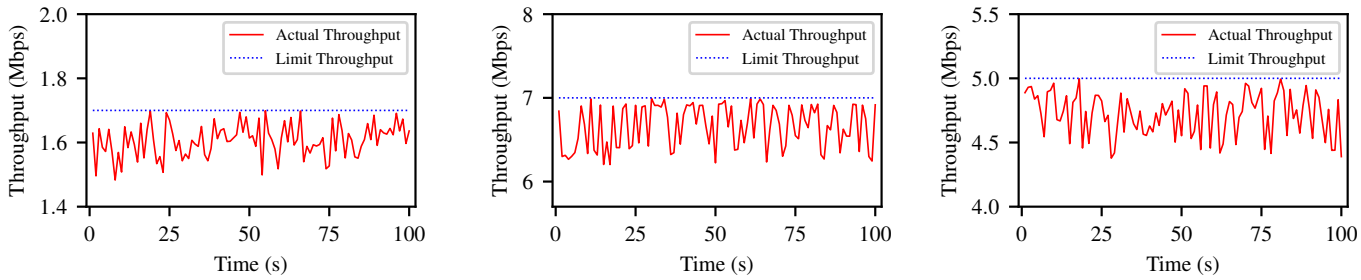
C. Rate Limiter

A rate limiter is a tool designed to control the number of requests that users can send within a time interval, and it is usually adopted to ensure fair bandwidth distribution, to maintain the stability and performance of networks while also protecting from unintentional overuses [58]. A possible user intent can be, for example, "Can you cap the bandwidth of the inbound traffic to 7 Mbps?" or "Set a maximum traffic rate of the outbound traffic to 15 Mbps." When an intent is inserted to limit traffic bandwidth to or from a specific switch, we have two possible solutions depending on the network plane of interest. Regardless of the technology, switches can track the rate within a time interval of 1 s and enforce limits on the rate of requests made using a sort of leaky bucket algorithm, a well-known approach in the literature, where tokens are added to the bucket at a rate that is used to control the rate of packets.
In the control-plane. When a rate-limiting rule is applied in a control plane-managed network using Ryu, a specialized meter entry is configured with the specified rate. This configuration generates an *OFPMeterMod* message, which is then populated with the meter settings and prepared to be sent to the targeted switch, identified by its datapath ID (DPID). The controller ensures that the desired rate is enforced on the intent's specified switch, thereby managing traffic flow appropriately.
In the data-plane. When applied to the dataplane, an API integrated with the Mininet emulator is invoked. Using the NIKSS libraries, we leverage built-in constructs called "meters" to implement rate limiting on the ports of the targeted switch using the *nikss-ctl meter update pipe* structure. This allows precise control over the traffic rates at the data plane level, ensuring efficient bandwidth management across the emulated network.

To test both scenarios, we used the *iperf3* tool, which simulates realistic traffic on the network. Finally, we report the results in Fig. 6 and Fig. 7. It is visible from both figures that for all the targeted switches, the desired throughput is satisfied, never exceeding the limit set by the user.

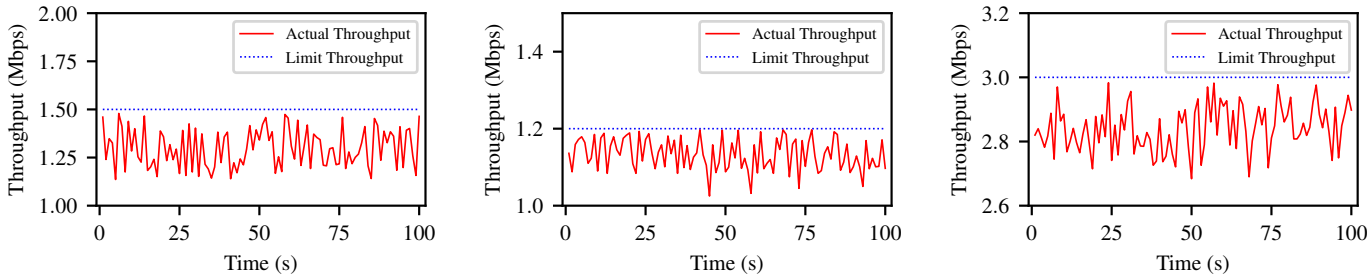
D. Load Profiling

Load profiling (LP), a method of categorizing network traffic based on its characteristics and demands, plays a crucial role in optimizing performance, particularly in large-scale environments such as data centers. By employing these algorithms, network administrators can control how traffic is distributed, ensuring optimal performance and resource utilization. Such approaches allow for more dynamic traffic management, where paths or links are chosen based on predefined weights, demonstrating the network's adaptability and ultimately aiming to prevent congestion and improve data transmission efficiency. This not only helps prevent congestion and improves data transmission efficiency, but also plays a key role in identifying network bottlenecks and enhancing the overall network performance. Furthermore, load profiling ensures that traffic distribution aligns with Quality of Service (QoS) requirements, thereby maximizing the overall network



(a) (b) (c)

Fig. 6: Throughput when the rule is applied on the control-plane with Ryu on switch 1 (a), on switch 6 (b), and on switch 7 (c).



(a) (b) (c)

Fig. 7: Throughput when the rule is applied on the data plane with P4 on switch 1 (a), on switch 6 (b), and on switch 7 (c).

performance [59]. We assume user intents convey information on the desired traffic profiles over links.

In LLNet, we implemented the load profiling in both control-plane and data-plane. We then simulated realistic network traffic using the *iperf3* tool, allowing us to verify that the load is distributed as desired.

In the control-plane. The LP rule on the target switch is installed by creating multiple buckets within a group. Each of these buckets has an associated weight and an action that forwards packets to a specific port according to the desired weight. The group is then updated using a *OFPGroupMod* message, and finally, a new flow entry is added to forward matching packets to the group. We simulate one user request with a diverse desired load on all links of the network, generate user traffic with *iperf3*, collect the network performance, and report the results in Fig. 8. It is visible from the figure that the rule was correctly satisfied across all ports of the targeted switches: the monitored load (in green) satisfied most of the time the desirable load (in white). However, we noticed a decrease in throughput when the rule was applied. This behavior can be caused by the constant interaction with the controller for every received packet. At higher traffic rates and with multiple concurrent connections, this can lead to congestion.

In the data-plane. The LP rule is applied to the dataplane with the nikss-ctl API to interact with the targeted switch. In detail, we populate a table by adding a new entry corresponding to the specified switch and ports' weights. This entry is then sent to the switch, enabling it to handle incoming packets using a weighted random choice algorithm to match the defined ports' selection. We tested the injected rule using

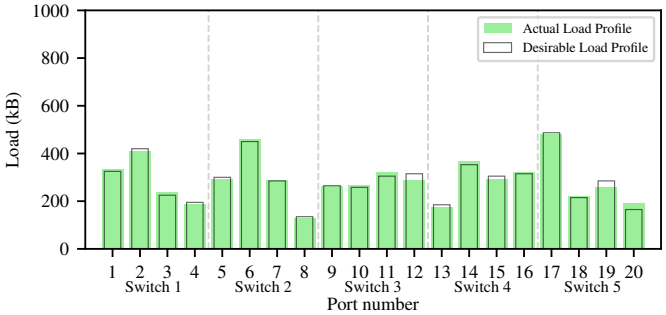


Fig. 8: Traffic follows the desired load profiling rule specified by the intent when installed on the control-plane.

the *iperf3* command tool and reported the difference between the actual load (in kB) and the desired one specified with the intent in Fig. 9. As visible from the figure, the actual load profile (in orange) satisfies the desired one (in white) for all the switches of our network. This alignment indicates that the installed weights on the different ports effectively satisfy the desired load conditions specified by the intent, ensuring optimal functionality and efficiency throughout the entire network.

E. LLM Effectiveness

When working with LLMs, it is essential to obtain the best results (*e.g.,* highest accuracy), while also maintaining contained costs and limited resource utilization. For this reason, we investigated two parameters that impact how our model translates intents: (i) Prompting techniques, e.g., few-shot examples, known to guide the model's understanding to

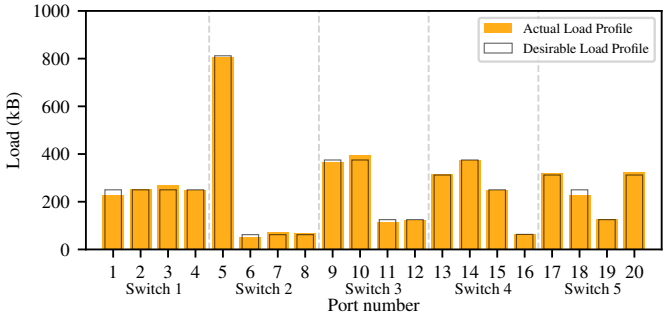


Fig. 9: Traffic follows the desired load profiling rule specified by the intent when installed directly on the data-plane.

obtain better responses [60], [61]; (ii) Sampling Temperature, a model's hyperparameter that is known to affect the model's results [62], [63]. The obtained results are then evaluated with the expected responses of the created dataset (as explained in Sec. VII). This analysis allowed us to address two main challenges: the human language's ambiguity and the potential instability observed in LLMs [64], which could lead to the generation of inconsistent but plausible information, a phenomenon known as hallucination. This phenomenon impacts the accuracy of the resulting formal specification, potentially leading to unexpected responses.

Few-Shot examples. A few-shot example is a prompting technique that involves providing a limited number of examples, along with a prompting template, to an LLM with the goal of guiding it to a specific scenario so that the model's results are more aligned with the user's intentions. In order to prove the efficiency of this technique in reducing ambiguity and improving the model's outputs, in LLNet we investigated the performance and costs *with* and *without* these advanced prompting techniques. We used the aforementioned dataset as input, and for each analyzed network function, we repeated the experiments three times. Finally, we report the obtained results when using this prompting technique in Table III and without it in Table IV, respectively. The edit distance score is obtained as in Eq. (1), where a lower value indicates higher accuracy when translating the input. At lower percentiles, translations likely contain more errors, requiring a greater number of string modifications (*e.g.,* insertions, deletions) to match the given reference, leading to a higher edit distance. While, as the percentile increases, fewer modifications are needed, resulting in a lower edit distance score.

As visible from Table III, for all the studied network functions, the overall cost (computed as the sum of the input and output costs) and latency (measured as the time to send and receive the response) are slightly higher than those obtained without the prompting technique, shown in Table IV. This is due to the additional input from the prompting template, which raises the number of input tokens, thereby increasing input size, latency, and overall cost. However, this increment is very limited, and it also brings more correctly translated intents, with a notable reduction of more than 50% for the edit distance score. Thus, considering the low overall cost and small latency difference, the use of examples justifies the adoption. Interestingly, the non-linear relationship between the correctly translated intents and the edit-distance score is due to the Damerau-Levenshtein definition (see Sec. IV-C).

It is important to note that the cost estimations were averaged by taking into consideration the two adopted LLMs. For Gemini 1-Pro, the prices described on the model's webpage [65]: $\$0.0003125$ for 1000 input characters and $\$0.00125$ for 1000 output characters, where 4 characters correspond approximately to 1 token. For Llama3-8B, we considered the prices described in [66]: 0.14 for $1M$ input tokens and 0.20 for $1M$ output tokens. Due to the very low prices, the open-sourced nature, and the possibility of running locally, we decided to exclude Llama's associated costs from Table III and Table IV, and considered only those from Gemini.

Sampling Temperature. LLMs are known to be flexible and to be personalized to different computing requirements and use-case scenarios. This can be achieved by adopting suitable models, fine-tuning them with appropriate datasets, and adjusting hyperparameters based on the desired output. One such adjustable hyperparameter is the sampling temperature, which controls the randomness of the generated output sequence during inference [67], [68]. Due to the importance of this hyperparameter, researchers have studied how varying the sampling temperature affects the accuracy of the LLMs' output [62], [63], [69].

For this reason, we decided to test LLNet at an incremental sampling temperature of 0.1, from 0.0 to 1.0 (due to the temperature limit of the Langchain API for Gemini [70]). We used the previously described generated dataset as input for our model and repeated the experiment three times to reduce the possibility of ambiguity. We tested two different scenarios: with few-shot examples and without, keeping the prompt template to support the output. Finally, we report the results in Fig. 10. As visible from the figure, the presence of advanced prompting techniques like few-shot examples helps achieve the best accuracy (in terms of the number of correctly translated intents) when compared to the solution without them, at all sampled temperatures. Furthermore, it is noticeable that the accuracy starts decreasing whenever the temperature increases, a sign that an optimal temperature can be found at lower values. Since we provide few-shot examples to guide the model, we aim for outputs that closely follow these patterns, which is essential for accurately identifying the network elements specified by the introduced intent. Similar behavior is also shown in other studies where the accuracy is highly dependent on the sampled temperature and it decreases at higher temperature values [14], [62], [63].

F. Impact of SLMs on Performance

Lastly, we study how to possibly reduce the energy consumption while preserving the accuracy. In an era where LLMs are widely used across various domains, their energy consumption has become a significant concern, especially in a society that prioritizes sustainable computing [71]. Trained on billions of parameters, these models are known to be extremely high-consuming for both computational resources and energy, posing challenges and debates for the sustainability, their environmental impact, and consequently, the ethical considerations of their adoption [72]. For this reason, we examined

Network Function	Avg. Latency (*s*)	Avg. Intent Correctly Trans. (%)	Avg. Edit Distance Score	Avg. Tokens (Input)	Avg. Tokens (Output)	Avg. Input Cost (USD)	Avg. Output Cost (USD)	Avg. Energy Cons. (kWh)
ACLs Firewall	3.76	83.6	0.06	619	23	0.000774	0.000115	5.18e-06
Rate Limiter	2.58	88.5	0.04	615	12	0.000769	0.000060	5.22e-06
Load Profiling	3.71	92.1	0.03	620	17	0.000776	0.000088	6.69e-06

TABLE III: LLNet's performance and cost metrics when an LLM is adopted with Few-shot examples. All the provided intents are correctly translated at a negligible cost and latency (when compared to Table IV).

Network Function	Avg. Latency (*s*)	Avg. Intent Correctly Trans. (%)	Avg. Edit Distance Score	Avg. Tokens (Input)	Avg. Tokens (Output)	Avg. Input Cost (in USD)	Avg. Output Cost (in USD)	Avg. Energy Cons. (kWh)
ACLs Firewall	2.86	79.2	0.07	455	23	0.000570	0.000115	4.87e-06
Rate Limiter	1.97	63.2	0.10	451	12	0.000565	0.000060	5.23e-06
Load Profiling	2.64	75.8	0.08	456	17	0.000571	0.000088	5.77e-06

TABLE IV: LLNet's performance and cost metrics when an LLM is adopted without Few-shot examples. Despite their smaller latency and cost (when compared to Table III), the absence of advanced prompting techniques leads to a significantly lower number of correctly translated intents.

Network Function	Avg. Latency (*s*)	Avg. Intent Correctly Trans. (%)	Avg. Edit Distance Score	Avg. Tokens (Input)	Avg. Tokens (Output)	Avg. Input Cost (USD)	Avg. Output Cost (USD)	Avg. Energy Cons. (kWh)
ACLs Firewall	1.78	86.3	0.04	619	23	0.000081	0.000012	1.64e-06
Rate Limiter	1.67	94.6	0.01	615	12	0.000080	0.000006	1.29e-06
Load Profiling	1.86	87.7	0.02	620	17	0.000081	0.000009	1.42e-06

TABLE V: LLNet's performance and cost metrics when an SLM is adopted with Few-shot examples. The SLM allows for a decrease in energy consumption while maintaining an exceptional intent translation rate.

Network Function	Avg. Latency (*s*)	Avg. Intent Correctly Trans. (%)	Avg. Edit Distance Score	Avg. Tokens (Input)	Avg. Tokens (Output)	Avg. Input Cost (USD)	Avg. Output Cost (USD)	Avg. Energy Cons. (kWh)
ACLs Firewall	1.68	66.3	0.09	455	23	0.000059	0.000012	1.86e-07
Rate Limiter	1.62	78.4	0.08	451	12	0.000059	0.000006	1.93e-07
Load Profiling	1.69	67.9	0.09	456	17	0.000059	0.000009	2.05e-07

TABLE VI: LLNet's performance and cost metrics when an SLM is adopted without Few-shot examples. The absence of advanced prompting techniques results in lower intents being correctly translated.

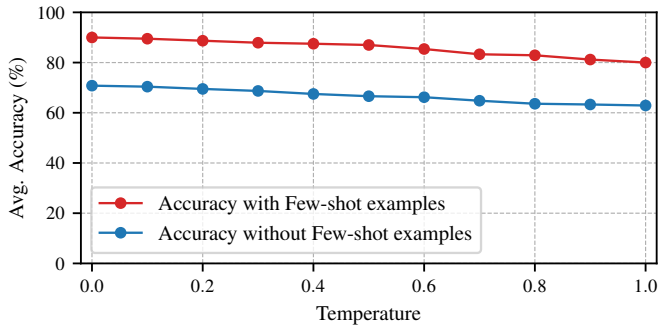


Fig. 10: Impact of sampling temperature in translating the generated dataset with and without Few-shot examples.

the sustainability aspect of LLNet by analyzing the average energy consumption used during our experiments. To do so, we exploited the MELODI energy analysis software [73] that allowed us to monitor the CPU's energy consumption of LLMs inference by leveraging the *Scaphandre* power usage tool [74]. It is important to mention that the analysis was conducted on an Intel i7 8-core CPU with $32GB$ of RAM and only analyzed Llama3-8B due to our laptop's hardware limitations.

We investigate how lighter models, such as SLMs, can reduce the overall carbon footprint. In particular, in LLNet we adopted Microsoft's Phi3-mini-instruct with 3.8B parameters only, and we performed the evaluation *with* and *without* advanced prompting techniques such as few-shot examples and reported the results in Table V and Table VI. As visible from Table V, the adoption of a lighter model significantly reduces energy consumption while still maintaining an excellent percentile of intent translation for all network functions. The adoption of this SLM also results in a lower overall cost and lower energy consumption when compared to the LLM-based version. It is important to notice that prices are extracted from the model's webpage updated on the date of access [75]: 0.00013$ for 1000 input tokens, and 0.00052$ for 1000 output tokens. The same energy reduction can be seen in Table VI where no advanced prompting techniques, such as few-shot examples, are provided. The table shows a decreased energy consumption compared to an LLM implementation. In particular, compared to Table III and Table IV, the energy consumed is reduced by 91.86% on average, which can be extremely significant if LLNet is applied to bigger topologies where more intents are needed to personalize each network device. At the same time, and somehow surprisingly, the accuracy and the distance score can even improve in two use cases. In the third use case (load profiling), the accuracy minimally downgrades (*i.e.*, 3.9%). In conclusion, using SLM in LLNet is considered a valid solution and trade-off between correctness and sustainability, being also able to overcome larger models such as Llama3-8B and Gemini 1-Pro, as also recently evinced in other studies [76]. This approach

provides higher flexibility and efficiency even for less-capable hardware, while also reducing our overall carbon footprint.

VIII. Conclusion

In this paper, we presented LLNet, our approach to IBN that leverages the recent advances of LLMs to facilitate the programming of network management and the deployment of network management schema. By also exploiting the Software-Defined Networking (SDN) paradigm, network devices can be programmed via writing code, and LLNet facilitates this operation, making it accessible even for non-experts. We design the solution in a way that can accurately generate network programs even when using an SLM to replace resource-consuming LLMs and make the approach more practical. The system also provides a way to monitor whether the deployed intent actually meets the intended goal. Results validate the accuracy of the SLM as an interpreter and the sustainability of our solution compared to other LLM-based techniques. The domain of LLM-assisted tasks is growing fast, and we plan to explore the daily advances in this area to extend LLNet to more network scenarios and with techniques that can further reduce energy consumption.

References

[1] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations: brief reflections on abstractions for network programming," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 5, pp. 104–106, 2019.

[2] S. K. Mani, Y. Zhou, K. Hsieh, S. Segarra, T. Eberl, E. Azulai, I. Frizler, R. Chandra, and S. Kandula, "Enhancing network management using code generated by large language models," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*. ACM, 2023, pp. 196–204.

[3] A. Sacco, F. Esposito, and G. Marchetto, "Restoring Application Traffic of Latency-Sensitive Networked Systems using Adversarial Autoencoders," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2521–2535, 2022.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A General Approach to Network Configuration Verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 2017, pp. 155–168.

[5] A. Sacco, M. Flocco, F. Esposito, and G. Marchetto, "Supporting Sustainable Virtual Network Mutations with Mystique," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2714–2727, 2021.

[6] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, 2018, pp. 609–623.

[7] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models (2023)," *arXiv preprint arXiv:2302.13971*, 2023.

[8] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[9] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "PaLM: Scaling Language Modeling with Pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[11] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024.

[12] M.-V. Dumitru, V.-A. Bădoiu, A. M. Gherghescu, and C. Raiciu, "Generating P4 Dataplanes Using LLMs," in *IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2024, pp. 31–36.

[13] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "Llm-based policy generation for intent-based management of applications," in *2023 19th International Conference on Network and Service Management (CNSM)*. IEEE, Oct. 2023. [Online]. Available: http://dx.doi.org/10.23919/CNSM59352.2023.10327837

[14] B. Ifland, E. Duani, R. Krief, M. Ohana, A. Zilberman, A. Murillo, O. Manor, O. Lavi, H. Kenji, A. Shabtai *et al.*, "Genet: A multi-modal llm-based co-pilot for network topology and configuration," *arXiv preprint arXiv:2407.08249*, 2024.

[15] C. Wang, M. Scazzariello, A. Farshin, D. Kostic, and M. Chiesa, "Making network configuration human friendly," *arXiv preprint arXiv:2309.06342*, 2023.

[16] A. Mekrache, A. Ksentini, and C. Verikoukis, "Intent-Based Management of Next-Generation Networks: an LLM-Centric Approach," *IEEE Network*, vol. 38, no. 5, pp. 29–36, 2024.

[17] A. Mekrache and A. Ksentini, "LLM-enabled Intent-driven Service Configuration for Next Generation Networks," in *IEEE 10th International Conference on Network Softwarization (NetSoft)*. IEEE, 2024, pp. 253–257.

[18] A. Angi, A. Sacco, F. Esposito, G. Marchetto, and A. Clemm, "NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches," *IEEE Communications Magazine*, vol. 62, no. 6, pp. 28–34, 2024.

[19] R. Eldan and Y. Li, "Tinystories: How small can language models be and still speak coherent english?" *arXiv preprint arXiv:2305.07759*, 2023.

[20] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency (FAccT)*, 2021, pp. 610–623.

[21] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," *Advances in Neural Information Processing Systems (NeurIPS 2023)*, vol. 36, 2024.

[22] A. Leivadeas and M. Falkner, "A survey on intent-based networking," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 625–655, 2022.

[23] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, Lumi! Using Natural Language for Intent-Based Network Management," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 625–639.

[24] R. Caldelli, P. Castoldi, M. Gharbaoui, B. Martini, M. Matarazzo, and F. Sciarrone, "On helping users in writing network slice intents through nlp and user profiling," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 545–550.

[25] C. H. Cesila, R. P. Pinto, K. S. Mayer, A. F. Escallón-Portilla, D. A. Mello, D. S. Arantes, and C. E. Rothenberg, "Chat-ibn-rasa: Building an intent translator for packet-optical networks based on rasa," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 534–538.

[26] H. Lyu, S. Jiang, H. Zeng, Y. Xia, and J. Luo, "Llm-rec: Personalized recommendation via prompting large language models," *arXiv preprint arXiv:2307.15780*, 2023.

[27] P. B. Mensah, N. S. Quao, and P. G. C. E. Group, "Can large language models provide emergency medical help where there is no ambulance? a comparative study on large language model understanding of emergency medical scenarios in resource-constrained settings," *medRxiv*, pp. 2024–04, 2024.

[28] O. G. Lira, O. M. Caicedo, and N. L. da Fonseca, "Large Language Models for Zero Touch Network Configuration Management," *arXiv preprint arXiv:2408.13298*, 2024.

[29] A. Fuad, A. H. Ahmed, M. A. Riegler, and T. Čičić, "An intent-based networks framework based on large language models," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. IEEE, 2024, pp. 7–12.

[30] S. Chakraborty, N. Chitta, and R. Sundaresan, "Automation of Network Configuration Generation using Large Language Models," in *20th International Conference on Network and Service Management (CNSM)*. IEEE, 2024, pp. 1–7.

[31] M.-V. Dumitru, V.-A. Bădoiu, and C. Raiciu, "Prose-to-p4: Leveraging high level languages," *arXiv preprint arXiv:2406.13679*, 2024.

[32] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa, "Netconfeval: Can llms facilitate network configuration?" *Proc. ACM Netw.*, vol. 2, no. CoNEXT2, june 2024. [Online]. Available: https://doi.org/10.1145/3656296

[33] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Who answers it better? an in-depth analysis of chatgpt and stack overflow answers to software engineering questions," *arXiv preprint arXiv:2308.02312*, 2023.

[34] M. Riftadi and F. Kuipers, "P4I/O: Intent-Based Networking with P4," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 438–443.

[35] J. Anderson, J. T. Tarigan, and A. Sharif, "Damerau-levenshtein distance and cosine similarity to select the optimal word in word typing game," in *AIP Conference Proceedings*, vol. 2987, no. 1. AIP Publishing, 2024.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[37] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[39] D. Yugeswardeenoo, K. Zhu, and S. O'Brien, "Question-analysis prompting improves llm performance in reasoning tasks," *arXiv preprint arXiv:2407.03624*, 2024.

[40] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff *et al.*, "The prompt report: A systematic survey of prompting techniques," *arXiv preprint arXiv:2406.06608*, 2024.

[41] L. Boytsov, "Indexing methods for approximate dictionary searching: Comparative analysis," *Journal of Experimental Algorithmics (JEA)*, vol. 16, pp. 1–1, 2011.

[42] A. Nuraminah and A. Ammar, "Damerau-levenshtein distance algorithm based on abstract syntax tree to detect code plagiarism," *Scientific Journal of Informatics*, vol. 11, no. 1, pp. 11–20, 2024.

[43] C. Zhao and S. Sahni, "String correction using the damerau-levenshtein distance," *BMC bioinformatics*, vol. 20, pp. 1–28, 2019.

[44] JsonEditDistanceEvaluator in LangChain. Accessed: 2024-1-29. [Online]. Available: https://api.python.langchain.com/en/latest/evaluation/langchain.evaluation.parsing.json_distance.JsonEditDistanceEvaluator.html

[45] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[46] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[47] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[48] A. Angi, A. Sacco, F. Esposito, and G. Marchetto, "Routing with ART: Adaptive Routing for P4 Switches With In-Network Decision Trees," in *GLOBECOM 2024-2024 IEEE Global Communications Conference*. IEEE, 2024, pp. 3291–3296.

[49] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[50] PSA implementation for eBPF backend. Accessed: 2024-6-7. [Online]. Available: https://github.com/p4lang/p4c/tree/main/backends/ebpf/psa

[51] eBPF Backend. Accessed: 2024-6-7. [Online]. Available: https://github.com/p4lang/p4c/tree/main/backends/ebpf

[52] T. Osiński, J. Palimąka, M. Kossakowski, F. D. Tran, E.-F. Bonfoh, and H. Tarasiuk, "A novel programmable software datapath for software-defined networking," in *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2022, pp. 245–260.

[53] Artificial analysis - gemini 1-pro vs llama3-instruct-8b. Accessed: 2024-10-03. [Online]. Available: https://artificialanalysis.ai/models/gemini-pro?models_selected=llama-3-instruct-8b%2Cgemini-pro

[54] (2024) Ollama (v0.3.12). Accessed: 2024-9-28. [Online]. Available: https://github.com/ollama/ollama

[55] O. Topsakal and T. C. Akinci, "Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast," in *International Conference on Applied Engineering and Natural Sciences*, vol. 1, no. 1, 2023, pp. 1050–1056.

[56] Chatito. Accessed: 2024-6-7. [Online]. Available: https://github.com/rodrigopivi/Chatito/tree/master

[57] J.-K. Lee, T. Hong, and G. Li, "Traffic and overhead analysis of applied pre-filtering acl firewall on hpc service network," *Journal of Communications and Networks*, vol. 23, no. 3, pp. 192–200, 2021.

[58] M. Bartkov and D. Borovikov, "Selection of a suitable algorithm for the implementation of rate-limiter based on bucket4j." *International Journal of Online & Biomedical Engineering*, vol. 16, no. 4, 2022.

[59] S. Kumar, V. Maurya, and R. Gupta, "A distributed load balancing technique for multitenant edge servers with bottleneck resources," *IEEE Transactions on Reliability*, 2023.

[60] H. Ma, C. Zhang, Y. Bian, L. Liu, Z. Zhang, P. Zhao, S. Zhang, H. Fu, Q. Hu, and B. Wu, "Fairness-guided few-shot prompting for large language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 43 136–43 155, 2023.

[61] S. Gaur, A. Dagar, A. Punia, and P. Kumar, "A brief study of prompting techniques for reasoning tasks," in *NIELIT's International Conference on Communication, Electronics and Digital Technologies*. Springer, 2024, pp. 147–159.

[62] M. Renze and E. Guven, "The effect of sampling temperature on problem solving in large language models," *arXiv preprint arXiv:2402.05201*, 2024.

[63] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "Llm for soc security: A paradigm shift," *IEEE Access*, 2024.

[64] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[65] Vertex AI pricing. Accessed: 2024-9-30. [Online]. Available: https://cloud.google.com/vertex-ai/generative-ai/pricing

[66] Artificial analysis - llama3-instruct-8b prices. Accessed: 2024-10-03. [Online]. Available: https://artificialanalysis.ai/models/llama-3-instruct-8b

[67] C. Wang, X. Liu, and A. H. Awadallah, "Cost-effective hyperparameter optimization for large language model generation inference," in *International Conference on Automated Machine Learning*. PMLR, 2023, pp. 21–1.

[68] P.-H. Wang, S.-I. Hsieh, S.-C. Chang, Y.-T. Chen, J.-Y. Pan, W. Wei, and D.-C. Juan, "Contextual temperature for language modeling," *arXiv preprint arXiv:2012.13575*, 2020.

[69] M. Shen, S. Das, K. Greenewald, P. Sattigeri, G. Wornell, and S. Ghosh, "Thermometer: Towards universal calibration for large language models," *arXiv preprint arXiv:2403.08819*, 2024.

[70] Langchain chatgooglegenerativeai. Accessed: 2024-10-04. [Online]. Available: https://python.langchain.com/api_reference/google_genai/chat_models/langchain_google_genai.chat_models.ChatGoogleGenerativeAI.html

[71] M. H. Alsharif, A. H. Kelechi, A. Jahid, R. Kannadasan, M. K. Singla, J. Gupta, and Z. W. Geem, "A comprehensive survey of energy-efficient computing to enable sustainable massive iot networks," *Alexandria Engineering Journal*, vol. 91, pp. 12–29, 2024.

[72] G. Piatti, Z. Jin, M. Kleiman-Weiner, B. Schölkopf, M. Sachan, and R. Mihalcea, "Cooperate or collapse: Emergence of sustainability behaviors in a society of llm agents," *arXiv preprint arXiv:2404.16698*, 2024.

[73] E. J. Husom, A. Goknil, L. K. Shar, and S. Sen, "The Price of Prompting: Profiling Energy Use in Large Language Models Inference," *arXiv preprint arXiv:2407.16893*, 2024.

[74] B. Petit. (2023) scaphandre (v1.0). Accessed: 2024-9-27. [Online]. Available: https://github.com/hubblo-org/scaphandre

[75] Phi-3 pricing. Accessed: 2024-9-30. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/phi-3/

[76] M. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. Behl *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.

Antonino Angi received his M.Sc. degree in Computer Engineering (major in Data Science) from Politecnico di Torino, Italy in 2020, and he is currently enrolled in a Ph.D. program at the same university. His research interests include protocols for network architecture and management; applying Large Language Models (LLM) and Machine Learning algorithms to Software Defined Networks (SDN) and Intent-based Networks (IBN), used in conjunction with data-plane programming languages.

Alessio Sacco is an Assistant Professor at Politecnico di Torino. He received the M.Sc. degree (summa cum laude) and the Ph.D. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Torino, Italy, in 2018 and 2022, respectively. His research interests include architecture and protocols for network management; implementation and design of cloud computing applications; algorithms and protocols for service-based architecture, such as SDN used in conjunction with Machine Learning algorithms.

Guido Marchetto received the Ph.D. degree in computer engineering from the Politecnico di Torino, in 2008, where he is currently a Full Professor with the Department of Control and Computer Engineering. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.