OSS-GPT: An LLM-Powered Intent-Driven Operations Support System for 6G Networks

Abdelkader Mekrache
EURECOM
France
abdelkader.mekrache@eurecom.fr

Adlen Ksentini
EURECOM
France
adlen.ksentini@eurecom.fr

Christos Verikoukis
University of Patras and ISI/ATH
Greece
chverik@gmail.com

***Abstract*—With the high demands of 6G networking services in terms of Quality of Service (QoS), managing these networks requires intelligent next-generation Operations Support Systems (OSSs). According to standardization bodies such as ETSI and 3GPP, OSS must support end-to-end, cross-domain management across all 6G domains. They are making significant efforts to standardize Application Programming Interfaces (APIs) to enable Intent-Based Networking (IBN), which simplifies network management by allowing users to express their intentions in a declarative manner. However, these systems remain complex for users with limited domain knowledge who need to interact with these standardized APIs. Moreover, adding new functionalities to OSS often requires users to learn new API endpoints and structures, which can be time-consuming. To address these challenges, we propose enabling natural language interaction with OSS by leveraging Large Language Models (LLMs). Our approach offers two key advantages: simplifying user interaction with the system using natural language, and enabling the system to autonomously adapt to new API features. Since fulfilling a user's intent may involve multiple low-level API calls, our solution is designed to plan and execute them in a coordinated manner. We employ multi-agent LLMs with a hierarchical planning mechanism, creating a chatbot-like system that processes natural language inputs effectively. Real-world experiments conducted at EURECOM's OSS demonstrated that the proposed approach can efficiently manage all 6G domains using natural language.**

***Index Terms*—6G, Operations support systems, Intent-based networking, Large language models.**

I. Introduction

The era of 6G is expected to support a wide range of advanced services, such as eXtended Reality (XR) and Virtual Reality (VR), which will enhance human lifestyles by enabling critical use cases like teleoperated medical surgery and autonomous transportation [1]. These applications will impose stringent Quality of Service (QoS) and Quality of Experience (QoE) requirements that traditional networking systems are unable to meet. To address this, existing infrastructures will not only be upgraded with additional functionalities like Reconfigurable Intelligent Surfaces (RIS) and Cell-Free networks, but methods for sharing and virtualizing these infrastructure components will also become more advanced and efficient [2]. Consequently, the 6G network management layer, enabled by the Operations Support System (OSS) and responsible for managing and orchestrating this infrastructure, must be highly advanced and AI-powered, as emphasized by standardization institutions such as ETSI and 3GPP [3, 4].

The next-generation OSS will have diverse and heterogeneous responsibilities. Some of the key functionalities are: (i) managing the lifecycle of end-to-end (E2E) 6G services (e.g., creation, activation, deletion), which may span one or more 6G domains, such as the Radio Access Network (RAN), Transport Network (TN), Core Network (CN), and Edge/Cloud; (ii) managing infrastructure resources that are shared across different services and with other OSSs (e.g., those belonging to different network operators); and (iii) enabling observability and monitoring, as well as supporting Zero-touch network and Service Management (ZSM), which involves detecting and resolving anomalies without human intervention in services or infrastructure components [5]. These requirements have led standardization institutions to propose various Application Programming Interfaces (APIs) to facilitate interactions between users and OSSs, using simplified structures to declare user intention (intent), a concept known as Intent-Based Networking (IBN) [6].

The IBN concept was introduced as a simplified way to interact with the OSS by defining the user's intent in a declarative manner, using standardized APIs. However, a key drawback of IBN APIs is that their declarative structures often rely on structured formats, such as YAML or JSON, which may not be user-friendly for those with limited domain knowledge. Furthermore, there are many heterogeneous IBN APIs, each designed to address a specific functionality within the OSS (e.g., TMF APIs) requiring users to learn and manage a variety of APIs, which is time-consuming and inefficient. These challenges have pushed the research community to explore next-generation IBN systems that support natural language interaction, allowing users to define their intent in everyday language without imposing a predefined structure [7, 8, 9]. This can be enabled by Large Language Models (LLMs), which excel at understanding human language.

However, existing research often focuses on specific functionalities, which can be inefficient, such as addressing only the configuration aspect of 6G services [7], typically involving the generation of the body for a single API call. A key challenge in research is enabling LLM-based systems to autonomously execute multiple API calls, create their request bodies (i.e., payload) and parameters, and adapt to new functionalities and APIs. Our approach, OSS-GPT, addresses this challenge by planning a sequence of API calls with their bod-

ies and parameters and executing them to fulfill a single high-level intent. It adapts to new functionalities and API endpoints by requiring updates only to the OSS API specifications [10]. To achieve this, we employ multi-agent LLMs that collaborate to perform hierarchical planning and execution of OSS API calls. The system relies solely on the API specifications as its knowledge base, ensuring automatic adaptation when new functionalities or endpoints are introduced.

The main contributions of this work are manifold:

- Designing a general-purpose IBN system that allows natural language interaction with the OSS.
- The IBN system uses multi-agent LLMs to satisfy high-level user intent. Together, these agents plan API calls, execute them, and report the results to the user, enabling a chatbot system. Among the agents, we propose a specialized agent trained solely for generating request bodies following specific standards, called blueprint generator.
- The system was implemented at EURECOM's OSS, adhering to ETSI standards for deploying and configuring 6G services using the Network Service Descriptor [11]. In this context, all agents are trained using In-Context Learning (ICL) at inference time, except for the NSD generator. The latter was trained using Low Rank adaptation (LoRa) [12], resulting in an LLM specialized in generating NSDs.
- Evaluation was performed in real-world conditions using the EURECOM's OSS [13], demonstrating that all these API calls can be replaced by a single chatbot, enabling natural language IBN.

The rest of the paper is organized as follows: Section II describes background on LLMs, IBN and the research gap. Section III introduces the design of the OSS-GPT. Our OSS-GPT at EURECOM is presented in Section IV and evaluated in Section V. Finally, we conclude the paper in Section VI.

II. Background and Related Works

In this section, we present the related works and background on LLMs and IBN. We then identify research gaps relevant to our study.

A. Large language models

LLMs are advanced transformer-based AI systems designed to comprehend human language and generate coherent responses based on their training data. These models excel in various Natural Language Processing (NLP) tasks, such as question answering, code generation, and sentiment analysis [14]. However, their performance is typically tied to the tasks covered by their training data, and they may underperform on new, unseen tasks. Adapting LLMs to novel tasks has become a significant area of research. Supervised fine-tuning is a common approach for retraining LLMs on new datasets [14], but it is computationally intensive and resource-heavy. To mitigate these challenges, alternative methods like ICL and Parameter-Efficient Fine-Tuning (PEFT) [15] have gained attention for their efficiency. On the one hand, ICL allows the model to learn tasks by embedding training data directly within the context (input) during inference, without modifying the model's parameters. This technique facilitates the creation of agentic LLMs, where the agent's role is dynamically defined in the model's context [16]. PEFT, on the other hand, includes approaches like LoRA (Low-Rank Adaptation) [12], which freeze the LLM's original weights and introduce small, trainable parameter sets. These new parameters typically represent a fraction of the model's overall size, significantly reducing computational costs.

An interesting application of ICL is the use of agentic LLMs. With this approach, a single LLM can generate multiple profiles (agents) during each inference run, using profile descriptions embedded in the context. This enables the creation of multi-agent LLM systems where agents can specialize in different tasks and communicate with each other. This concept arises from the observation that LLMs perform exceptionally well on simple tasks, making a multi-agent system effective for handling complex tasks by delegating specific responsibilities to specialized agents, as demonstrated in [16]. Another notable application of ICL is in data augmentation. Researchers have leveraged powerful closed-source LLMs to generate synthetic datasets, which are subsequently used to train smaller, more efficient LLMs [17]. In this paper, we explore both use cases. First, we apply data augmentation to create specialized service blueprints using LoRA. Second, we implement agentic LLMs using ICL for intent planning and execution, demonstrating their utility in constructing robust, efficient multi-agent systems.

B. Intent-based networking

IBN is a transformative approach to network management, aimed at simplifying interactions with/within the OSS to enable autonomous networking [6]. It is being widely standardized by organizations like ETSI, TMF, and 3GPP [3, 18, 19]. In these standards, IBN consists of five main stages: Intent Profiling, where users interact with the system to express their intent; Intent Translation, where high-level intent is translated into low-level configurations; Intent Resolution, to resolve conflicts arising from multiple users' intents; Intent Activation, to activate and provision the necessary resources; and Intent Assurance, to ensure that the intent is satisfied throughout its lifecycle. In these standards, the intent is typically declared using YAML or JSON structures, which can be time-consuming for beginner users to learn.

To address these complexities, researchers emphasize the use of natural language for intent profiling, rather than predefined structures. For example, in [7], the authors tackle profiling with natural language and leverage LLMs to translate the intent into an NSD. This work initially focused on the Cloud/Edge domain but was later expanded to cover the RAN domain in [8]. However, the system is limited to configuring only a network service with the NSD. Similarly, Fuad et al. in [20] proposed an LLM-based approach for intent profiling and translation, but their work is limited to the TN domain where they generate BGP and firewall configurations from natural language. In addition, researchers in [21] also explored natural

language intent profiling using a different NLP technique, called Named-Entity Recognition (NER), for intent detection and translation applied specifically to the TN domain. However, their approach focuses on a single, clear intent definition and one translation step. A significant limitation of NER tools is their inability to handle different languages of intent or address issues like typing errors, unlike LLMs, which are more flexible in dealing with such complexities [8]. These studies highlight that no work has yet implemented a natural language-based approach for all the stages of IBN that addresses complex Intents across a wide range of use cases, while encompassing all 6G technological domains.

C. Research gap

Our goal is to enable natural language-based IBN using multi-agent LLMs. This system translates high-level natural language intents into a sequence of OSS API calls, executing them sequentially to fulfill the user's original request. However, existing approaches, such as offline introspective plan-then-execute methods [22] or the ReAct framework [23], face challenges in adapting to API feedback and generating viable plans. RestGPT [24], attempts to address these issues by employing a more dynamic planning process. It adopts an iterative, coarse-to-fine online planning mechanism, i.e., when given a complex instruction, an LLM first generates high-level natural language sub-tasks, which are then mapped by another LLM to more granular API call plans. While RestGPT excels at handling complex tasks for simpler API definitions, it faces two primary challenges: (i) when applied to 6G OSS with long, standardized request bodies (service blueprints), this method struggles with generating these complex request structures, and (ii) it relies heavily on few-shot learning for each LLM agent, which is tightly coupled to the API specifications. As a result, any changes in the API specifications require generating new few-shot examples, making the learning process inefficient.

In OSS-GPT, we opt for an online planning and execution approach, similar to RestGPT, but address its challenges in two ways. First, to improve the efficiency of generating standardized service blueprints, we introduce a specialized LLM trained using the LoRa technique. This LLM is specifically designed to generate the complex, standardized service blueprints required for 6G OSSs. Second, to overcome the inefficiency of few-shot learning, we designed OSS-GPT to learn directly from the API specifications, without the need for additional few-shot examples. Unlike RestGPT, which employs a coarse-to-fine approach to translate high-level intents into multiple natural language intents using few-shot examples, OSS-GPT, trained solely on API specifications, directly selects and executes individual API calls, one by one. This approach enables autonomous updates within the OSS, driven solely by changes in the API specifications. In summary, we present a fully autonomous, user-friendly, intent-driven OSS for next-generation 6G network management that offers enhanced efficiency by eliminating the need for manual learning updates.

III. OSS-GPT: System design

In this section, we will define the problem related to planning and executing API calls, followed by the presentation of our solution design, OSS-GPT.

A. Problem Definition

Let the natural language intent be represented as a textual input I. The objective is to map I to an output, a natural language response R, which summarizes the results of executing $\mathcal{C}$. The response R provides an intent-driven summary of I, where $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ is a sequence of API calls. Executing these calls sequentially ensures that the intent I is fulfilled. Each API call C_i is defined by an endpoint $E_i \in \mathcal{E}$, where $\mathcal{E} = \{E_1, E_2, \ldots, E_m\}$ is the set of available API endpoints. Each endpoint E_i is characterized by:

$$E_i = \langle m, p, i_s, o_s \rangle,$$

where m is the HTTP method (e.g., `GET`, `POST`, `PUT`), p is the endpoint URI structure, i_s is the expected input parameters (e.g., body, headers, query), and o_s is the structure of the output or response.

The problem is defined as finding a mapping $f : I \to R$, where:

$$f = (f_p, f_e, f_s),$$

with:

- $f_p : I \to \mathcal{C}$: generates the plan (i.e., the sequence of API calls) by selecting and ordering API endpoints $E_i \in \mathcal{E}$.
- $f_e : \mathcal{C} \to \mathcal{D}$: executes the planned API calls $\mathcal{C}$, producing a set of execution results $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, where D_i is the output of C_i.
- $f_s : \mathcal{D} \to R$: synthesizes the execution results $\mathcal{D}$ into a natural language response R, summarizing the outcome of the API sequence.

The planning step must consider:

- Logical dependencies between API calls (e.g., when one API's output is required as input for another).
- Constraints defined by the input and output schemas of the APIs.
- The overall intent I, ensuring that $\mathcal{C}$ addresses all necessary sub-tasks implied by I.

The execution step must:

- Generate or retrieve the input required for each API call C_i based on i_s.
- Monitor the responses to ensure they conform to the expected o_s.
- Handle any errors or replan f_p dynamically as needed.

The summarization step must:

- Extract key information from $\mathcal{D}$ based on the outputs o_s of the APIs.
- Generate a concise and coherent natural language response R that aligns with intent I.
- Handle ambiguities or incomplete information in $\mathcal{D}$ using contextual cues from I.

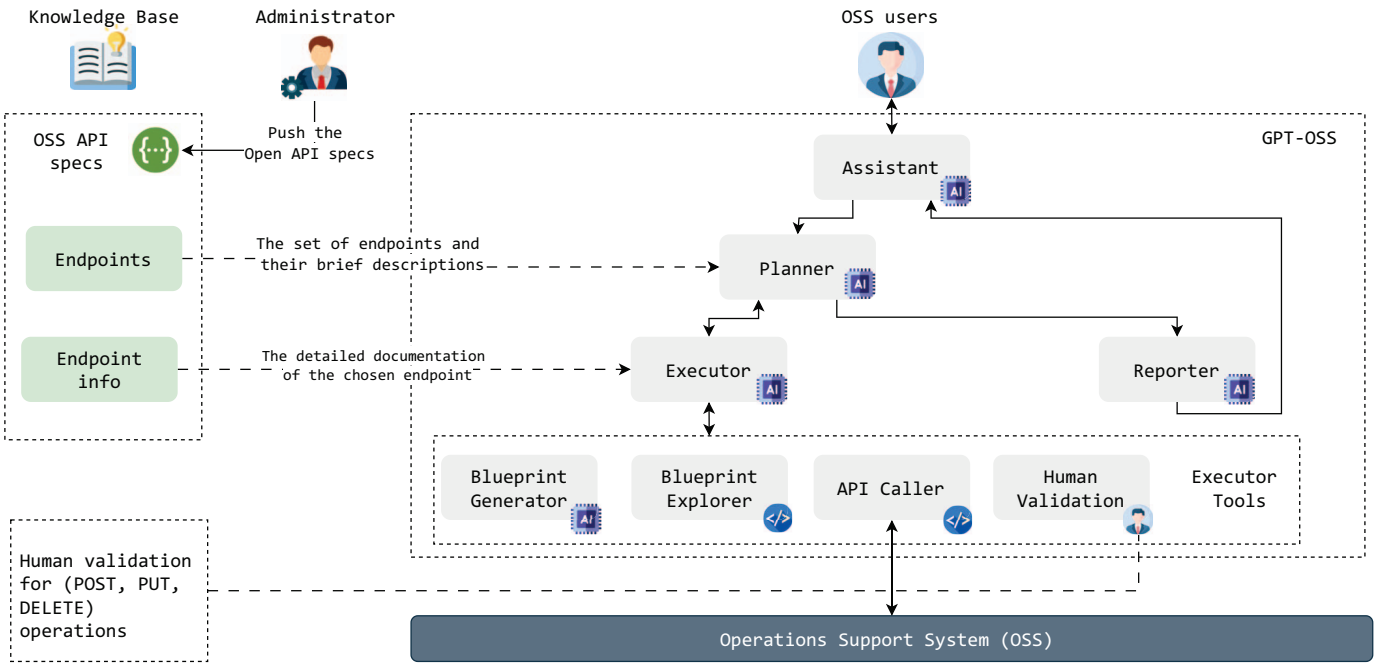

Fig. 1: OSS-GPT design.

B. Solution design

To tackle the aforementioned problem, we propose a multi-agent LLM framework, called OSS-GPT. Our solution involves multiple LLMs that work collaboratively to fulfill the intent I and provide the natural language response R. These LLMs function as a chatbot, providing OSS users with the results of their intents. As shown in Fig. 1, OSS-GPT is composed of the following components:

1) Assistant: The role of the *Assistant* is to maintain a chatbot-like interaction with the user. It accepts natural language input I and responds in natural language output R. This agent is trained using ICL, which handles two cases: (i) If the user asks general questions not related to OSS functions, the *Assistant* will respond directly to the user's query; (ii) If the user requests intents to be executed by the OSS, the *Assistant* forwards the intent to the *Planner* and waits for the *Reporter*'s response to provide it to the user. This design allows for seamless natural language interaction between the OSS users and the system.

2) Planner: The *Planner*'s primary responsibility is to generate a sequence of API calls $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ that will fulfill the intent of the user I. Upon receiving the intent from the *Assistant*, the *Planner* analyzes the request and determines the appropriate set of API calls needed to achieve the desired outcome. To accomplish this, the *Planner* is trained using ICL, where it leverages a knowledge base containing brief descriptions of available API endpoints. Due to context size limitations, the full specification of each endpoint $E_i \in \mathcal{E}$ cannot be included in the context; instead, the *Planner* only has access to these concise descriptions, which provide enough information to select relevant API calls without overloading the context. The *Planner* operates by selecting and sending one API call at a time to the *Executor*. Once the *Executor* has executed the API call:

- If the previous execution was successful, the *Planner* proceeds to generate the next API call in the sequence.
- If an error or failure occurs during execution, the *Planner* will dynamically replan, adjusting the sequence of API calls as needed to fulfill the intent I.
- If no further planning is required, meaning I has been fulfilled, the *Planner* requests the *Reporter* to generate the natural language response R.

This dynamic replanning mechanism ensures that the system remains flexible and robust, even when some executions fail or need to be modified in response to changing conditions.

3) Executor: The primary responsibility of the *Executor* is to execute each API call selected by the *Planner* and return the execution results. The *Executor* is also trained using ICL, which provides detailed descriptions of the available endpoints E_i to guide the execution process. Upon receiving the API call instructions from the *Planner*, the *Executor* not only executes the request but also plan the necessary tools required for the successful execution of the API call. Although additional tools may be employed, the Executor typically relies on the following three core tools:

- *Blueprint Generator*: Used to generate the necessary blueprints for POST requests, such as standardized structures (e.g., Network Service Descriptions, NSD). This tool is an LLM specifically trained to generate request bodies from natural language inputs.
- *Blueprint Explorer*: Utilized for retrieving existing information for PUT requests (e.g., when updating or

modifying data), ensuring relevant context is included in the execution. This is a straightforward program that performs GET requests to the OSS to fetch the required data.

- *API Caller*: Responsible for executing API endpoints by making appropriate HTTP requests, including all necessary parameters (headers, body, query parameters, etc.). This is a straightforward program that executes all API calls via the OSS.
- *Human Validation*: Ensures user approval is obtained before executing critical API operations, such as POST requests for creating services, PUT requests for modifications, and DELETE requests. This tool is requested before the *API Caller* is invoked for these operations. Users can activate or deactivate this tool based on their preference for system autonomy, enabling either human oversight or a fully autonomous intent-driven system.

For example, when the *Executor* receives a request to create a 6G service, such as a 6G CN, it first invokes the *Blueprint Generator* to produce the necessary blueprints for the POST request. Afterward, the *Executor* uses the *API Caller* to send the request to the appropriate API endpoint E_i, based on the specified HTTP method m, and waits for the response. Upon receiving the response, the *Executor* performs the following tasks:

- It verifies that the response conforms to the expected output structure o_s.
- If the response is successful and matches the expected schema, the *Executor* returns the result D_i to the Planner for further processing.
- If the response indicates an error, or if there is a mismatch with o_s, the *Executor* notifies the *Planner*. In such cases, the *Planner* may initiate a replan.

In addition to these tasks, the *Executor* is responsible for providing debugging information, which supports dynamic replanning by the *Planner*. This ensures that the system remains resilient and capable of adapting to unexpected issues that may arise during the execution of the API calls.

4) Reporter: The *Reporter* agent is an LLM that takes the conversation history between the *Planner* and *Executor*, along with the user's intent I, as input, and generates a natural language report R as output. It is trained using ICL and sends the response to the *Assistant* agent.

IV. EURECOM's OSS-GPT

In this section, we first present the design of EURECOM's OSS and then proceed with the instantiation of OSS-GPT within EURECOM's OSS framework, focusing specifically on the *Blueprint Generator* tool, which is specific to the OSS. This contrasts with the other agents described in the previous section, which are general-purpose.

A. EURECOM's OSS

The design of EURECOM's OSS is inspired by the 3GPP MANO architecture standards, adhering to the principles of the Service-Based Management Architecture (SBMA) [25]. The OSS is structured into multiple microservices, each dedicated to a specific role: (i) Service Manager, responsible for deploying and configuring 6G services, using the NSD from ETSI to define service configurations; (ii) Infrastructure Manager, tasked with provisioning virtualized and physical infrastructure; (iii) Monitoring Manager, deploying components responsible for KPI collection and monitoring of services and infrastructure [26]; and (iv) Fault Manager, which detects and resolves faults within services and infrastructure, leveraging the KPIs collected by the monitoring manager. These microservices communicate with each other via API calls, coordinated through a centralized API gateway that serves as the entry point and houses the complete API specifications. Without OSS-GPT, users interact with the API gateway through HTTP requests, which are routed to the appropriate microservice based on the type of request (e.g., service creation, infrastructure provisioning, KPI collection, or ZSM activation). Figure 2 illustrates the high-level design of EURECOM's OSS.

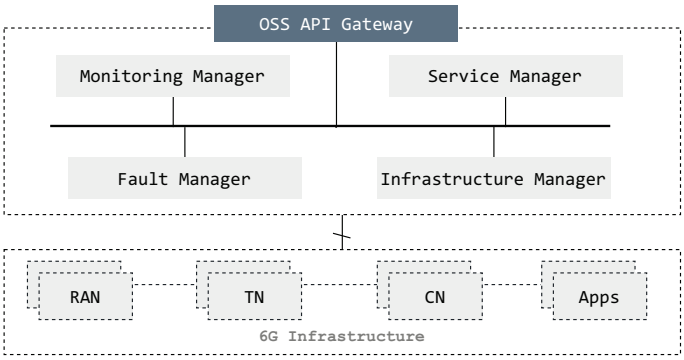

Fig. 2: EURECOM's OSS components.

To integrate OSS-GPT within EURECOM's OSS, training all the AI components (LLM agents) shown in Fig. 1 is essential. For all AI components except the *Blueprint Generator*, training with EURECOM's OSS API specifications is sufficient, as these components rely on ICL using the specifications during inference, thus only pushing the API specs into the knowledge base is required. However, the *Blueprint Generator* has a distinct role: it must generate accurate NSDs directly from natural language inputs. To achieve this, we developed a specialized LLM specifically trained for NSD generation. We opted for fine-tuning in this case because, in our previous work [7], this task was explored using ICL with a 34B LLM. While ICL demonstrated high accuracy, the complexity of the task resulted in significant generation times (over one minute for three applications). This delay created a bottleneck, particularly in multi-agent LLM systems where other agents must wait for completion. Fine-tuning enabled us to create a smaller, specialized LLM that eliminates the need for extensive context learning, processing only the natural language intent to generate accurate NSDs efficiently.

B. Blueprint Generator - The NSD-expert

To train the *NSD-expert*, two main steps are crucial: (i) dataset generation; and (ii) LLM training. Dataset generation

posed a significant challenge because there is no open dataset available containing natural language intents paired with their corresponding NSDs. To address this, we developed a dataset from scratch, containing examples that map natural language inputs to their respective NSD representations. Another challenge was adapting existing open-source LLMs through fine-tuning without compromising their general abilities to generate coherent text and respond to human queries. The goal was to create an LLM capable of generating NSDs when required, while still understanding and responding to natural language in a chatbot-like manner. This capability is essential for use cases where the user might need to modify a generated NSD via natural language interactions. To achieve this, a method was needed to inject the specialized knowledge of NSD generation into an existing LLM without erasing or degrading its prior knowledge. Next, we describe these two steps.

1) Dataset collection & augmentation: In the experiments described in [7], we requested volunteers to test an ICL-based NSD generator LLM. Volunteers provided scores for the LLM's output and corrected any mistakes in the generated NSDs. This process resulted in a small, high-quality dataset containing 100 entries of natural language intents paired with their corresponding NSDs. To augment this dataset, we leveraged the ICL capabilities of LLMs. Specifically, we used few-shot learning by providing 3-4 examples from the existing dataset in the LLM's context, accompanied by an instruction to generate a new example. For instance, let

$$\mathcal{D}_{\text{existing}} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

represent the original dataset, where x_i denotes the natural language intent and y_i is the corresponding NSD. Given a subset $\mathcal{D}_{\text{few-shot}} \subseteq \mathcal{D}_{\text{existing}}$, the LLM generates a new example (x_{n+1}, y_{n+1}) by conditioning on the context:

$$P(y_{n+1}|x_{n+1}, \mathcal{D}_{\text{few-shot}})$$

The newly generated examples were reviewed for quality and subsequently added to the knowledge base, effectively expanding the dataset iteratively. This method ensured that the augmented dataset retained high relevance and correctness.

2) Training: For training, we employed LoRa [12], a PEFT technique designed to reduce computational complexity while achieving effective adaptation of LLMs to specific tasks. As shown in Fig. 3, LoRA adds trainable low-rank matrices to the pre-trained weights of an existing LLM, freezing most of the original weights and significantly reducing the number of parameters that need adjustment.

Let $W \in \mathbb{R}^{d\times k}$ denote a weight matrix in the pre-trained LLM. In LoRA, this matrix is approximated as:

$$W \approx W_0 + \Delta W$$

where W_0 is the frozen pre-trained weight, and

$$\Delta W = AB, \quad A \in \mathbb{R}^{d\times r}, \quad B \in \mathbb{R}^{r\times k}, \quad r \ll \min(d, k)$$

ensures the rank of the adaptation matrices A and B is low, reducing the number of trainable parameters from $d \times k$ to $r \times (d + k)$. The process of fine-tuning can thus be expressed as minimizing the loss $\mathcal{L}$ over the task-specific data:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

where f_θ is the LLM with LoRA applied, and $\ell(\cdot)$ is the task-specific loss function. The adaptation is efficient because only ΔW is updated during training, while W_0 remains static. LoRa enabled us to specialize the LLM for NSD generation while retaining its original language understanding capabilities.

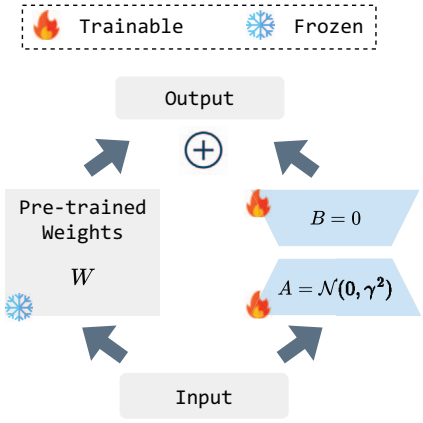

Fig. 3: LoRa [12].

V. Performance Evaluation

The section is structured into three subsections: *Experiment Setup*, which describes the experimental configuration; *Experiment Results*, which presents and analyzes the performance of the OSS-GPT approach; and *Experiment Conclusion*, which provides additional insights from the experiment, discusses challenges, and outlines potential future work.

A. Experiment setup

Our experimental setup, illustrated in Fig. 4, involves two machines hosting the Kubernetes-based cluster and the OSS-GPT framework, respectively. The first machine, equipped with an Intel(R) Xeon(R) Silver 4314 CPU (2.40GHz) and running Ubuntu 22.04.3 LTS, manages a single-node Kubernetes cluster that hosts the virtualized 6G services and infrastructure components. At EURECOM, we utilize OpenAirInterface components to create an end-to-end 6G service, encompassing the RAN (oai-nr-ue and oai-gnb), CN (oai-amf, oai-smf, oai-upf, etc.), and edge applications. The second machine, a MacBook with an Apple M1 Pro chip featuring an integrated GPU, hosts the infrastructure management components, including OSS components and OSS-GPT, running on Docker and bare-metal, respectively. The OSS-GPT is implemented using LangGraph[1], and interacts with two LLMs: OpenAI's GPT-4[2], accessed remotely, and *NSD-expert*, a locally deployed model with 8 billion parameters running on the Ollama framework[3].

[1]https://www.langchain.com/langgraph
[2]gpt-4o-2024-08-06
[3]https://ollama.com

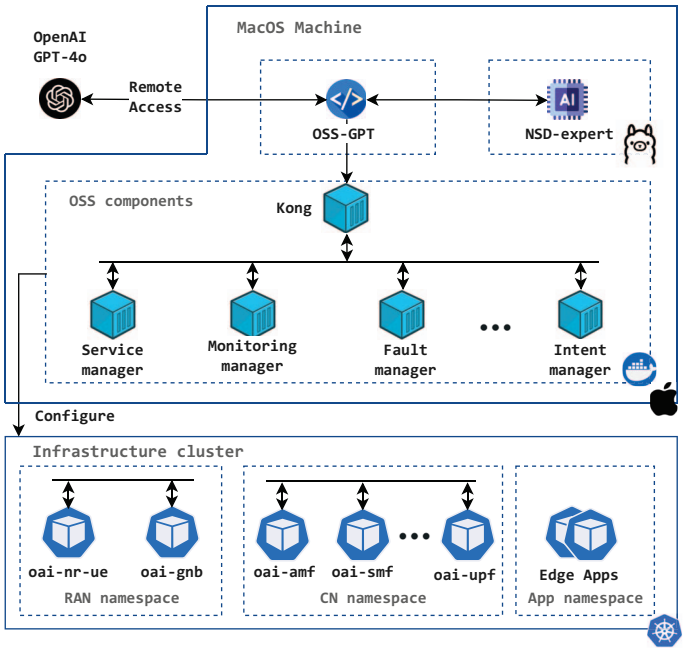

Fig. 4: Experimentation setup.

The *NSD-expert* model was fine-tuned on a third machine equipped with an NVIDIA A100 GPU with 40GB of vRAM, using Llama 3.2 (3B parameters)[4] as the base model and applying the LoRA technique. For training, we employed the Unsloth framework[5] with a training setup that included 120 steps, a learning rate of 2e-4, and a linear learning rate scheduler. The model was trained using a per-device batch size of 2 with gradient accumulation over 4 steps to simulate a larger effective batch size. Mixed-precision training was enabled, dynamically selecting FP16 or BF16 based on hardware compatibility, and the AdamW optimizer was used with an 8-bit implementation to reduce memory usage.

B. Experiment results

In this subsection, we first evaluate the quality of the *NSD-expert*. Next, we assess the overall performance of the framework, OSS-GPT. Finally, we analyze the cost-effectiveness of OSS-GPT in realistic scenarios, focusing on training costs, inference execution time, and pricing.

1) NSD-expert evaluation: Fig.5 shows the loss function of fine-tuning the Llama3 LLM on the generated dataset. This demonstrates that the LLM's loss function was decreasing, indicating effective training, and it converged after 80 steps. The resulting *NSD-expert* is evaluated in Fig.6 against the previous Llama3 version using four metrics: (i) Perplexity, which measures how well the model predicts the next token in a sequence; (ii) BERTScore, which evaluates the semantic similarity between generated and reference text using contextual embeddings; (iii) Cosine similarity, which measures the cosine of the angle between the vector representations of the generated and reference texts; and (iv) Exact Match (EM), which assesses the percentage of exact matches between the generated and reference outputs.

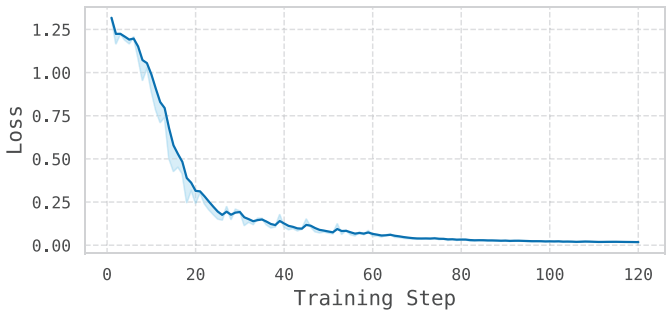

Fig. 5: Training loss.

As shown in Fig.6, the perplexity metric of *NSD-expert* is very low, indicating that the model was confident in generating the JSON files. However, the perplexity of the old version of Llama3 is smaller than that of *NSD-expert*, suggesting that the Llama3 distribution makes more sense for text generation. This is because perplexity is an appropriate metric for evaluating text generation, but it does not effectively assess structured outputs like JSON. To evaluate the quality of the JSON generation, we used Cosine similarity, BERTScore, and Exact Match (EM). The BERTScore and Cosine similarity values are low for Llama3, indicating that this model struggled to generate the JSON structure correctly. In fact, it did not even recognize that the NSD output is a JSON structure. This is further evidenced by the 0 score for Llama3 in the EM metric. In contrast, *NSD-expert* generated highly accurate JSON files, closely matching the reference JSON file, achieving approximately 0.99 for BERTScore and Cosine similarity, and a 60% EM, which is a very strong score. This demonstrates that the *NSD-expert* LLM can be trusted by the OSS-GPT system. Next, we evaluate the whole OSS-GPT system, using also the *NSD-expert*.

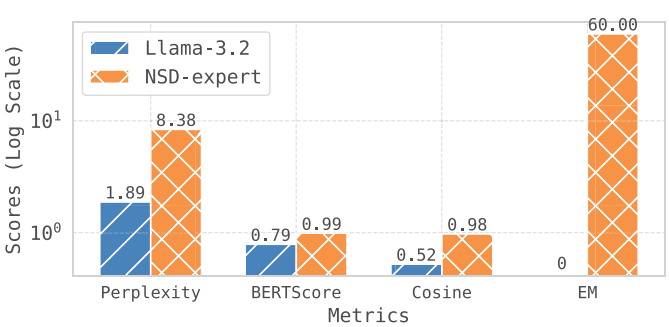

Fig. 6: Metrics.

2) OSS-GPT evaluation: As mentioned earlier, the OSS-GPT functions as a chatbot from the OSS user's perspective, enabling interactions to obtain information about services and infrastructure, create or delete them, and more, using high-level intents. An example of this interaction is illustrated in Fig. 7, where OSS-GPT generates a plan consisting of a set

[4]https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct
[5]https://unsloth.ai

of API calls, including parameters and body requests for each API call, and executes them sequentially in the background. The minimum number of API calls required to satisfy an intent is referred to as the golden path. For example, in Fig. 7, two questions are presented. In the first question, the golden path consists of one API call, a single GET request to fetch the list of services. In the second question, the golden path consists of three API calls: a GET request to retrieve the list of services, a GET request to obtain information about a specific service, and a POST request to create a subservice with gNB configuration (generated by the blueprint generator).

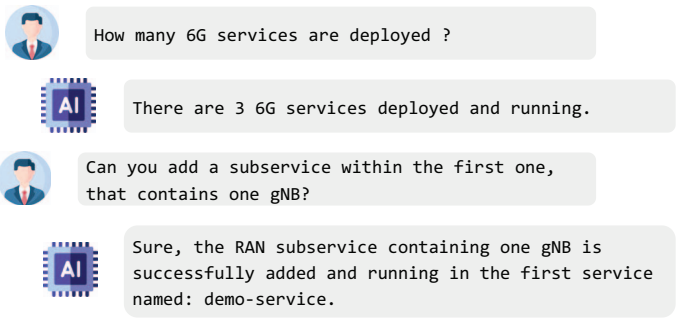

Fig. 7: Chatbot example when trust is activated.

To evaluate the quality of OSS-GPT, we prepared a dataset of intents generated by domain experts, ranging from simple to complex ones requiring multiple API calls (golden paths). For this purpose, we defined 8 levels of intents, corresponding to 8 golden paths (n). A golden path of i indicates that OSS-GPT needs to make at least i API calls (where $i = 0$ means that OSS-GPT responds entirely on its own without making any requests to the OSS). Fig. 8 illustrates two key metrics: (i) the number of API calls executed to fulfill an intent (Path Cost) compared to the golden path (orange curve); and (ii) the accuracy of OSS-GPT in successfully fulfilling intents for each golden path (blue). It can be observed that for intents requiring fewer than 4 API calls, OSS-GPT achieved an accuracy close to 1.0, successfully fulfilling all intents, which represents a perfect score. However, for intents with golden paths greater than 4, the accuracy gradually decreases as the complexity of the intents increases and more API calls are required. Notably, for golden path 7, OSS-GPT maintained an accuracy of 0.80, which is a strong performance given the complexity of the task. This demonstrates the capability of LLMs in planning and executing API calls effectively. On the other hand, as the golden path increases, OSS-GPT tends to make more API calls than necessary. This indicates occasional errors in the planning process. Nevertheless, thanks to its replanning mechanism, most intents are eventually fulfilled successfully.

3) Cost assessment: In this part, we evaluate the costs associated with training and inferring OSS-GPT. Our approach involves creating a custom LLM, designated as the blueprint generator (i.e., *NSD-expert*), while leveraging GPT-4 for other components, as it is currently the leading reasoning model. Thus, the costs can be divided into two categories: training and inference: (i) The training of the *NSD-expert* required

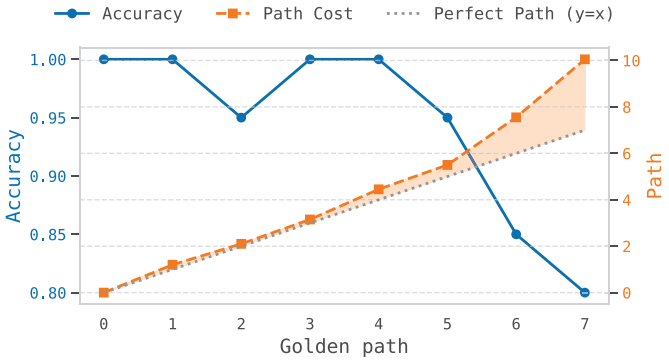

Fig. 8: Accuracy vs golden path.

only 3.07 minutes, with a reserved memory of 4.006 GB (of which 1.371 GB was allocated for LoRA). This training process is highly efficient and inexpensive, as the primary task of this LLM is generating NSDs. The domain-specific knowledge was injected rapidly, minimizing resource consumption; (ii) The inference cost is measured using two metrics: the generation time required for OSS-GPT to respond to queries and OpenAI's pricing, as GPT-4 is not an open-source LLM. These metrics were evaluated per golden path. As shown in Fig. 9, the execution time increases with the golden path. Remarkably, for golden paths up to 7, the execution time remained under one minute, demonstrating very fast planning and execution. Additionally, the pricing increases with the golden path due to multiple LLM requests. However, even for complex requests requiring seven API calls, the cost of $0.175 per request is affordable, especially for large-scale industrial OSS deployments. It is worth noting that OpenAI's pricing is continuously decreasing due to the competition with other closed-source and open-source models. Furthermore, advancements in open-source models, particularly in reasoning capabilities, are narrowing the gap between open- and closed-source solutions. This progression makes open-source models a viable alternative for systems like OSS-GPT in the near future.

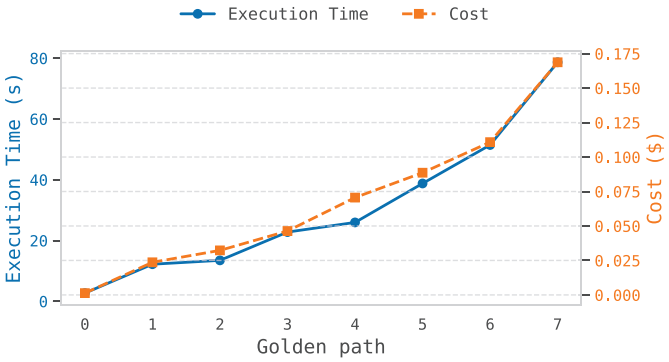

Fig. 9: OSS-GPT cost assessment.

C. Experiment conclusion

From the results, we can conclude that implementing natural language-based IBN is a challenging but highly useful approach for enhancing user experience and eliminating the complexities associated with traditional standardized structures. Indeed, OSS-GPT demonstrates remarkable performance in simplifying the user experience for OSS users and reducing time spent on learning standards and adapting to new APIs. However, as the experimental setup revealed, EURECOM's implementation of OSS-GPT relies heavily on closed-source LLMs, such as GPT-4, which are less secure compared to local private LLMs. This reliance arises because existing open-source LLMs struggle to consistently generate structured outputs. This capability is critical for OSS-GPT, as each LLM must adhere to a predefined output structure to communicate effectively with other LLM agents. Current advancements in open-source LLMs face challenges in consistently generating accurate structures (structured output), which is crucial in multi-agent LLM systems. In such systems, other agents rely on the LLM's output to proceed with their tasks. This represents a critical area where open-source LLMs lag behind closed-source LLMs, such as OpenAI's GPT. Open-source LLM frameworks should focus on enhancing the accuracy of structured outputs to a 99% level. Achieving this would make multi-agent LLMs based on open-source LLMs a feasible solution. Another important consideration is that OSS-GPT, as presented, is not equipped to handle multiple OSSs. This capability is essential for the future of 6G networking, where tenants or service providers can send intent to multiple OSSs corresponding to different Mobile Network Operators (MNOs). Our future work will focus on addressing this limitation by developing a chatbot that implements natural language-based, intent-driven networking across multiple OSSs that serve multiple MNOs.

VI. Conclusion

This paper presents a novel approach to managing 6G networks by leveraging natural language-based, intent-driven management powered by multi-agent LLMs. By enabling natural language interactions, the proposed system simplifies user engagement with complex OSS platforms and autonomously adapts to evolving API functionalities. This is achieved through advancements in LLMs, which enable the understanding of natural language intents and the collaborative, hierarchical planning and execution of API calls. Experimental results at EURECOM demonstrate the effectiveness of this approach in automating 6G network management. Future work should focus on extending OSS-GPT to support multiple OSSs corresponding to different MNOs and optimizing inference costs by adopting open-source LLMs instead of closed-source alternatives.

Acknowledgment

This work is supported by the European Union's Horizon Program under the 6G-Sunrise project (Grant No. 101139257).

References

[1] Wei Jiang et al. "The road towards 6G: A comprehensive survey". In: *IEEE Open Journal of the Communications Society* 2 (2021).

[2] Abdelkader Mekrache, Adlen Ksentini, and Christos Verikoukis. "Machine Reasoning in FCAPS: Towards Enhanced Beyond 5G Network Management". In: *IEEE Communications Surveys & Tutorials* (2024).

[3] ETSI. *Zero Touch Network and Service Management (ZSM) Means of Automation*. 2018.

[4] 3rd Generation Partnership Project (3GPP). *Study on Artificial Intelligence (AI) and Machine Learning (ML) for NR Mobility*. Tech. rep. TR 38.744. Release 19. 3GPP, 2024.

[5] Abdelkader Mekrache et al. "On Combining XAI and LLMs for Trustworthy Zero-Touch Network and Service Management in 6G". In: *IEEE Communications Magazine* (2024).

[6] Aris Leivadeas and Matthias Falkner. "A survey on intent-based networking". In: *IEEE Communications Surveys & Tutorials* 25.1 (2022), pp. 625–655.

[7] Abdelkader Mekrache and Adlen Ksentini. "LLM-enabled intent-driven service configuration for next generation networks". In: *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. IEEE. 2024, pp. 253–257.

[8] Abdelkader Mekrache, Adlen Ksentini, and Christos Verikoukis. "Intent-based management of next-generation networks: An LLM-centric approach". In: *Ieee Network* (2024).

[9] Akram Boutouchent et al. "6G-INTENSE: Intent-Driven Native Artificial Intelligence Architecture Supporting Network-Compute Abstraction and Sensing at the Deep Edge". In: *IEEE Vehicular Technology Magazine* (2025).

[10] OpenAPI Initiative. *OpenAPI Specification Version 3.1.0*. https://github.com/OAI/OpenAPI-Specification. Accessed: 2024-12-03. 2021.

[11] Sagar Arora, Adlen Ksentini, and Christian Bonnet. "Lightweight edge slice orchestration framework". In: *ICC 2022-IEEE International Conference on Communications*. IEEE. 2022, pp. 865–870.

[12] Edward J Hu et al. "Lora: Low-rank adaptation of large language models". In: *arXiv preprint arXiv:2106.09685* (2021).

[13] Sagar Arora et al. "A 5G Facility for Trialing and Testing Vertical Services and Applications". In: *IEEE Internet of Things Magazine* 5.4 (2022), pp. 150–155.

[14] Wayne Xin Zhao et al. "A survey of large language models". In: *arXiv preprint arXiv:2303.18223* (2023).

[15] Ning Ding et al. "Parameter-efficient fine-tuning of large-scale pre-trained language models". In: *Nature Machine Intelligence* 5.3 (2023), pp. 220–235.

[16] Junyou Li et al. "More agents is all you need". In: *arXiv preprint arXiv:2402.05120* (2024).

[17] Bosheng Ding et al. "Data augmentation using llms: Data perspectives, learning paradigms and challenges". In: *arXiv preprint arXiv:2403.02990* (2024).

[18] *TM Forum TMF921A - Intent Management API*.

[19] 3GPP Technical Specification Group Services and System Aspects. *Study on scenarios for Intent driven management services for mobile networks, Telecommunication management*. Tech. rep. 2019.

[20] Ahlam Fuad et al. "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.

[21] Arthur S Jacobs et al. "Hey, lumi! using natural language for {intent-based} network management". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 625–639.

[22] Yongliang Shen et al. "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face". In: *Advances in Neural Information Processing Systems* 36 (2024).

[23] Shunyu Yao et al. "React: Synergizing reasoning and acting in language models". In: *arXiv preprint arXiv:2210.03629* (2022).

[24] Yifan Song et al. "RestGPT: Connecting Large Language Models with Real-World RESTful APIs". In: *arXiv preprint arXiv:2306.06624* (2023).

[25] *3GPP TS 28.533: Management and orchestration; Architecture framework (Release 17)*. Technical Specification TS 28.533. Release 17. 3rd Generation Partnership Project (3GPP), 2024.

[26] Mohamed Mekki, Sagar Arora, and Adlen Ksentini. "A scalable monitoring framework for network slicing in 5G and beyond mobile networks". In: *IEEE Transactions on Network and Service Management* 19.1 (2021), pp. 413–423.