

Received XX Month, XXXX; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version 11 January, 2024.

Digital Object Identifier 10.1109/OJCOMS.2024.011100

NetIntent: Leveraging Large Language Models for End-to-End Intent-Based SDN Automation

Md. Kamrul Hossain¹, Walid Aljoby¹

¹Information and Computer Science Department, King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia

CORRESPONDING AUTHOR: Walid Aljoby (e-mail: waleed.gobi@kfupm.edu.sa).

ABSTRACT Intent-Based Networking (IBN) often leverages the programmability of Software-Defined Networking (SDN) to simplify network management. However, significant challenges remain in automating the entire pipeline, from user-specified high-level intents to device-specific low-level configurations. Existing solutions often rely on rigid, rule-based translators and fixed APIs, limiting extensibility and adaptability. By contrast, recent advances in large language models (LLMs) offer a promising pathway that leverages natural language understanding and flexible reasoning. However, it is unclear to what extent LLMs can perform IBN tasks. To address this, we introduce **IBNBench**, a first-of-its-kind benchmarking suite comprising four novel datasets: Intent2Flow-ODL, Intent2Flow-ONOS, FlowConflict-ODL, and FlowConflict-ONOS. These datasets are specifically designed for evaluating LLMs performance in intent translation and conflict detection tasks within the industry-grade SDN controllers ODL and ONOS. Our results provide the first comprehensive comparison of 33 open-source LLMs on IBNBench and related datasets, revealing a wide range of performance outcomes. However, while these results demonstrate the potential of LLMs for isolated IBN tasks, integrating LLMs into a fully autonomous IBN pipeline remains unexplored. Thus, our second contribution is **NetIntent**, a unified and adaptable framework that leverages LLMs to automate the full IBN lifecycle, including translation, activation, and assurance within SDN systems. NetIntent orchestrates both LLM and non-LLM agents, supporting dynamic re-prompting and contextual feedback to robustly execute user-defined intents with minimal human intervention. Our implementation of NetIntent across both ODL and ONOS SDN controllers achieves a consistent and adaptive end-to-end IBN realization.

INDEX TERMS Intent-Based Networking, Software-Defined Network, Large Language Models

I. INTRODUCTION

Software-Defined Networking (SDN) was emerged as a revolutionary paradigm to address the challenges of traditional networks [1], [2]. SDN decouples the network's control functions from its data forwarding functions and enables centralized control and programmability, which are instrumental in simplifying and automating network operations. SDN has become a cornerstone for building agile, efficient networks that can adapt to future needs. Several SDN deployments, such as Google's B4 [3] and Microsoft's SWAN [4], demonstrate real-world scalability. However, SDN's promise of automation is not without limitations. Although it reduces the need for manual configuration at the device level, human intervention is still required to translate business goals or high-level policies into instructions for the SDN controller to realize them on the data plane. OpenDaylight

(ODL) [5] and Open Network Operating System (ONOS) [6] are two commonly used industry-grade SDN controllers. Organizations like AT&T and Orange extensively use both ODL and ONOS [7]–[9]. ODL uses DLUX and ONOS uses web GUI to enable users to configure policies through a user-friendly interface [10]. However, this process remains time-consuming and prone to human error, especially during initial setup and for complex network configurations.

Intent-Based Networking (IBN) emerges as a natural evolution of SDN, with the aim of simplifying network management by bridging the gap between high-level policy objectives and low-level network configurations [11]–[13]. Industry leaders such as Cisco and Nokia have been instrumental in developing and popularizing IBN in their data centers [14], [15]. IBN enables operators to define what the network should achieve using high-level, human-readable

intents, rather than specifying how the network should be configured. Formally, intent is defined as a set of operational goals that a network is supposed to meet and outcomes that a network is supposed to deliver, expressed declaratively without specifying how to implement them [13]. Intents such as prioritizing video traffic over other types of traffic in an ISP network, or ensuring latency below 5 ms with a minimum bandwidth allocation of 50 Mbps for the URLLC slice in a 5G core network, express desired outcomes without specifying implementation details. This declarative approach enables greater automation and dynamic optimization across different types of networks. Thus, IBN elevates SDN from being a merely programming platform to an autonomous goal-driven architecture.

While IBN significantly reduces the complexity of network management, current implementations still often require operators to express these intents in structured formats like NSD, JSON, XML, or YAML [12]. Translating natural-language intents into these machine-readable formats imposes a technical barrier, as it demands familiarity with underlying data models (e.g., YANG [16]) and their associated syntax and semantics. For instance, installing flow rules via an ODL controller requires the operator to craft a JSON message aligned with a specific YANG schema, which is an error-prone task that increases operational overhead and risk of misconfiguration.

This gap between high-level intent expression and low-level implementation limits the full potential of IBN. To truly realize the vision of intent-driven networking in which users can express goals in natural language and delegate implementation to the system, there is a need for intelligent mechanisms that can parse, interpret, and compile intents with minimal human intervention. To unlock the full potential of IBN, the network system must support three critical functions: intent translation, intent activation, and intent assurance, to ensure seamless validation and automation of user-defined intents. These functions, when described sequentially, are known as intent lifecycle [12].

Recent research efforts used large language models (LLMs) [17]–[19] to enhance network management through IBN. These studies mainly focused on intent translation and did not adequately address the challenges of conflicting intents as well as intent assurance. Further, autonomous end-to-end orchestration of intent lifecycle is still under-addressed. For example, previous work [20]–[26] did not systematically evaluate the performance of LLMs throughout the entire intent management lifecycle, particularly in critical aspects such as conflict detection and intent assurance. In addition, these works do not cater to SDN controllers like ODL and ONOS without a major modification to their proposed system. Moreover, existing research often relies on closed-source models like ChatGPT, which limit the applicability and transparency of the real world.

To systematically assess the capabilities of LLMs in automating end-to-end IBN tasks, we introduce *IBNBench* to

perform a comprehensive benchmarking study that includes intent translation and conflict detection. Existing efforts in this domain typically evaluate few LLMs and focus on limited datasets. In contrast, we benchmark 33 open-source LLMs spanning a diverse range of model sizes and architectures on six datasets, including two existing ones and four newly proposed benchmarks by us. These include the *Intent2Flow-ODL* and *Intent2Flow-ONOS* datasets, which, to the best of our knowledge, are the first to represent natural language intents paired with controller-specific flow rule configurations for both ODL and ONOS. In addition, our *FlowConflict-ODL* and *FlowConflict-ONOS* datasets provide the first curated examples of annotated flow rule pairs for benchmarking LLMs on conflict detection. These datasets enable structured and reproducible comparisons across models and serve as practical testbeds for real-world SDN scenarios. By releasing these datasets and benchmarking results, we provide the research community with essential tools and baselines to further advance LLM-driven IBN systems. To the best of our knowledge, no prior work has conducted such an extensive benchmarking effort across this combination of *tasks, models, and datasets*.

While IBNBench reveals the capabilities and limitations of current LLMs in handling core IBN tasks such as translation and conflict detection, the broader question of how LLMs can be harnessed for end-to-end IBN realization remains unexplored. Previous studies predominantly addressed intent translation, often overlooking critical challenges such as conflict detection, closed-loop assurance, and controller-specific adaptability. To address these gaps, we develop *NetIntent*, an end-to-end intent lifecycle architecture that encompasses intent translation, activation, and assurance, specifically targeting real-world applications for industry-grade SDN controllers, ODL, and ONOS.

Although modern networks encompass a vast and evolving landscape of thousands of operational intents and configuration tasks, in this paper, we focus specifically on three foundational categories of network intents: *Forwarding* (e.g., routing traffic to specific ports or interfaces), *Security* (e.g., blocking or dropping traffic), and *QoS (Quality of service)* (e.g., prioritizing specific traffic using queues). Our objective is to demonstrate how LLMs can be effectively integrated within an IBN system to enable intent translation, activation, and assurance in a structured and explainable manner. By focusing on a representative subset of fundamental intents, we provide a practical and reproducible system that serves as a proof-of-concept. This example can act as a template for future research aimed at extending LLM-based IBN systems to support broader and more intents in a complex operational environments.

In Fig. 1, we present a comparative overview of NetIntent and traditional IBN systems throughout the entire intent processing workflow. On the left side of Fig. 1, conventional IBN systems rely on structured or semi-structured user interfaces to capture operator intent. These systems typically

TABLE 1: Comparison of NetIntent with relevant works on IBN

Category	Aspect	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[21]	[22]	[23]	[24]	[25]	[20]	[26]	[34]	NetIntent
Methodology																	
LLM-based	–	–	–	–	–	–	–	–	✓	✓	✓	✓	✓	✓	✓	✓	✓
Non-LLM	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–
Implementation																	
Automated	✓	✓	–	–	–	–	✓	✓	✓	✓	–	–	✓	✓	✓	✓	✓
Input Method																	
Natural Language Input	✓	–	–	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IBN Lifecycle Functions																	
Intent Translation	✓	–	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Conflict Detection	✓	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–	✓
Activation/Deployment	✓	✓	–	–	✓	✓	✓	✓	✓	–	✓	✓	–	–	✓	✓	✓
Intent Assurance	–	✓	–	–	–	✓	–	✓	–	–	–	–	–	–	✓	✓	✓
LLM Application																	
# LLMs Evaluated	–	–	–	–	–	–	–	–	1	3	at least 7	8	7	3	at least 3	at least 1	at least 33
LLM Type	–	–	–	–	–	–	–	–	open-src	open-src	mixed	mixed	mixed	open-src	closed-src	open-src	

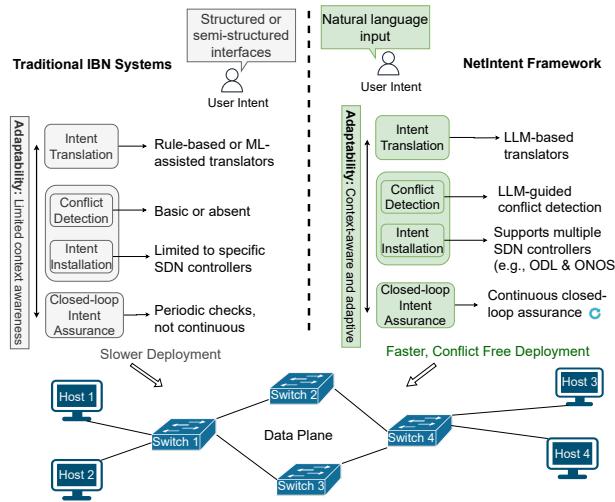


FIGURE 1: Illustration of NetIntent framework; comparison with prior IBN systems in terms of intent life-cycle.

execute a linear and static sequence that includes intent translation, conflict detection, flow rule installation, and optional closed-loop assurance. However, their adaptability is limited, as they often lack contextual reasoning and dynamic reactivity, resulting in slower deployments and increased likelihood of misconfigurations. In contrast, NetIntent, depicted on the right side of Fig. 1, enables users to express intents directly in natural language. It employs LLMs in a context-aware and adaptive manner which enables it to dynamically update LLM-prompts based on feedback and failed assurance checks. This design enables NetIntent to achieve faster and conflict-free intent deployment by proactively identifying issues, minimizing the need for manual intervention.

Table 1 presents a comparative overview of the most relevant work on IBN, categorized by key attributes of the system. The methodology category distinguishes between LLM-based approaches, where at least one LLM is em-

ployed at any stage of the system pipeline (e.g., intent parsing, translation, deployment) as seen in works [20]–[26], [34]—and Non-LLM systems [27]–[33] that rely on traditional programmatic logic or conventional machine learning. The Implementation category indicates whether the system is fully automated (no human input required beyond initial intent, except in rare exceptions), which applies to works like [20]–[22], [25], [27], [32], [33], or semi-automated (requiring human assistance for prompting or execution), as in [23], [24], [29]–[31]. Natural language input highlights whether the system can parse unstructured user intents, and IBN Lifecycle Functions include capabilities such as intent translation, conflict detection, deployment, and assurance. Finally, the LLM Application category reflects the extent and type of LLM usage, where applicable. While these works demonstrate early promise in automating various IBN tasks, our understanding of how open-source LLM-based systems can autonomously manage the full IBN lifecycle—from natural language input to assured deployment—remains limited.

In the following points, we summarize our contributions.

- We perform a benchmarking of 33 open-source LLMs using new and existing datasets comprising natural language intents and their corresponding JSON configurations. This evaluation assesses LLM performance in both intent translation and conflict detection, offering insights into their suitability for IBN tasks.
- We introduce IBNBench, a suit of four new datasets, *Intent2Flow-ODL*, *Intent2Flow-ONOS*, *FlowConflict-ODL*, and *FlowConflict-ONOS*, make them open source to support reproducible research. Intent2Flow datasets consist of natural language configuration intents paired with structured flow rules, while the FlowConflict datasets contain flow rule pairs annotated for conflict detection. These resources serve as practical benchmarks for evaluating LLM-driven IBN systems.
- We propose *NetIntent*, a novel end-to-end architecture for intent-based networking that leverages LLMs to

- automate the full intent lifecycle, including translation, activation, and assurance.
- To demonstrate the generalizability and practical applicability of our approach, we implement and evaluate NetIntent on two widely-used SDN controllers: ODL and ONOS.

II. BACKGROUND AND RELATED WORKS

The lifecycle of an intent in IBN encompasses distinct phases, namely Intent Translation, Intent Activation, and Intent Assurance. Each phase involves specific tasks and challenges essential to effectively operationalize high-level network objectives.

A. INTENT TRANSLATION

Intent translation is the process of converting user-defined high-level goals into precise, low-level network configurations that can be executed by controllers. This step is fundamental to IBN and its success determines how accurately the network enforces operator intentions. Key challenges in this phase include managing the inherent ambiguity of natural language, refining hierarchical or multi-layered intents, resolving multiple valid configurations, and supporting controller- or vendor-specific implementations. Traditional NLP techniques, while useful for rule-based intent parsing [27], [29]–[33], [35], often fall short due to the linguistic variability and context dependence of human input in natural language [36].

Recent works [20]–[26], [34] have leveraged LLMs to improve intent translation accuracy. For example, NetConfEval [23] evaluates ChatGPT and Codellama on translating formal network specifications—such as reachability, waypoints, and load balancing—into structured JSON rules. Their dataset includes intents like “*Traffic originating from Istanbul can reach the subnet 100.0.9.0/24 via Rotterdam, using two paths*”. The work in [25] introduces another benchmark focused on NFV (Network Function Virtualization) configuration, covering service function chaining and resource allocation. Both studies use in-context learning [37], where example input-output pairs are embedded in the LLM prompt to guide translation. The NFV configuration work also explores a continuous learning setup, where previous corrections are reused to improve future translations.

Despite these advances, several limitations persist. First, many existing approaches rely on closed-source models such as ChatGPT [23], [24], which limit reproducibility due to closed-source nature, raise privacy concerns due to lack of transparency, and hinder real-world deployment due to the cost. Second, while previous work benchmarks LLMs for translation, they typically evaluate only a handful of models, for example, 3 in [22], 6 in [25], or 7 in [23]—and restrict evaluation to one or two datasets. Third, these studies focus only on abstract configuration goals (e.g., formal policies or NFV rules), overlooking practical translation targets like SDN controller-specific formats. Finally, there remains a critical gap in understanding how LLMs perform across a

wider range of IBN tasks, particularly when the target output must conform to real-world SDN controller schemas like ODL or ONOS.

B. INTENT ACTIVATION

Intent activation refers to the deployment of translated intents onto the network infrastructure. This phase encompasses two critical tasks: detecting and resolving conflicts between new and existing configurations, and correctly installing the validated configuration on the appropriate network devices.

Conflicts can arise from multiple overlapping intents that often issued by different users or systems and target the same network scope with contradictory goals. These conflicts may stem from policy misalignment, intent ambiguity, or resource contention.

Existing works [38], [39] described six types of conflicts in SDN environments and developed methods to identify them. In Table 2, we present 9 flow rules. This table is adapted from [38] to demonstrate different types of conflicts. A redundancy conflict occurs when a specific rule (Rule 2) is fully covered by a more general one (Rule 1) with the same action. Shadowing conflict is observed when a more general rule (Rule 4) with higher priority overrides a more specific rule (Rule 1) with a different action. Generalization conflict happens when a specific rule (Rule 5) with higher priority conflicts in action with a broader rule (Rule 1). Correlation conflict is identified between Rule 6 and Rule 1, where their match spaces partially overlap without a subset relationship and their actions differ. Overlap conflict, as seen between Rule 6 and Rule 7, involves intersecting address spaces with the same action. Lastly, Rule 4 and Rule 8 illustrate imbrication conflict, where rules overlap on one protocol layer (e.g., MAC) but not on another (e.g., IP), causing cross-layer ambiguity.

Traditional conflict resolution techniques—such as logical policy evaluation, verification against the network state, or graph-based mapping [11], [12], [40], [41]. For example, VeriFlow [42] and NetPlumber [43] enable real-time conflict detection in SDN by analyzing flow updates for violations such as loops or black holes, with support for basic automated resolution like blocking or flagging rules. However, they lack support for complex or non-flow-based intent conflicts. In contrast, Batfish [44] performs offline static analysis to detect configuration errors with detailed provenance but does not support real-time detection or automated resolution, making it less suitable for dynamic IBN scenarios. Furthermore, current approaches largely lack mechanism to explain why a conflict occurred, limiting their ability to clearly indicate underlying causes of conflicts or recommend actionable resolution strategies, thus highlighting the need for enhanced intent resolution methods. While some recent works have begun to explore conflict handling using LLMs, their scope remains narrow. For example, NetConfEval [23] demonstrates basic conflict detection using LLMs by identifying mutually exclusive reachability goals. However, to

TABLE 2: Flow rule table

Rule #	Priority	Source MAC	Dest MAC	Source IP	Dest IP	Protocol	Source Port	Dest Port	Action
1	61	*	*	192.168.10.0/24	172.16.1.100	tcp	*	*	forward
2	60	*	*	192.168.10.25	172.16.1.100	tcp	*	443	forward
3	62	*	*	192.168.10.25	172.16.1.0/24	tcp	*	*	forward
4	63	*	*	192.168.10.0/24	172.16.1.100	tcp	*	*	drop
5	64	*	*	192.168.10.25	172.16.1.100	tcp	*	*	drop
6	61	*	*	192.168.0.0/16	172.16.1.100	tcp	*	*	drop
7	65	*	*	192.168.10.25	172.16.1.0/24	tcp	*	4000–4010	drop
8	67	aa:bb:cc:dd:ee:01	ff:ee:dd:cc:bb:aa	*	*	*	*	*	forward
9	68	*	*	*	*	tcp	*	443	drop

extent to which LLM can do conflict detection for SDN environments remains underexplored.

C. INTENT ASSURANCE

Intent assurance represents the final and most iterative phase of the IBN lifecycle, responsible for continuously verifying that the operational network state aligns with the originally expressed user intents [45]. This involves not only validating the correctness of intent translation and deployment but also ensuring ongoing enforcement through real-time monitoring, predictive analytics, and intelligent feedback mechanisms. A critical challenge in this phase is managing intent drift [13], a condition where network behavior gradually deviates from the intended objectives due to dynamic changes in traffic patterns, topology, or device states. Effective intent assurance must detect such drift and trigger corrective actions to restore compliance and maintain intent fidelity over time.

Traditional approaches to intent assurance typically fall into static and dynamic categories [12]. Static methods verify if the intended configurations exist on the correct device and match the expected structure. For instance, in ODL, there are configurational data store and operational data store. Configurational data store reflects all installed flow rules (active or passive) while the operational data store reflects the active flow rules. For static assurance, the operational data store can be used to determine the status of flow rules. For example, the operational data store in ODL is queried to inspect whether the `packet-count` field in the `flow-statistics` section of a flow rule contains a non-zero value. A non-zero count indicates that the rule has matched traffic, suggesting packet drops if the rule is intended to drop packets, or forwarding if it is meant to forward them. However, static assurance often fails to detect runtime violations.

In contrast, dynamic assurance mechanisms rely on continuous monitoring of key performance indicators (KPIs) such as delay, throughput, packet loss, and queue utilization to ensure that the deployed flows exhibit the desired behavior. For example, VeriFlow [42] and NetPlumber [43] provide real-time flow rule assurance in SDN by incrementally checking rule compliance with network invariants or policies during updates, offering fast but limited verification focused on flow-level behavior. Batfish [44], on the other hand, performs static assurance by analyzing configurations pre-deployment using constraint solving, which enables thorough

checks but lacks the real-time responsiveness needed for dynamic IBN scenarios.

LLM-based approaches have only recently begun to touch on intent assurance. Although the work in [20] is implemented on a 5G testbed and introduces a modular LLM-centric framework to cover the full intent lifecycle, there is no evidence on how and to what extent intent assurance can be realized. Other works such as [46] explore fault localization using LLMs but do not provide a generalized assurance framework. While most existing LLM-based IBN systems [20], [22]–[26] do not implement assurance in a closed-loop or controller-aware manner, the work in [34] is among the earliest to propose an LLM-driven assurance system capable of detecting intent performance drift and triggering corrective actions. However, its reliance on the closed-source LLM ChatGPT may limit transparency, reproducibility, and broader adoption.

III. IBNBENCH: LLM-BASED IBN EVALUATION BENCHMARK

A. SELECTION OF LLMs

To ensure a focused yet practical evaluation, we selected 33 open-source LLMs, as in Table 3, based on a combination of resource feasibility, task relevance, and community adoption. Due to computational and time constraints, we limited the scope to a representative subset of models that are well-suited for intent translation and conflict detection—excluding models designed for unrelated domains. Additionally, we prioritized LLMs that are popular within the research community, frequently cited, and actively maintained. All selected models are readily deployable, making them suitable candidates for reproducible benchmarking and real-world integration in IBN systems. These LLMs vary in the number of parameters, which directly influence their language processing and generation capabilities. Larger models with more parameters generally exhibit better understanding and nuanced responses but at the cost of increased memory requirements, higher energy consumption and slower processing speeds.

B. SELECTION OF DATASETS

We benchmarked the LLMs using six datasets—four newly proposed by us and two existing ones [23], [25]. The creation process of the proposed datasets is detailed in Sec. IV, and a summary of all datasets is provided in Table 4.

TABLE 3: List of LLMs used for benchmarking (grouped by parameter size in billions)

Parameter Size	Models
1–3B	Starcoder:3b, Llama3.2:3b, Phi3:3.8b, Orca-mini:3b, Starcoder2:3b, TinyLlama:1.1b, Deepseek-coder:1.3b, Phi:2.7b
4–6B	Qwen:4b, Yi:6b
7–9B	Codellama:7b, Llama2:7b, Llama3:8b, Llama3.1:8b, Qwen2.5:7b, Openchat:7b, Marco-01:7b, Mistral:7b, Dolphin-Mistral:7b, Wizardlm2:7b, Codegemma:7b, Zephyr:7b, Llava-Llama3:8b, Qwen2:7b
10–20B	Mistral-nemo:12b, Deepseek-coder-v2:16b
20–30B	Gemma2:27b, Codestrail:22b
30B+	QwQ-abilitated:32b, QwQ-fusion:32b, QwQ:32b, Codellama:34b, Command-r:35b

TABLE 4: Summary of datasets used for LLM benchmarking; four newly proposed by us and two existing ones

Dataset	Proposed	Task	Samples	Conflict Pairs	Evaluation Metric
Intent2Flow-ODL	Yes	Translation	52	—	Semantic Accuracy, Runtime
Intent2Flow-ONOS	Yes	Translation	50	—	Semantic Accuracy, Runtime
Formal Spec. [23]	No	Translation	1500	—	Field Presence Accuracy, Runtime
NFV Config. [25]	No	Translation	120	—	Exact Structural Match, Runtime
FlowConflict-ODL	Yes	Conflict Detection	50	4	TP, TN, FP, FN, Runtime
FlowConflict-ONOS	Yes	Conflict Detection	62	10	TP, TN, FP, FN, Runtime

The LLMs are benchmarked for two different IBN tasks: intent translation (from natural language intent to JSON structured flow rules) and conflict detection. The benchmarking for intent translation is done using four different datasets. Among them, two belong to the proposed IBNBench (Intent2Flow-ODL and Intent2Flow-ONOS) mentioned in Sec. IV, the third one is the Formal specification dataset [23] and the fourth one is the NFV configuration dataset [25]. The reason to choose the datasets [23] and [25] is that they contain natural language intent and corresponding JSON formatted translation. The Formal specification dataset translates natural language intents into JSON structure for specifically three type of requirements: reachability, waypoints and load balancing, while the NFV configuration dataset include natural language intent and corresponding JSON formatted NFV configuration. However, these datasets are different from our proposed datasets. Our datasets specifically target ODL and ONOS SDN controllers and contains actual flow rules that were tested and verified. Hence, they can be readily used to benchmark any LLM for ODL or ONOS SDN controller application.

As for the benchmarking the LLMs for conflict detection task, we use our proposed IBNBench’s FlowConflict-ODL and FlowConflict-ONOS datasets. We do not include the Formal specification JSON and NFV configuration JSON. The reasons is that we detect conflict based on JSON structured configuration and all the datasets use JSON formatted configuration. Hence, it is sufficient to evaluate the LLMs on the ODL and ONOS flow rules as they represent well structured JSON configuration found in other datasets.

As for the sample size of the datasets, the NFV configuration dataset contains 120 pairs of samples, the Formal specification dataset comprises 1500 sample pairs. Our proposed datasets Intent2Flow-ODL and Intent2Flow-ONOS include 52 and 50 pairs of samples receptively, while the

FlowConflict-ODL and FlowConflict-ONOS datasets include 60 and 74 pairs of samples receptively.

For intent translation, each dataset was split, with 50% used as the source of context examples and the remaining 50% as the source of test cases. A context example (context example) in the LLM context is an input-output pair provided in a prompt to guide the model’s behavior in tasks like few-shot learning, demonstrating the desired format and content for the output. For conflict detection, 62 pairs of flow rules were selected from the FlowConflict-ONOS dataset, including 10 pairs with conflicts. From the FlowConflict-ODL dataset, 50 pairs were selected, of which 4 contained conflicts.

C. BENCHMARK METRICS

For evaluating the performance of LLMs in the intent translation task, we use accuracy as the primary metric. Besides we report the running time. The running time includes the time from the submission of the query to the LLM until the LLM produces the output, including the time for dynamic selection of context examples.

The method of computing accuracy varies across the four datasets, depending on the nature of their expected outputs. For the proposed Intent2Flow-ODL and Intent2Flow-ONOS datasets, a translation is considered correct if the generated JSON is semantically equivalent to the expected JSON—allowing for differences in field ordering, formatting, or numeric representation. In contrast, the Formal Specification dataset uses a more relaxed comparison: it considers a translation correct if key expected intent fields (e.g., reachability, waypoint, load balancing) are present in the result, without requiring full structural matching. For the NFV Configuration dataset, a stricter approach is adopted where accuracy is computed by checking for exact structural equality after recursively sorting dictionary keys and list elements.

As for evaluating the performance of LLMs in conflict detection task, we set one conflict as defined in Sec. II-B for the flow rule pairs given to an LLM. We record true positive, true negative, false positive, false negative for each LLM by comparing against the ground truth. Besides, we report the time duration from the submission of the prompt to the LLM until the LLM produces an output. In the benchmarking process, we ask LLMs to identifying potential conflicts between flow rules which requires examining whether multiple rules overlap in their matching criteria while prescribing different output actions. For example, two rules that match the same type of traffic, such as TCP packets on a specific port from the same input port, and apply different output ports or actions like drop versus forward, are considered conflicting. This is evident when two flow entries match the same traffic conditions but direct it differently, creating ambiguity in packet handling. Conflict detection, therefore, involves comparing match fields and examining whether their actions diverge for overlapping traffic.

D. LIMITATIONS

The evaluation focuses on three categories of intent: forwarding, security, and QoS. Broader categories of network intents, such as service function chaining or dynamic path optimization, are not currently evaluated. However, the benchmarking framework is designed to be extensible. New intent categories and flow rule patterns can be integrated in IBNBench, and we plan to expand it over time while enabling community contributions to promote broader coverage and reproducibility.

IV. DEVELOPING IBNBENCH

To address the gap in evaluating LLM-based systems on intent translation and detection of conflicting flow rules, we designed IBNBench, a set of four datasets, two for intent translation and two for conflict detection. The datasets for translating natural language intents into JSON formatted flow rules are named Intent2Flow-ODL and Intent2Flow-ONOS, based on the target SDN controller. As for the datasets developed for conflict detection, they are named FlowConflict-ODL and FlowConflict-ONOS according to the target SDN controller.

A. Intent2Flow-ODL AND Intent2Flow-ONOS DATASETS

Each of the datasets, Intent2Flow-ODL, Intent2Flow-ONOS, consist of 50 pairs of real-world-inspired network intents in natural language and their JSON formatted translation. Each intent specifies high-level operational goals such as routing, blocking, or prioritizing traffic within a programmable network environment. We consider that the user intents belong to three primary categories: *Forwarding*, *Security*, and *QoS* based on the semantics of their described action. Intents that included explicit blocking, dropping, or denying of traffic are under *Security*, while intents specifying packet forwarding without prioritization are under *Forwarding*. Intents specifying queue assignments, traffic prioritization, or latency/bandwidth guarantees are *QoS* intents. The datasets contain 22 forwarding intents, 10 security intents and 18 QoS intents.

To construct the datasets, we adopted both manual and LLM-generated content. A diamond topology was created using Mininet [47] for both ODL and ONOS, consisting of four OpenFlow switches and four hosts, as shown in Fig. 4. Based on this network, we crafted intents and produced their corresponding ODL/ONOS flow rules in JSON format. LLMs were leveraged to generate supplementary intents with variation in linguistic expression as well as corresponding JSON flow rules. Before inclusion in the datasets, these LLM-generated flow rules were manually corrected and deployed in switches to verify their effectiveness. We provide the full datasets on GitHub [48]. In Table 5, some examples of intents are shown.

ODL Flow Rules: flow rule is a structured configuration that defines how network traffic is managed within an SDN environment. In ODL, these rules are determined by pre-defined YANG model [49] and typically represented in

TABLE 5: Example intents from IBNBench

Intent Type	Example Intent
Forwarding	<ul style="list-style-type: none"> Forward traffic entering on port 1 of switch 2 to port 2. In switch 1, forward all TCP traffic not matching higher-priority rules through port 1.
Security	<ul style="list-style-type: none"> In switch 4, block all IPv4 traffic from 10.0.0.1 to 10.0.0.4 with a high priority, ensuring the switch operates as a firewall. Drop all packets with a source IP of 10.0.0.1 and destination IP of 10.0.0.4 using node 4.
QoS	<ul style="list-style-type: none"> Forward TCP traffic on port 80 destined for 10.0.0.3 via interface 2 of switch 1, assigning it to queue 0 for prioritized handling. Route HTTP traffic originating from 192.168.1.2 on port 1 of switch 4 and destined for 10.0.0.5/32 through port 2, ensuring packets are assigned to queue 0 for low-latency and apply VLAN tag 100.

JSON format. It includes fields such as flow IDs, table IDs, priority, match conditions (e.g., source/destination IP, protocol type), and actions (e.g., forwarding, dropping, or modifying packets). This structured approach allows ODL to efficiently describe and implement complex networking policies, making it suitable for configuring large-scale networks, traffic engineering, and advanced use cases like network slicing or QoS enforcement. In Appendix A, an example intent and its corresponding ODL flow rule representation is included. More details on ODL flow rules can be found in [49].

ONOS Flow Rules: In ONOS, flow rules serve the same purpose as in ODL, defining how packets should be processed based on structured match conditions and corresponding actions. These rules are expressed in JSON format and include key components such as device identifiers, match criteria, priority levels, and treatments. The match field specifies the conditions under which the rule applies, such as Ethernet type, IP protocol, and source or destination IP addresses. The action field, referred to as the treatment, outlines how packets should be handled when a match occurs—for example, forwarding to a specific port, dropping, or assigning traffic to a QoS queue. In Appendix A, an example intent and its corresponding ONOS flow rule representation is included. More details on ONOS flow rules can be found in [50].

B. FlowConflict-ODL AND FlowConflict-ONOS DATASETS

We used JSON formatted flow rules to create two new datasets to benchmark the LLMs on conflict detection task. They are FlowConflict-ODL and FlowConflict-ONOS datasets. FlowConflict-ODL contains 60 pairs of JSON formatted flow rules designed for ODL SDN controller and FlowConflict-ONOS contains 74 pairs of JSON formatted flow rules designed for ONOS SDN controller. As for the number of conflicting rule pairs, FlowConflict-ODL has 19 pairs of conflicting rules and FlowConflict-ONOS has 27 pairs of conflicting rules. These conflicting rules belong to six conflict category discussed in Sec. II-B. We provide the full datasets on GitHub [48].

V. NetIntent DESIGN

A. PROBLEM MODELING

Automating the full lifecycle of IBN from high-level natural language intent expression to low-level SDN configuration

and validation requires a principled formulation of the underlying decision process. We model this IBN orchestration as a mapping from user intents and network context to controller-executable configurations into the network devices.

Let \mathcal{I} denote the space of user-defined natural language intents (e.g., QoS policies, security rules, routing goals), and let \mathcal{N} represent the network context, including topology, installed configurations, and performance metrics. The goal is to synthesize a configuration $c \in \mathcal{C}$, where \mathcal{C} is the space of target configurations that can be applied to the SDN controller, such that:

$$\Pi : \mathcal{I} \times \mathcal{N} \rightarrow \mathcal{C} \quad (1)$$

This configuration must accurately implement the operator's intent under the current network state. We define a semantic alignment function $\text{SemMatch}(i, c)$, which returns 1 if the configuration c satisfies the operational goals embedded in the intent i , and 0 otherwise:

$$\text{SemMatch}(i, c) = 1 \quad (2)$$

This process must support diverse intent types including forwarding, security, and QoS, across heterogeneous SDN controllers (e.g., ODL, ONOS). The mapping is further constrained by the compliance with the controller-specific schema.

To ensure safe deployment, the generated configuration must be valid and non-conflicting with respect to the current configuration state C_{curr} (e.g., overlapping match fields with contradictory actions). This requires satisfying two constraints: (1) schema validity, expressed as $\text{Valid}(c) = 1$, and (2) absence of rule-level conflict with existing flow entries, defined as $\text{Conflict}(c, C_{\text{curr}}) = 0$.

If a conflict is detected, the system resolves it using a deterministic policy function:

$$\rho : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C} \quad (3)$$

Here, the first argument to ρ represents the newly generated candidate configuration, and the second argument represents an existing configuration already deployed in the network. Although both inputs are elements of the same configuration space \mathcal{C} , their roles are distinct in the resolution process. The function ρ selects which configuration to prioritize, either retaining the new rule or preserving the existing one, based on factors such as intent type (e.g., security over QoS or forwarding), match specificity, and contextual metadata.

After successful installation, the system must verify whether the deployed configuration produces the intended behavior in the live network. This process, known as *intent assurance*, compares the actual operational state N_{obs} , including real-time telemetry such as packet counters, queue lengths, and latency metrics, associated with the deployed rule c , against the behavioral expectations implied by the original intent i . An assurance function is defined as:

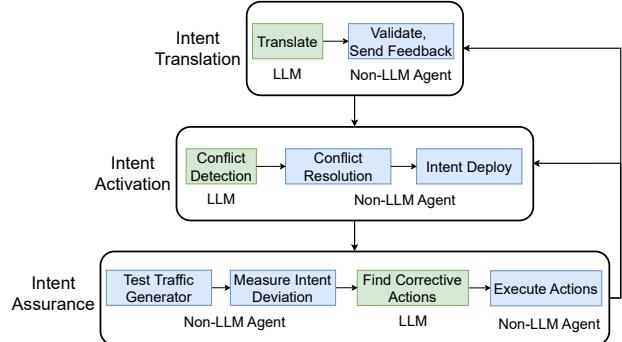


FIGURE 2: NetIntent overview.

$$\text{Assure}(i, c, N_{\text{obs}}) = \begin{cases} 1, & \text{if behavior aligns with intent} \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

The ultimate objective is to maximize the number of intents that are semantically realized, safely activated, and successfully assured. Given a batch of intents $\{i_1, i_2, \dots, i_n\}$, the autonomous orchestration of the intent-based SDN system becomes:

$$\max_{\Pi} \sum_{j=1}^n \left[\text{SemMatch}(i_j, c_j) \cdot \text{Valid}(c_j) \cdot \neg \text{Conflict}(c_j, C_{\text{curr}}) \cdot \text{Assure}(i_j, c_j, N_{\text{obs}}) \right] \quad (5)$$

This formulation abstracts the IBN pipeline as a composite optimization problem, where the orchestration function Π performs a sequence of reasoning and verification steps. Each step is assisted by LLMs through prompt engineering, conflict resolution, fallback mechanisms, and closed-loop assurance. In the next section, we present our proposed NetIntent system, which operationalizes this formulation to autonomously solve the orchestration problem across the full intent lifecycle.

B. NetIntent OVERVIEW

In Fig. 2, we illustrate an overview of NetIntent, the proposed end-to-end IBN system. This architecture leverages LLMs and non-LLM agents to automate the processes of intent translation, activation, and assurance. In the translation stage, user-defined natural language intents are fed into the system. An LLM translates the user's intent into a JSON-formatted configuration for the target SDN controller. This stage is supported by non-LLM agents, including a JSON validator that checks the syntactic and structural correctness of the LLM output, and a feedback module that prompts the LLM to refine or correct its output based on validation results. The next stage is intent activation which does conflict detection, conflict resolution, and intent deployment. Here, an LLM checks the newly translated configuration for conflicts against existing configurations. If

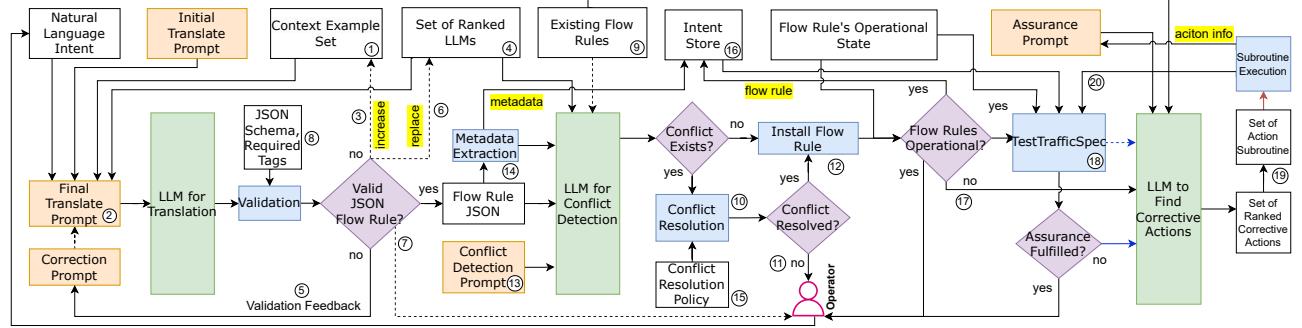


FIGURE 3: NetIntent architecture; end-to-end intent-based automation.

a conflict is found, a resolver tries to resolve it. Once this is done, the new configuration is deployed on the target SDN controller. If this succeeds, the third stage begins, that is, intent assurance. Here, non-LLM agents provide the LLM information on intent drift and current state of the device. LLM then generates corrective actions which are executed by code modules. This is continued in a closed-loop fashion to provide intent assurance. In Fig. 3, we illustrate the full architecture of NetIntent. In the following sections, we describe the architecture in details.

C. NetIntent WORKFLOW and DETAILS

1) INTENT TRANSLATION

NetIntent asks the network operator for intent, expressed in natural language. Supported by context examples and prompt, LLM starts translating the intent. The carefully designed prompt guides the LLM to translate intent into JSON structured configuration. The output is then fed to a validator to check for any syntax error or missing tags necessary for the target SDN controller. If the configuration is not valid, a feedback is sent back to the LLM automatically. Several data is given to the LLM to fix the error which includes the flow rule configuration, the validation feedback, and a new guiding prompt. This prompt instructs the LLM to fix the error and regenerate the corrected configuration. This process is repeated until a valid configuration is obtained. Once a valid configuration is found, it serves as the SDN-controller-readable representation of the user's intent, prepared to be sent to the intent activation stage. In Algorithm 1, we formally describe the intent translation process.

Next, we describe the related components such as context examples, set of LLMs, and validator, which play very important roles in intent translation.

a: CONTEXT EXAMPLE SELECTION

The set of context examples (① in Fig. 3) is a dataset consisting of input and expected output pairs for the intent translation task. For few-shot learning of LLM, these input-output pairs are used as context examples which tell LLM how the output should look like for the corresponding input. Studies [51], [52] have shown that adding more examples in the LLM prompt helps produce better output. Moreover,

Algorithm 1 Intent Translation using Ranked LLMs

```

Input: User intent  $U$ , context examples  $\mathcal{C}$ , ranked LLMs  $\mathcal{M} = \{M_1, M_2, \dots, M_k\}$ , initial context count  $x_0$ , maximum context examples  $Y$ 
Output: Valid JSON configuration  $\Phi$  or validation feedback  $\epsilon$ 

1: for each LLM  $M_i \in \mathcal{M}$  do
2:    $x \leftarrow x_0$ 
3:   while  $x \leq Y$  do
4:     Construct prompt using  $U$  and relevant- $x$  examples from  $\mathcal{C}$ 
5:     Generate output  $\Phi = M_i(\text{prompt})$ 
6:     Validate  $\Phi$  against SDN controller specification
7:     if  $\Phi$  is valid then
8:       return  $\Phi$ 
9:     else
10:      Extract validation error information  $\epsilon$ 
11:      Update the correction prompt with  $\epsilon$  and  $\Phi$ 
12:       $x \leftarrow x + 1$ 
13:    end if
14:   end while
15: end for
16: return Report with validation feedback  $\epsilon$  to operator for manual resolution

```

the more relevant the examples are to the target task, the more accurate result can be found from LLM. As outlined in Sec. V-C-1-b, the context examples are incorporated in the prompt (②). We update the number of context examples fed to the LLM prompt when necessary which is indicated by dotted line (③). For instance, when LLM fails to generate a valid configuration with a given set of context examples, the number of context examples is increased. In Algorithm 1, we formally described it. However, there should be a trade-off between LLM output accuracy and its response time since adding more context examples to the LLM prompt increases the output generation latency. As for the selection of context examples for the target task, we use the strategy called Max Marginal Relevance example selector [51]. It ensures that

the chosen examples are not only highly similar to the input query, but also maximally diverse from each other, preventing redundancy and improving the overall quality and coverage of the provided context. The reason we leverage dynamic context example selection for intent translation is that the input intents can be linguistically diverse and network can have thousands of different configurations. Fixed examples in prompt work well for a small number of use cases but become impractical with larger use cases due to LLM context length limitations and increased processing time. In order to provide context examples for intent translation, we use Intent2Flow-ODL and Intent2Flow-ONOS datasets.

b: LLM PROMPT FOR INTENT TRANSLATION

Here, we describe how the prompt is designed to guide the LLM for intent translation task. A prompt is the input instruction in text form given to the LLM to elicit a desired output. The prompt we design guides the LLM to generate a JSON formatted configuration for the target SDN controller’s device. We chose JSON format due to several advantages. JSON templates are widely used in configuration tools which allows for smooth integration into deployment workflows. Additionally, LLMs, trained on vast amounts of data, are already familiar with JSON examples which enhances their accuracy in generation of JSON outputs.

The prompt we designed for intent translation consists of four parts: general instructions, the output template, examples, and user intent. The prompt begins by defining the task of converting natural language network intents into JSON-formatted flow rules for the target SDN controller. It explicitly states that the response must contain only valid JSON, with no natural language. Controller-specific nuances, such as omitting the *treatment* field in ONOS to drop traffic, are clearly explained. The prompt includes a detailed JSON template that reflects the expected data model of the target controller. Mandatory and optional fields are carefully annotated. Usage rules are specified—for example, assigning high priority to queue-based flows or including VLAN fields only when explicitly required. Higher priority is also assigned for security related flows such as blocking a certain prefix. While not embedded in the prompt body, context examples are dynamically inserted at runtime after the JSON template part. The last part of the prompt contains the actual operator-provided intent. Additionally, a specialized slicing prompt is used to detect and extract slice-related metadata such as *switch_id*, *queue_id*, and *port_id*, etc. The slicing prompt follows the same design structure. In Table 6, the structure of the prompt is described. We provide the full prompt details on GitHub [48].

c: MULTI-LLM COORDINATION

We use more than one LLM in NetIntent (④) so that the repeated failure of one LLM does not put the system in an infinite loop with no progress. If there are N LLMs in the system, there will be a ranking of the LLMs based on their performance. Starting from the top-ranked model,

TABLE 6: Structured LLM prompt input

Item	Description / Source
Intent	User-provided natural language description of desired network behavior or policy
Intent Type Classifier	Determines whether the intent relates to queuing/slicing (QoS) or not, using a rule-based or LLM-driven slicing prompt
Context Examples	Few-shot in-context examples drawn from the same dataset to aid translation performance
Controller Type	Identifier indicating the SDN controller (e.g., ODL or ONOS), used to guide schema selection
Schema Template	JSON policy format that matches the controller’s northbound API specification (e.g., ODL-style flow entries or ONOS-style flow rules)
Match Fields	Derived from the intent (e.g., source/destination IPs, transport ports, IP protocol, in-port) to define traffic selectors
Treatment Fields	Derived from the intent (e.g., output port, drop, queue ID, VLAN tag) to define actions taken on matched traffic
Flow Metadata	Fields such as flow ID, priority, timeout, and flow name—either set to defaults or policy-specific values
Optional Fields Filter	Logic to ensure optional fields (e.g., queue, VLAN, port) are only included if explicitly mentioned in the intent
Validation Rules	Structural and semantic checks to ensure JSON output conforms to the controller’s schema and is conflict-free
Fallback Behavior	If the intent cannot be reliably translated into a policy, the LLM should return an empty JSON object

NetIntent incrementally augments the prompt with additional context examples and validation feedback (⑤) until a valid output, i.e., correct JSON formatted flow rule, is produced or the maximum context example limit is reached. In no valid output is generated and context example limit is exceeded, then the current LLM is replaced (⑥). However, if no LLM generates a valid configuration, a validation report is returned for manual intervention (⑦). In Sec. VI, we benchmarked several LLMs which can be used to rank them. In Algorithm 1, we describe how the NetIntent updates the context example and LLM dynamically.

d: LLM OUTPUT VALIDATOR

After an LLM generates a configuration in JSON format, it is sent through a validator to check for syntax error as well as conformity with the required tags and values for the target SDN controller. Although, LLM itself can check this, adding a separate validator further ensures that only the correct configuration is sent to the SDN controller device. NetIntent loads ⑧ controller-specific syntax rules and required tags, checks for structural and controller-specific syntax violations, and verifies the presence of all necessary tags. The tags needed depend on the intent type, which we describe in Sec. V-C-2-c. If no errors are found, the valid configuration is returned; otherwise, a validation feedback detailing the errors and their locations is produced.

2) INTENT ACTIVATION

In the intent activation stage, the generated configuration is checked for conflicts with the configurations already installed in the target SDN controller device. If the new configuration has no conflict with existing configurations, then it is installed on the target SDN controller device.

a: CONFLICT DETECTION

We leverage LLM to find conflicts between pairs of JSON formatted configurations, also called flow rules. They are usually saved in the intent *Configuration data store* [5] in ODL SDN controller and in *FlowRuleStore* [53] in ONOS

SDN controller. In Fig. 3, the conflict detection module retrieves existing configurations from the target SDN controller (shown in ⑨) and compare against the new flow rule using LLM. For accurate conflict detection, writing an appropriate prompt is necessary. The prompt describes how to handle the JSON flow rules and what constitutes a conflict. Unlike intent translation, we use single LLM in the conflict detection system. This is based on our evaluation of LLMs on conflict detection task where we found that a well-prompted LLM can detect almost every conflict. However, if multiple LLMs are used for conflict detection, it is recommended to use them in a voting fashion where LLM’s decision on conflict existence will be treated as a vote and the final decision will be based on majority votes. In Algorithm 2, we show how conflict detection is done through LLM. If any conflict is found, the system tries to resolve the conflict (⑩ in Fig. 3) as described in Sec. V-C-2-c. If it cannot be resolved, NetIntent sends the detail of the conflict to the operator (⑪) and does not attempt to install the new flow rule. In case there are no conflicts, NetIntent attempts to install the flow rule in the target SDN controller device (⑫), as described in Sec. V-C-2-d.

b: LLM PROMPT FOR CONFLICT DETECTION

To detect conflicts between flow rules, we designed controller-specific prompts (⑬ in Fig. 3) that guide an LLM to perform strict, field-by-field comparisons of flow configurations in JSON format. The prompt logic is customized for each SDN controller—ODL and ONOS—based on their schema and semantics. A conflict is identified only when two conditions are simultaneously satisfied: the match criteria of both rules overlap, and their actions are contradictory. Overlap requires both flows to specify identical or overlapping values for key fields such as source and destination IP addresses, transport protocols, and port numbers. For IP fields, CIDR prefix matching is used to assess overlap; however, if any critical field is missing in either flow, the match for that field is considered non-overlapping. Once overlapping criteria are established, the prompt checks whether the actions differ in a meaningful way. For ODL, this includes mismatches in output ports, presence of a *drop-action* in one rule versus an output-action in another, or conflicting queue IDs in *set-queue-actions*. For ONOS, action differences are determined by inspecting the *treatment* field; examples of conflict include differing output ports, presence of a queue instruction with different *queueId*, or cases where one rule uses *NOACTION* (interpreted as drop) while the other forwards packets. Importantly, priority is disregarded in this stage; priority determines which rule takes effect in deployment but not whether a conflict exists. Since a priority value may be set incorrectly by LLM during translation, reporting conflict based on that may produce false positives. However, we don’t overlook the priority, rather we resolve priority if LLM reports a conflicts. The resolved priority takes effect in real network operation where packets are matched by the network device based on rule

TABLE 7: Structured LLM prompt input for conflict detection

Item	Description
Flow 1 (JSON)	First flow rule in controller-specific JSON format (ODL or ONOS)
Flow 2 (JSON)	Second flow rule to be compared with Flow 1
Match Fields	Literal comparison of fields: IP source/destination, protocol, ports, in-port, and Ethernet type
Wildcard Handling	Missing fields are treated as general (apply to all); mismatched fields imply no overlap
Conflict Conditions	Triggered only when match fields overlap <i>and</i> actions contradict (e.g., drop vs. forward, different output ports or queues)
Exceptions	Additional match fields in one rule that are absent in the other prevent conflict declaration
Priority Handling	Ignored during conflict detection
Action Comparison	Drop vs. forward, differing queues or ports are considered conflicting if matches align
Conflict Output	JSON object with <i>conflict_status</i> (0/1) and <i>conflict_explanation</i> string

priority. The LLM is instructed to return a structured JSON response with a conflict status field and an explanation field. In Table 7, the structure of the prompt is described. We provide the full prompt on GitHub [48].

c: CONFLICT RESOLUTION

To determine which flow rule should take precedence in the event of a detected conflict, we propose a resolution policy denoted as \mathcal{P} . This policy defines a structured set of decision rules used to evaluate and prioritize conflicting flow configurations based on their semantic intent, specificity, and contextual metadata. This is shown in Fig. 3 in ⑯. To use \mathcal{P} , we first classify the flow rules. Next, we describe classification approach.

Flow Rule Classification: During the conflict detection stage, each flow rule is annotated with metadata (⑭ in Fig. 3) to support downstream reasoning and resolution. This metadata includes the rule’s *type* (e.g., security, forwarding, or QoS) and a numerical measure of *specificity*, which reflects how narrowly scoped the rule is. To extract this metadata, we parse the flow rule translated by LLM. The metadata fields are inferred as follows:

- **Type:** It is determined from the rule’s action structure. For ODL, this includes fields within *apply-actions.action* such as *drop-action*, *set-queue-action*, or *output-action*. For ONOS, this is inferred from the *treatment.instructions* field, e.g., *type: NOACTION* implies *security*, *QUEUE* implies *qos*, and *OUTPUT* implies *forwarding*. Also, the absence of *treatment.instructions* field implies *security*.
- **Specificity:** It is computed as the number of explicit match fields in the rule. For ODL, these are found under the *match* field; for ONOS, under *selector.criteria*. If the rule includes IP-based fields with CIDR notation (e.g., $10.0.0.1/32$), the prefix length is normalized (e.g., $/32 \rightarrow 1.0$) and added to the specificity score to reflect its precision.

Conflict Resolution Policy: The resolution policy \mathcal{P} is a deterministic rule-based mechanism designed to evaluate conflicting flow rules based on a predefined set of priorities. Using the flow rule classification data, the logic first checks whether either rule represents a security intent or not. If so,

Algorithm 2 Intent Activation using LLM

Input: Flow rules ϕ_1, ϕ_2 , SDN controller S , LLM M , resolution policy \mathcal{P}

Output: Conflict status, explanation, and resolution decision

```

1: Construct prompt for  $S$ 
2: Provide  $\phi_1, \phi_2$  as input to  $M$ 
3: Query  $M$  and parse response ( $conflict\_status$ ,  $conflict\_explanation$ )
4: if  $conflict\_status = 1$  then
5:   Use Metadata inference algorithm to infer metadata
      for  $\phi_1, \phi_2$ 
6:   goto Conflict Resolution
7: end if
8: return ( $conflict\_status = 0$ ,  $conflict\_explanation = ""$ )
Conflict Resolution:
9: Evaluate  $\phi_1, \phi_2$  using policy  $\mathcal{P}$  with inferred metadata
10: if resolution decision cannot be made (e.g., policy  $\mathcal{P}$ 
     yields no clear priority) then
11:   Generate detailed conflict report including  $\phi_1, \phi_2$ ,
      and  $conflict\_explanation$ 
12:   Send report to network operator for manual resolution
13:   return ( $conflict\_status=1$ ,  $conflict\_explanation$ ,  $priority\_rule=None$ )
14: else
15:   Determine priority rule  $\Phi^*$  based on rule type, specificity,
      and priority
16:   Deploy  $\Phi^*$  to controller  $S$ 
17:   Log the non-priority rule for audit or operator review
18:   Optionally notify assurance module or user of conflict resolution outcome
19:   return ( $conflict\_status = 1$ ,  $conflict\_explanation$ ,  $priority\_rule = \Phi^*$ )
20: end if
```

the security-related rule is favored, reflecting the importance of enforcing access control and traffic blocking over general forwarding or performance behaviors. If both rules are of the same type, the policy then compares their specificity. The more specific rule that typically targets a narrower traffic subset is preferred to ensure precise enforcement. If both rules are of equal type and specificity, the policy falls back to comparing their assigned priority values, selecting the one with the higher priority. This approach ensures that critical intents are not inadvertently overridden by broader or lower-priority rules. While this default logic aligns with common security and operational goals, it can be extended or customized to reflect domain-specific resolution strategies or business policies. In Algorithm 2, we formally describe the steps of conflict detection and resolution approach.

d: FLOW RULE INSTALLATION

If no conflicts are found or an identified conflict is resolved, the JSON flow rule is pushed to the target device of the target

SDN controller (12 in Fig. 3). For ODL, it is installed in ODL's Configuration data store via the RESTCONF API. For ONOS, it is installed in FlowRuleStore via REST API. To install the flow rule, first we infer the device ID of the target SDN controller where the rule will be installed from the input intent. Then we use POST method for ONOS and PUT method for ODL to install the flow rule to the target device. If the installation succeeds without any error, we verify if the rule is reflected on the target device. For ODL, we check if it is present in ODL's Operational data store while for ONOS, we check if it is present in FlowRuleStore. We keep a copy of the installed intent, its JSON flow rule, metadata, along with the target device information in our system. We save it in a file called "IntentStore" for future reference. The IntentStore is updated with the insertion and deletion of flow rules in the system.

3) INTENT ASSURANCE

Intent assurance is the mechanism by which an IBN system continuously validates and enforces that the network state and behavior align with the user-defined intents. To implement it, the intent assurance module of NetIntent does the following tasks: 1) Verify that the flow rules generated from user intents are correctly applied on all relevant devices by utilizing real-time traffic. 2) Trigger corrective actions if any intent is not being fulfilled.

These tasks are repeated in a closed-loop fashion for continuous monitoring of the system. Below, we describe them in details. In Algorithm 3, we formally describe the closed-loop assurance module of NetIntent. Next, we describe each step in details.

a: TEST TRAFFIC GENERATION

The *TestTrafficSpec* module is responsible for generating synthetic traffic tailored to the intent under verification. In Fig. 3, it is shown near (18). It provides a structured interface for specifying the traffic profile required to exercise an installed flow rule and serves as the foundation for capturing expected packet and byte-level behavior in the data plane. Operating in coordination with the assurance engine, the module produces traffic that aligns with the flow rule's match criteria, directionality, and semantic purpose (e.g., forwarding, blocking, or QoS enforcement).

Forwarding Intents: For intents involving basic packet forwarding, simple ICMP ping is sufficient. The number of packets transmitted is explicitly specified in the *TestTraffic-Spec*, enabling direct comparison with the observed *packet-count* delta ($\Delta S.packet-count$) for the flow rule.

Security (drop) Intents: These are verified similarly using ICMP ping to ensure that packets match the drop rule and are not forwarded. This lightweight approach provides deterministic control over packet generation with minimal overhead.

QoS (queue) Intents: For intents involving queue assignment or traffic prioritization, ping is insufficient. The module uses TCP-based synthetic traffic (e.g., generated via tools

Algorithm 3 Closed-Loop Intent Assurance using LLM

```

Input: Intent, IntentMetadata, NodeID, TableID,
FlowID, [QueueID], TestTrafficSpec, ControllerInfo,
MaxAttempts
Output: Verified Intent or User Alert
1: for attempt = 1 to MaxAttempts do
2:   Retrieve FlowRuleJSON, IntentType from IntentStore
3:   Retrieve installed flow rule using NodeID, TableID,
FlowID
4:   if rule missing or mismatched then
5:     Reinstall rule from IntentStore
6:     continue
7:   end if
8:   Retrieve  $S_{\text{initial}}$  (packet, byte, and queue stats if
applicable)
9:   Transmit test traffic; record  $T_{\text{start}}$ ,  $T_{\text{end}}$ 
10:  Retrieve  $S_{\text{final}}$  and compute  $\Delta S = S_{\text{final}} - S_{\text{initial}}$ 
11:  if IntentType = “Forwarding” then
12:    if  $\Delta S.\text{packet-count} < \text{ExpectedPacketCount}$  then
13:      goto LLM-Remediation
14:    else
15:      Log: “Forwarding verified”; return Verified
16:    end if
17:  else if IntentType = “Security” then
18:    if rule has forwarding behavior (ODL: output-
action, ONOS: not NOACTION) then
19:      goto LLM-Remediation
20:    else if  $\Delta S.\text{packet-count} < \text{ExpectedBlockedPackets}$ 
then
21:      goto LLM-Remediation
22:    else
23:      Log: “Security verified”; return Verified
24:    end if
25:  else if IntentType = “QoS” then
26:    Let  $B_t$ ,  $R_t$  = expected byte count, rate;  $\Delta B$  =
change in tx_bytes[QueueID]
27:     $R_{\text{measured}} = (\Delta B \times 8) / (T_{\text{end}} - T_{\text{start}})$ 
28:    if  $(\Delta B < \alpha B_t)$  or  $(|R_{\text{measured}} - R_t| > \epsilon)$  then
29:      goto LLM-Remediation
30:    else
31:      Log: “QoS verified”; return Verified
32:    end if
33:  end if
34: LLM-Remediation:
35:   Compile current context: FlowRule, Deviation-
Metrics, TestTrafficSpec, ControllerInfo
36:   Query LLM with context for ranked root causes and
recommended actions
37:   for each action  $a$  in ranked action list do
38:     Map  $a$  to corresponding subroutine
39:     Execute subroutine
40:     break (to retry assurance in next iteration)
41:   end for
42: end for
43: Escalate to operator.
  
```

like *iperf*) to verify both byte volume and data rate. The traffic is generated with a predefined size (B_t) and duration, and the observed throughput (R_{measured}) is compared to the expected rate (R_t) as part of assurance.

To support controller-agnostic deployment, the *TestTrafficSpec* includes source/destination resolution, traffic characteristics (e.g., volume, duration, protocol), and expected performance metrics.

b: IDENTIFYING INTENT DRIFT

NetIntent identifies intent drift by dealing with the dual challenge of verifying that (i) high-level intents have been correctly translated and deployed as low-level flow rules, and (ii) these flow rules are actively enforcing the intended behavior in the data plane.

These challenges becomes more nuanced across diverse intent types, such as forwarding, security, and QoS, each of which requires distinct criteria and metrics for effective verification. To address this, we maintain a storage called *IntentStore* that retains the original natural language intent, its translated JSON flow rule, and relevant metadata (e.g., intent type, specificity, and expected traffic characteristics). It is shown in Fig. 3 (16). During the assurance stage, we iterate over each intent and retrieve the corresponding flow rule using its unique identifiers: *NodeID*, *TableID*, and *FlowID* from the target SDN controller. We verify that the deployed rule is present and structurally consistent with its stored specification. If not, a corrective action is sought immediately using LLM (17). In Sec. V-C-2-c, we describe how corrective actions is taken.

Upon confirming rule presence, we collect its initial operational statistics, including *packet-count* (or *packets*), *byte-count* (or *bytes*), and where applicable, queue-level counters retrieved from the data plane via standard switch-level interfaces (e.g., Open Virtual Switch (OVS)). We denote the statistics collected prior to traffic generation as S_{initial} and those collected afterward as S_{final} . Their difference, $\Delta S = S_{\text{final}} - S_{\text{initial}}$, reflects the observed activity of the rule over the assurance window.

To exercise the rule, we generate synthetic test traffic via our *TestTrafficSpec* module, as described in Sec. V-C-3-a, which produces packets matched to the flow rule’s criteria and includes an expected byte volume B_t , expected packet count P_t , and expected transmission rate R_t . After the traffic completes, we analyze ΔS in conjunction with B_t , P_t , and R_t to determine whether the intent has been correctly enforced. The evaluation criteria depend on the type of intent under test, as described next.

Forwarding Intents: For forwarding intents, we validate whether the installed flow rule actively forwards matching packets. Let S_{initial} and S_{final} denote the flow’s operational statistics before and after test traffic, respectively. We define the packet-count delta as $\Delta S.\text{packet-count} = S_{\text{final}}.\text{packet-count} - S_{\text{initial}}.\text{packet-count}$. We expect this to align with the expected test packet count, denoted $P_t = \text{ExpectedPacketCount}$, derived from the test specification. A

value of $\Delta S.\text{packet-count} < P_t$ may indicate forwarding misbehavior due to rule shadowing, incorrect match fields, or silent drops.

Security Intents: For drop or block intents, assurance requires confirming that packets match the rule but are not forwarded. We verify that $\Delta S.\text{packet-count}$ increases, indicating that traffic was matched. Drop behavior is determined by rule semantics: in ODL, a drop rule explicitly includes *drop-action*:`{}` in the *apply-actions* part; in ONOS, it is indicated by the presence of only *NOACTION* in the *treatment.instructions* list. A rule is considered verified for a security intent if it satisfies these controller-specific drop semantics and $\Delta S.\text{packet-count} \geq B_p$, where $B_p = \text{ExpectedBlockedPackets}$ is the expected number of dropped packets.

QoS Intents: To validate QoS intents, we ensure that traffic matching the flow rule is correctly classified into the assigned queue and forwarded with expected volume and rate. Let B_{before} and B_{after} be the queue's *tx_bytes* counter before and after test traffic. The byte-count delta is defined as $\Delta B = B_{\text{after}} - B_{\text{before}}$. The test specification defines $B_t = \text{ExpectedByteCount}$ and $R_t = \text{ExpectedRate}$ as the expected volume and rate. We record the start and end timestamps T_{start} and T_{end} to calculate the observed data rate as $R_{\text{measured}} = (\Delta B \times 8) / (T_{\text{end}} - T_{\text{start}})$. A QoS intent is considered successfully enforced if both the volume and rate satisfy: $\Delta B \geq \alpha \times B_t$ and $|R_{\text{measured}} - R_t| \leq \epsilon$, where α is a volume margin factor (e.g., 0.98) and ϵ is a tolerance for rate deviation.

This validation framework supports both ODL and ONOS while it can be extended to other controllers. It enables dynamic intent assurance across SDN platforms. In Algorithm 3, we formally describe the intent assurance stage of NetIntent. This assurance mechanism enables the system to automatically identify stale, misbehaving, or misconfigured intents by correlating observed network behavior with expected outcomes defined in the test traffic specification. Upon detection of inconsistency between control plane configuration and data plane behavior, corrective actions are triggered to restore alignment.

c: TRIGGERING CORRECTIVE ACTIONS

If an intended rule does not exist in the target device, then the rule will be reinstalled using the information of *IntentStore*. However, if the rule exists and found to be identical to the originally installed one but it fails to meet its expected behavior during assurance, it becomes necessary to identify the root cause and apply corrective actions. While deterministic mappings from intent types to static fixes can handle known failure modes, they fall short when facing ambiguous behaviors or misconfigurations in underlying infrastructure such as queue bindings or port mappings. To overcome this, we integrate LLM as an intelligent diagnostic component that can reason over structured context and recommend ranked corrective actions.

TABLE 8: Structured LLM prompt input for generating corrective actions

Item	Description
Intent	Natural language description and metadata from <i>IntentStore</i>
Flow Rule	JSON-encoded rule deployed for the intent
TestTrafficSpec	Expected traffic behavior (packet count, byte volume, rate, source/destination)
Deviation	Summary of assurance failure metrics from ΔS
Installed Rules	JSON list of all flow rules installed on the device (from controller API)
Queue Stats	Switch-level QoS configuration (via <i>ovs-vsctl list qos/queue</i>)
Controller	Controller type identifier (e.g., ODL or ONOS)
Optional Feedback	Previous corrective actions and whether they succeeded

TABLE 9: Common action types and descriptions

Action Type	Description
check_match_fields	Re-evaluate the accuracy of source/destination, ports, and protocol in the flow rule
increase_priority	Raise flow rule priority to avoid shadowing
verify_queue_mapping	Check if the flow's <i>queue-id</i> is valid and mapped to the correct port
retranslate_intent	Use the LLM to regenerate a refined version of the flow rule
remove_output_action	For security intents, remove unintended forwarding behavior

Prompting Strategy: To enable contextual reasoning, the LLM is provided with a prompt that includes relevant data as shown in Table 8. The LLM is instructed to:

- Analyze the given context to identify potential root causes of the observed deviation.
- Rank the causes by likelihood and severity.
- Propose corrective actions for each cause.
- Output in structured format (e.g., JSON or numbered list) to support automated parsing.

Corrective Action Execution Engine: The LLM's output is parsed and mapped to a library of predefined actionable routines (⑯ on Fig. 3). These routines correspond to common remediation procedures mentioned in Table 9.

The actions are executed in the order ranked by the LLM. After each action (or set of actions), the assurance process is rerun to measure whether the deviation is resolved (⑰). If not, the failed result is appended to the prompt for the next round, enabling a feedback loop where the LLM can refine its hypothesis and suggest deeper diagnostics.

Closing Loop: The corrective framework described above is realized as a closed-loop control process, summarized in Algorithm 3. The loop continues and the network operator is informed of the intent assurance updates. This ensures adaptive and context-aware remediation based on real-time deviation analysis and feedback from prior attempts.

4) NetIntent's EXTENSIBILITY TO OTHER SDN CONTROLLERS

NetIntent is designed with modularity and controller-agnostic principles, making it extensible to a wide range of SDN controllers beyond ODL and ONOS. Its architecture separates controller-specific logic from core functionalities like intent translation, conflict detection, and assurance. To adapt NetIntent to another SDN controller, one would need to develop a controller-specific schema template that defines the required JSON structure for flow rules, compatible with the new controller's northbound API. Additionally, the validator

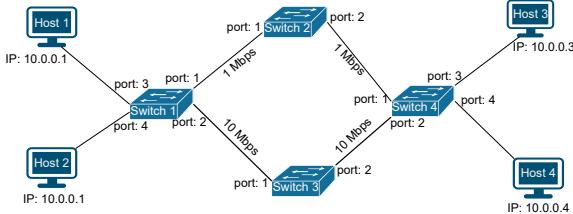


FIGURE 4: Topology used for flow rule installation

component should be updated to incorporate the syntax rules and required tags specific to the new controller. By leveraging the existing modular components and updating only the controller-specific modules, NetIntent can be effectively extended to support other SDN environments.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

A. HARDWARE SETUP

To evaluate the open-source LLMs, we utilized an AMD Ryzen Threadripper PRO 5995WX 64-core processor, 500 GB of RAM, and an NVIDIA RTX A6000 GPU with 48 GB of VRAM.

B. SOFTWARE USED FOR IMPLEMENTATION

We implemented NetIntent using Python programming language. The LangChain [54] library was used to manage all LLM operations. For ODL installation, Karaf-0.8.4 was used and for ONOS, version 2.0.0 was used. We used Mininet [47] to define a diamond-shaped network topology shown in Fig. 4. The topology comprises four switches (s_1-s_4) and four hosts (h_1-h_4), where h_1 and h_2 are connected to s_1 , and h_3 and h_4 are connected to s_4 . The topology provides two distinct paths between s_1 and s_4 : one via s_2 with lower bandwidth links (1 Mbps), and another via s_3 with higher bandwidth links (10 Mbps). To simulate QoS behavior, we created traffic queues on s_3 using Mininet, assigning 6 Mbps (Queue 0) and 4 Mbps (Queue 1) to two different forwarding paths. This topology was implemented separately for both ODL and ONOS SDN controllers, and used throughout our experiments to evaluate intent translation, conflict detection, QoS policy enforcement and assurance.

C. LLM HYPERPARAMETER

The LLMs were downloaded from OLLama [55] for experiments. For intent translation tasks, the temperature of LLM was set to 0.6 and top-p was set to 0.3, while for conflict detection tasks, the temperature was set to 0.3 and top-p to 0.5. These settings were chosen to balance accuracy and variability, but they remain flexible and adjustable for future experimentation or deployment needs.

D. LLM BENCHMARKING RESULTS FOR INTENT TRANSLATION

Here we present the result of natural language intent translation using the LLMs listed in Table 3. First we report the accuracy of translation for a dataset, then we sort out the best performing LLMs based on accuracy.

Formal Specification Dataset: We start with the Formal specification dataset. We mentioned earlier that this is a dataset of natural language intent and JSON formatted translation pair for intent covering three different requirements: reachability, waypoints and loadbalancing. To check accuracy of translation we used the verification tool provided by [25] to validate that the LLMs output JSON structure conforms to the testset data structure. Here the expected JSON is compared against the LLM-generated output JSON. The comparison is performed field by field and value by value. Each mismatch or absence of a field or value in the output is considered one mistake. The overall accuracy is calculated across all test cases, taking into account the total number of expected fields and values. Table 10 shows the accuracy and average run time of 32 LLM. We did the test for all 33 LLMs but the LLM Deepseek-coder-v2:16b failed to produce meaningful result as it does not support “K-shift” which is required for precessing long prompts.

Highlights: Large models like QwQ-fusion and Command-r achieve high accuracy ($\geq 99\%$) on the Formal specification dataset, while mid-sized models such as Codellama:7b and Mistral:7b perform strongly (up to 95%) with lower latency, especially when supported by in-context examples. The dataset’s structured, schema-driven format and tightly scoped prompt favor models trained for code or structured output; smaller models often fail to maintain JSON correctness or field alignment.

Table 10 showcases trends in runtime, accuracy, and the impact of context examples. Larger models, such as QwQ (32b), QwQ-abilitated (32b), and Command-r (35b), consistently achieve the highest accuracy, exceeding 99% in some cases, due to their extensive parameter capacity, enabling them to better generalize and handle complex patterns in the dataset. However, these models also demonstrate higher runtime, highlighting a trade-off between accuracy and computational efficiency. Smaller models, like TinyLlama (1.1b) and Deepseek-coder (1.3b), show significantly lower accuracy, especially with fewer context examples, due to limited capacity for nuanced understanding. The inclusion of context examples plays a pivotal role in improving accuracy across all models, with gains most pronounced in mid-sized models such as Codellama:7b and Mistral:7b, where accuracy increases by up to 20% from zero to nine context examples. This indicates that semantically relevant and diverse examples help LLMs refine predictions and minimize errors for intent translation task. Interestingly, some models, like Llama3.1 (8b), exhibit a sharp improvement with context examples, suggesting better optimization for in-context learning. However, the diminishing returns in accuracy observed for larger models like QwQ at higher context sizes may reflect saturation in learning capacity or increased task complexity due to larger input contexts. This

TABLE 10: Benchmarking of LLMs for intent translation for Formal specification dataset (**Ctx n** denotes n context examples)

Sl.	LLM	Ctx 0		Ctx 1		Ctx 3		Ctx 6		Ctx 9	
		Acc.	Time								
1	Codegemma:7b	71.00	1.60	75.33	2.20	78.70	2.50	76.29	3.50	68.91	4.30
2	Codellama:7b	75.22	1.40	81.27	1.30	83.63	1.50	82.34	2	81.57	2.30
3	Codellama:34b	89.80	5.60	88.21	5.30	91.72	5.80	91.87	7.20	92.69	8
4	Codestral:22b	87.14	3.7	98.52	4.3	99.23	4	98.81	5.1	95.02	5.9
5	Command-r:35b	96.84	4.7	96.97	4.2	98.13	5.3	97.52	6.5	97.17	7.4
6	Deepseek-coder:1.3b	1.67	0.4	26.43	0.5	39.36	0.6	40.89	0.8	42.5	1
7	Dolphin-Mistral:7b	83.93	1.2	92.64	1.3	93.51	1.5	92.24	1.9	91.89	2.2
8	Gemma2:27b	98.01	4.9	97.42	4.8	98.28	4.4	98.33	5.3	98.23	6.1
9	Llama2:7b	24.33	3.6	23.46	1.9	38.63	1.9	41.91	2.1	53.92	3
10	Llama3:8b	54.2	1.2	88.96	1.7	90.4	1.5	94.13	1.8	94.88	2
11	Llama3.1:8b	56.59	1.4	90.57	1.8	95.65	1.6	95.78	1.9	95.47	2.1
12	Llama3.2:3b	84.29	1	88.41	0.9	90.43	0.9	88.81	1.1	89.2	1.2
13	Llava-Llama3:8b	22.47	1.3	66.46	1.4	78.22	1.5	82.62	1.9	81.65	2.1
14	Marco-o1:7b	91.66	1.3	95.27	1.5	96.87	1.7	96.87	2	96.72	2.2
15	Mistral:7b	80.05	1.3	93.61	1.4	94.05	1.6	93.78	2	94.15	2.3
16	Mistral-nemo:12b	91.06	1.8	93.89	2.5	95.99	2.3	95.6	2.8	94.55	3.2
17	Openchat:7b	71.09	1	91.82	1.4	93.68	1.6	93.53	1.9	93.08	2.2
18	Orca-mini:3b	21.73	1.1	47.32	0.8	56.83	1.1	52.68	1.7	41.89	1.5
19	Phi:7b	34.92	0.8	50.96	0.9	54.2	0.9	60.36	0.9	36.68	1.3
20	Phi3:8b	65.42	1.1	67.78	1	70.88	1.1	20.11	1.2	35.38	1.3
21	Qwen:4b	0.86	6.7	21.79	1.5	22.29	2.3	10.92	3.4	7.44	3.4
22	Qwen2:7b	87.43	1.2	93.33	1.3	94.5	1.5	94.83	1.8	91.24	2
23	Qwen2.5:7b	88.46	1.2	94.78	1.4	95.97	1.6	95.97	1.9	96.94	2.2
24	QwQ:32b	99.43	5.4	99.25	4.8	99.38	5.4	98.03	6.5	98.76	7.5
25	QwQ-abliterated:32b	99.65	5.4	98.98	4.8	99	5.5	98.48	6.5	99.05	7.5
26	QwQ-fusion:32b	99.58	4.9	98.42	4.8	99.43	5.5	98.93	6.6	99.13	7.5
27	Starcoder:3b	0	1.3	0	1.3	0	1.2	0	1.3	0	1.3
28	Starcoder2:3b	0	1	0	1.1	0	1	0	1.2	0	1
29	TinyLlama:1.1b	3.11	0.8	28.24	0.6	35.8	0.7	42.88	1.1	41.6	1.7
30	Wizardlm2:7b	88.55	1.5	92.91	1.5	94.3	1.7	93.99	2	92.86	2.3
31	Yi:6b	55.8	1.7	83.4	1.6	86.89	1.7	79.91	2	58.88	2.3
32	Zephyr:7b	52.11	1.2	85.07	1.4	87.59	1.8	89.76	2.2	90.08	2.5

TABLE 11: Best LLMs for intent translation for Formal specification dataset

Context	LLM	Accuracy (%)	Avg. Time (s)
0	QwQ-abliterated:32b	99.65	5.4
	QwQ-fusion:32b	99.58	4.9
	QwQ:32b	99.43	5.4
1	QwQ:32b	99.25	4.8
	QwQ-abliterated:32b	98.98	4.8
	Codestral:22b	98.52	4.3
3	QwQ-fusion:32b	99.43	5.5
	QwQ:32b	99.38	5.4
	Codestral:22b	99.23	4.0
6	QwQ-fusion:32b	98.93	6.6
	Codestral:22b	98.81	5.1
	QwQ-abliterated:32b	98.48	6.5
9	QwQ-fusion:32b	99.13	7.5
	QwQ-abliterated:32b	99.05	7.5
	QwQ:32b	98.76	7.5

result underscores the importance of balancing model size, runtime, and the careful selection of context examples to optimize both accuracy and efficiency in practical scenarios. Larger models excel in complex tasks, but the improvement in smaller models with in-context learning highlights the potential for tailored optimizations in resource-constrained settings. In Table 11 we present the best LLMs in terms of accuracy for Formal specification Dataset with respect to different context examples.

NFV Configuration Dataset: Now we report the result of running the LLMs on NFV configuration dataset.

Highlights: In the NFV configuration translation task, context examples notably boost accuracy, and several smaller and mid-sized models (e.g., Qwen2:7b, Codellama:7b) reach very high accuracy with low latency. This shows that the rigid, narrowly scoped NFV intents allow efficient models to compete, underscoring the roles of task simplicity, prompt design, and in-context learning.

The dataset has 120 natural language intent and JSON structured translation pairs. As before, half of it was chosen for providing context examples and other half was used for testing. Table 12 shows the accuracy and average run time of 33 LLMs. The accuracy was determined the same as was used for formal specification dataset. The table indicate that larger models, such as Command-r:35b and Codellama:34b, demonstrate high accuracy (up to 100%) with increased context examples, leveraging their greater parameter capacity to handle the complexity of NFV configurations. However, their higher accuracy comes at the cost of increased runtime, reflecting the trade-off between model sophistication and computational efficiency. Conversely, smaller models like TinyLlama:1.1b and Orca-mini:3b struggle with accuracy across all contexts, indicating limited generalization capabilities due to their lower parameter sizes.

The inclusion of context examples significantly boosts performance for mid-range models like Codellama:7b and Dolphin-Mistral:7b, where accuracy improves up to 30%

TABLE 12: Benchmarking of LLMs for intent translation for NFV configuration dataset (Ctx n denotes n context examples)

Sl.	LLM	Ctx 0		Ctx 1		Ctx 3		Ctx 6		Ctx 9	
		Acc.	Time	Acc.	Time	Acc.	Time	Acc.	Time	Acc.	Time
1	Codegemma:7b	75.00	1.30	77.50	1.20	75.00	1.10	90.00	1.20	90.00	1.20
2	CodeLlama:34b	17.50	3.40	82.50	3.30	85.00	3.50	82.50	4.00	87.50	4.20
3	CodeLlama:7b	45.00	1.00	67.50	1.00	70.00	1.00	80.00	1.00	95.00	1.10
4	Codestral:22b	72.50	2.80	72.50	2.20	82.50	2.30	87.50	2.60	95.00	2.90
5	Command-r:35b	65.00	3.20	90.00	3.20	87.50	3.50	97.50	3.70	100.00	4.00
6	Deepseek-coder-v2:16b	60.00	1.30	85.00	1.00	80.00	1.10	92.50	1.30	95.00	1.40
7	Deepseek-coder:1.3b	0.00	1.00	13.51	0.60	47.06	0.60	37.14	0.60	36.84	0.50
8	Dolphin-Mistral:7b	40.00	1.30	75.00	1.00	62.50	1.00	92.50	1.00	92.50	1.10
9	Gemma2:27b	65.00	3.20	72.50	2.30	75.00	2.20	82.50	2.60	92.50	2.90
10	Llama2:7b	5.00	2.20	62.50	0.90	58.97	1.00	63.89	1.10	60.00	1.40
11	Llama3:8b	57.50	1.00	75.00	0.90	85.00	0.90	95.00	1.00	97.50	1.10
12	Llama3.1:8b	47.50	1.10	77.50	1.00	75.00	0.90	92.50	1.10	97.50	1.20
13	Llama3.2:3b	15.00	1.50	50.00	0.70	70.00	0.70	72.50	0.60	75.00	0.70
14	Llava-Llama3:8b	0.00	1.60	38.46	1.00	52.78	0.90	51.28	1.10	47.37	1.20
15	Marco-o1:7b	12.50	1.00	75.00	1.00	77.50	0.90	92.50	1.00	90.00	1.00
16	Mistral-nemo:12b	50.00	1.30	57.50	1.10	67.50	1.10	90.00	1.40	97.50	1.50
17	Mistral:7b	55.00	1.30	57.50	0.80	55.00	0.80	70.00	0.90	77.50	1.10
18	Openchat:7b	15.38	1.30	65.00	0.80	67.50	0.80	82.50	1.00	85.00	1.00
19	Orca-mini:3b	0.00	1.70	27.50	1.10	15.38	0.90	14.71	0.90	0.00	1.00
20	Phi:7b	2.63	0.90	10.26	0.60	27.50	0.70	43.59	0.80	37.50	0.70
21	Phi3:8b	0.00	1.40	29.73	0.90	43.24	0.80	47.37	0.80	52.63	0.90
22	Qwen:4b	0.00	1.40	17.50	0.90	23.68	1.00	12.82	1.00	7.89	1.30
23	Qwen2:7b	70.00	1.00	77.50	0.80	75.00	0.80	92.50	0.90	100.00	1.00
24	Qwen2.5:7b	22.50	0.90	70.00	0.90	77.50	0.90	90.00	1.00	95.00	1.10
25	QwQ-abliterated:32b	62.50	3.10	75.00	3.00	82.50	2.90	82.50	3.00	90.00	3.40
26	QwQ-fusion:32b	57.50	3.00	75.00	3.00	77.50	2.90	87.50	3.10	90.00	3.40
27	QwQ:32b	60.00	3.10	75.00	3.10	82.50	3.10	85.00	3.10	87.50	3.40
28	Starcoder:3b	0.00	1.50	0.00	1.50	0.00	1.70	0.00	1.40	0.00	1.10
29	Starcoder2:3b	0.00	0.40	0.00	0.40	0.00	0.40	0.00	0.50	0.00	0.40
30	TinyLlama:1.1b	0.00	0.70	17.95	0.60	15.00	0.40	20.51	0.50	17.50	0.50
31	Wizardlm2:7b	42.11	1.50	70.00	0.90	70.00	0.90	75.00	1.00	85.00	1.10
32	Yi:6b	17.50	1.20	60.00	1.00	60.00	1.10	68.42	1.00	66.67	1.10
33	Zephyr:7b	37.50	1.40	65.00	1.00	65.00	1.00	77.50	1.00	87.18	1.10

TABLE 13: Best LLMs for intent translation for NFV configuration dataset

Context	LLM	Accuracy (%)	Avg. Time (s)
0	Codegemma:7b	75.0	1.3
1	Command-r:35b	90.0	3.2
3	Command-r:35b	87.5	3.5
6	Command-r:35b	97.5	3.7
9	Command-r:35b	100.0	4.0
	Qwen2:7b	100.0	1.0

as context increases from 0 to 9 examples. This trend highlights the importance of providing relevant examples to guide LLMs in understanding intent-specific nuances, especially for models optimized for in-context learning. However, some models, such as QwQ-fusion:32b, exhibit diminishing accuracy gains beyond a certain number of examples, suggesting a saturation point in leveraging additional context. Notably, specialized models like Deepseek-coder-v2:16b perform competitively, achieving 95% accuracy with relatively low runtimes, showcasing the impact of domain-specific optimization. On the other hand, general-purpose models like Llama3.2:3b show steady but limited improvements, indicating a need for fine-tuning to handle domain-specific tasks effectively. In table 13 we present the best LLMs for NFV configuration translation with respect to different context example.

Intent2Flow-ODL Dataset: Now we present the outcome of benchmarking using the proposed Intent2Flow-ODL dataset. This dataset evaluates ODL flow rule translation. For

testing the LLMs translation capability, we used different natural language intents that use different network configuration tasks such as port based forwarding, firewall, flowspace slicing, IP based forwarding etc. Same as before, half of the dataset samples were used to give relevant examples to the LLM and the other half was used as test cases. We used three different values of context examples: 0, 1 and 3. Notably, on this dataset, we were able to get readable output from 30 LLMs out of targeted 33. Some LLMs failed to produce meaningful JSONs such as tinyLlama:1.1b, Orca-mini:3b and Deepseek-coder-v2:16b. The problem with Deepseek-coder-v2:16b was same as with Formal specification dataset, 'K-shift' not supported. As for tinyLlama:1.1b and Orca-mini:3b, the length of the designed prompt might cause the issue due to their context length limitation. We present the experimental results in Table 14.

The result is consistent with previous results from the Formal specification and NFV configuration datasets, larger models like QwQ-abliterated (32b) and QwQ-fusion (32b) achieve the highest accuracy (100% with three context examples) due to their ability to generalize complex patterns in the dataset. These models, however, exhibit significantly higher runtimes (over 6 seconds on average), reflecting the computational demands of their large parameter size. Models such as Command-r (35b) and Gemma2 (27b) also perform well, demonstrating accuracy improvements of up to 70% when transitioning from zero to three context examples, highlighting the importance of providing relevant examples to enhance in-context learning.

TABLE 14: Benchmarking of LLMs for intent translation for Intent2Flow-ODL dataset (**Ctx n** denotes n context examples)

Sl.	LLM	Ctx 0		Ctx 1		Ctx 3	
		Acc.	Time	Acc.	Time	Acc.	Time
1	Codegemma:7b	0.00	2.42	88.89	3.11	77.78	3.20
2	Codellama:34b	38.89	7.88	77.78	7.99	77.78	8.64
3	Codellama:7b	27.78	2.39	77.78	2.23	83.33	2.40
4	Codestral:22b	66.67	7.43	88.89	6.14	94.44	6.72
5	Command-r:35b	27.78	6.99	88.89	7.20	100.00	7.50
6	Deepseek-coder:1..3b	0.00	0.80	0.00	1.30	0.00	1.03
7	Dolphin-Mistral:7b	0.00	2.67	55.56	2.53	77.78	2.24
8	Gemma2:27b	50.00	6.04	88.89	6.01	100.00	5.66
9	Llama2:7b	0.00	2.80	38.89	2.76	22.22	3.12
10	Llama3:8b	33.33	2.54	94.44	2.18	83.33	1.98
11	Llama3.1:8b	22.22	2.81	55.56	2.95	88.89	2.17
12	Llama3.2:3b	0.00	1.51	50.00	1.17	72.22	1.30
13	llava-Llama3:8b	0.00	1.85	11.11	2.06	33.33	1.99
14	Marco-01:7b	33.33	2.29	72.22	2.55	61.11	2.77
15	Mistral-nemo:12b	50.00	3.11	55.56	2.08	77.78	2.42
16	Mistral:7b	0.00	2.84	66.67	2.01	72.22	2.05
17	Openchat:7b	0.00	2.12	72.22	1.82	83.33	1.96
18	Phi:7b	0.00	0.98	0.00	1.59	0.00	0.51
19	Phi3:8b	0.00	1.54	5.56	2.10	0.00	1.98
20	Qwen:4b	0.00	0.81	0.00	1.30	0.00	3.15
21	Qwen2:7b	38.89	2.25	61.11	1.76	83.33	1.90
22	Qwen2.5:7b	44.44	2.23	83.33	2.11	72.22	2.73
23	QwQ-abliterated:32b	83.33	8.29	94.44	6.51	100.00	6.31
24	QwQ-fusion:32b	72.22	8.33	94.44	6.97	100.00	6.60
25	QwQ:32b	83.33	8.26	94.44	6.98	100.00	6.25
26	Starcoder:3b	0.00	0.34	0.00	0.44	0.00	0.44
27	Starcoder2:3b	0.00	0.35	0.00	0.43	0.00	0.47
28	Wizardlm2:7b	0.00	2.67	61.11	1.87	77.78	2.09
29	Yi:6b	11.11	2.49	61.11	2.48	61.11	2.66
30	Zephyr:7b	0.00	2.37	50.00	2.75	55.56	2.93

TABLE 15: Best LLMs for intent translation for Intent2Flow-ODL dataset

Context	LLM	Accuracy (%)	Avg. Time (s)
0	QwQ-abliterated:32b	83.33	8.29
	QwQ:32b		8.26
	Llama3:8b	2.18	
1	QwQ-abliterated:32b	94.44	6.51
	QwQ-fusion:32b	6.97	
	QwQ:32b	6.98	
	Command-r:35b	7.50	
3	Gemma2:27b	5.66	
	QwQ-abliterated:32b	100.00	6.31
	QwQ-fusion:32b	6.60	
	QwQ:32b	6.25	

Mid-range models like Codegemma:7b and Codellama:7b show moderate improvements with added context but generally struggle to match the precision of larger models. Interestingly, Codellama:34b, despite its large size, exhibits less pronounced gains compared to QwQ-family models, suggesting that model specialization, in addition to parameter size, plays a critical role in handling domain-specific tasks like ODL flow rules. Consistent with observations from previous datasets, larger models are well-suited for complex intent translation tasks, but their high runtime makes them resource-intensive. The role of context examples is pivotal across all datasets, with significant accuracy improvements observed as examples increase, particularly for mid-sized models. In table 15 we present the best LLMs for ODL flow rule translation with respect to different context example.

Highlights: On the Intent2Flow-ODL dataset, models like QwQ-abliterated and Command-r achieve very high accuracy with just three examples. Notably, the structured, rule-based format—enhanced by prompt decomposition (especially for QoS intents)—allows even mid-sized models like Codellama:7b and Llama3:8b to make substantial gains. This demonstrates that modular, intent-specific prompts can enable accurate SDN rule synthesis.

Intent2Flow-ONOS Dataset: Now we present the outcome of benchmarking using the proposed Intent2Flow-ONOS dataset. We developed this dataset to evaluate LLMs on ONOS flow rule translation task. For testing the LLMs translation capability, we used diverse natural language intents that use different network configuration tasks such as port/IP based forwarding, blocking, flowspace slicing for QoS etc. Half of the dataset samples were used to give relevant examples to the LLM and the other half was used as test cases. We used five different values of context examples: 0, 1, 3, 6 and 9. Similar to Intent2Flow-ODL dataset, we were able to get readable output from 30 LLMs out of targeted 33. TinyLlama:1.1b, Orca-mini:3b and Deepseek-coder-v2:16b failed to produce meaningful JSON structures. We present the experimental results in Table 16.

Table 16 reveals some clear trends. Several models, notably Codestral:22b, Command-r:35b, and various QwQ models (32b), achieved high accuracy, often exceeding 90% and even reaching 100% on some contexts. This suggests that for this specific task, certain architectures and training regimens are more effective. Interestingly, parameter size doesn't seem to be the sole determinant of success. While some larger models like Command-r:35b performed reasonably well, they were outperformed by smaller models like Codestral:22b. This indicates that models specializing in code generation and structured output translation, like those found in the “code” prefixed models (Codegemma, Codellama, Codestral), play a crucial role. The trade-off between accuracy and inference time was also evident, with larger models taking significantly longer—Codellama-34b, for instance, required over 10 seconds per inference, whereas Qwen2.5-7B achieved competitive performance in under 2 seconds. Conversely, models like Deepseek-coder, Phi, Qwen, and Starcoder variants struggled, often achieving 0% accuracy, suggesting they might not be suitable for this type of translation task, perhaps being geared towards different NLP applications. The Llama family shows varied results, with some larger variants performing better than smaller ones, but still not reaching the top performers' level. This highlights the importance of not just scale, but also the data and training methodology. Furthermore, context sensitivity is also important as most models improved with more added examples. For instance, Dolphin-Mistral:7b jumped from 0% to 80%. Using 6–9 context examples is beneficial, as most models reach peak performance within this range.

TABLE 16: Benchmarking of LLMs for intent translation for Intent2Flow-ONOS dataset (Ctx n denotes n context examples)

Sl.	LLM	Ctx 0		Ctx 1		Ctx 3		Ctx 6		Ctx 9	
		Acc.	Time								
1	Codegemma:7b	36	2.25	84	2.71	92	2.74	92	2.56	88	2.66
2	Codellama:34b	48	6.92	72	7.03	72	8.09	72	9.26	64	10.26
3	Codellama:7b	40	1.95	68	1.59	80	1.68	80	1.99	88	2.22
4	Codestral:22b	96	5.93	96	3.59	100	4.15	100	4.57	100	5.14
5	Command-r:35b	92	5.77	92	6.39	96	6.68	100	7.13	96	7.68
6	Deepseek-coder:1.3b	0	1.25	0	1.46	0	1.84	0	1.66	0	1.89
7	Dolphin-Mistral:7b	0	2.36	40	2.48	60	2.98	68	3.17	80	2.82
8	Gemma2:27b	88	6.06	92	3.95	88	4.08	92	4.49	92	5.00
9	Llama2:7b	4	2.51	32	2.45	20	2.88	8	3.20	4	3.40
10	Llama3.1:8b	0	2.06	64	2.10	72	2.11	88	1.73	84	2.01
11	Llama3.2:3b	24	1.24	40	0.94	52	1.14	60	0.95	64	1.05
12	Llama3:8b	56	2.06	60	1.77	76	1.33	80	1.51	80	1.54
13	llava-Llama3:8b	16	2.02	48	1.49	56	1.41	64	2.01	60	1.75
14	Marco-o1:7b	68	1.80	88	2.09	92	2.20	84	1.84	80	1.98
15	Mistral:7b	44	2.34	20	1.57	44	1.60	72	1.82	68	1.93
16	Mistral-nemo:12b	80	2.65	88	2.60	92	2.59	96	2.74	92	3.32
17	Openchat:7b	24	2.02	72	2.76	72	3.40	80	2.11	68	1.91
18	Phi:7b	0	0.89	20	1.19	16	1.10	12	2.69	0	2.06
19	Phi3:8b	0	1.69	4	1.61	4	1.86	16	1.79	20	1.93
20	Qwen:4b	0	1.72	8	1.97	0	1.39	0	2.90	0	4.51
21	Qwen2.5	68	1.86	88	1.80	84	1.56	84	1.59	88	1.68
22	Qwen2:7b	40	1.66	64	1.60	80	1.33	88	1.46	88	1.60
23	QwQ:32b	88	6.80	80	7.04	92	5.92	96	5.74	96	6.06
24	QwQ-abliterated:32b	80	6.82	88	5.79	96	4.84	100	5.21	96	5.77
25	QwQ-fusion:32b	80	6.61	84	6.51	80	5.32	96	5.61	96	6.12
26	Starcoder:3b	0	0.64	0	1.19	0	0.90	0	1.24	0	0.67
27	Starcoder2:3b	0	0.45	0	0.45	0	0.65	0	0.70	0	0.63
28	Wizardlm2:7b	36	2.48	44	2.49	52	2.23	52	2.77	60	3.22
29	Yi:6b	36	2.20	40	2.00	52	1.87	48	2.08	64	2.46
30	Zephyr:7b	20	2.13	60	2.30	44	2.52	56	2.83	60	3.10

Highlights: In the ONOS flow rule translation task, structured prompts matching the ONOS JSON schema separated queue/VLAN intents from forwarding/blocking rules. Combined with rigid SDN-specific intents, this enabled Codestral:22b to achieve very high accuracy with low latency, outperforming larger LLMs. Specialized prompts, task regularity, and 6–9 in-context examples boosted performance, especially for mid-sized models like Qwen2.5:7b.

Given these findings, for ONOS intent translation, models like Codestral:22b, Command-r:35b, and the QwQ family appear to be the most promising. For other SDN controllers, the lessons learned should be similar: prioritize models specialized in code generation or translation and don't solely rely on parameter count. Benchmarking with representative intent examples is crucial for selecting the right LLM. Further investigation into the training data and architecture of high-performing models would be beneficial for developing even more effective solutions for intent translation in SDN controllers. Moreover, fine-tuning mid-sized model (e.g., Mistral-Nemo-12b or Qwen2.5-7b) could be a more practical choice, offering a trade off between speed and accuracy. In table 17 we present the best LLMs for ONOS flow rule translation with respect to different context example.

Evaluation of 70 Billion Parameter LLMs: Our main benchmarking focused on small to mid-sized LLMs, leaving out models at the extreme end of the parameter scale. To understand how very large models perform on the same task,

TABLE 17: Best LLMs for intent translation for Intent2Flow-ONOS dataset

Context	LLM	Accuracy (%)	Avg. Time (s)
0	Codestral:22b	96	5.93
1	Codestral:22b	96	3.59
3	Codestral:22b	100	4.15
6	Codestral:22b		4.57
	Command-r:35b	100	7.13
	QwQ-abliterated:32b		5.21
9	Codestral:22b	100	5.14

we evaluated three representative 70-billion-parameter LLMs on the Intent2Flow-ONOS dataset. Their results, shown in Table 18, provide insight into the capabilities and limitations of scaling up model size for SDN intent translation.

The evaluation reveals that larger model size does not inherently guarantee superior performance in translating high-level network intents into ONOS-compatible flow rules. Notably, Llama3.3:70b achieved an 88% accuracy with zero-shot prompts, surpassing both Llama2:70b and Codellama:70b. We could not collect results for larger context examples for Llama 3.3 due to hardware memory constraints. Codellama:70b and Llama2:70b showed only modest accuracy gains despite increasing context, and their overall performance lagged behind smaller, code-specialized models like Codestral:22b. This limited improvement may stem from a mismatch between the Intent2Flow-ONOS dataset (which is structurally rigid and highly schema-driven) and the broader training objectives of general-purpose 70b models. Additionally, the multi-prompt strategy used in our bench-

TABLE 18: Benchmarking of 70 billion parameter LLMs for intent translation for Intent2Flow-ONOS dataset (**Ctx n** denotes n context examples)

Sl.	LLM	Ctx 0		Ctx 1		Ctx 3		Ctx 6		Ctx 9	
		Acc.	Time								
1	Codellama:70b	60	14.73	48	10.45	60	11.89	72	12.37	68	15.50
2	Llama2:70b	8	16.26	64	14.08	72	14.89	76	15.77	76	17.18
3	Llama3.3:70b	88	10.43								

marking was tuned toward compact, code-oriented models; larger models not specifically fine-tuned for structured translation tasks may struggle to align with such narrow formats. These discrepancies suggest that the effectiveness of large models is heavily influenced by prompt design and the nature of the dataset.

Highlights: Despite their size, 70B models like Codellama and Llama2 delivered limited gains on the Intent2Flow-ONOS task, likely due to a mismatch between their general-purpose training and the dataset’s rigid, schema-driven structure—emphasizing that effective intent translation depends more on prompt-task alignment than on model scale alone.

Moreover, this underscores the importance of tailoring prompts to the specific capabilities of the model and the characteristics of the dataset. In contrast, smaller models have demonstrated more consistent and reliable performance which highlights that model size should be considered alongside other factors such as prompt engineering and dataset alignment when evaluating LLMs for IBN tasks.

Summary of Benchmarking for Intent Translation: The benchmarking of LLMs across four datasets—Formal Specification, NFV Configuration, Intent2Flow-ODL, and Intent2Flow-ONOS—reveals key trends in performance, efficiency, and generalization. Larger models such as the QwQ family (32b) and Command-r (35b) consistently achieved the highest accuracy (often $\geq 99\%$), particularly for structurally rigid tasks, though at the cost of higher runtimes. Mid-sized models like Codellama:7b and Dolphin-Mistral:7b showed substantial accuracy gains (up to 30%) with more context examples, highlighting the strong impact of in-context learning when paired with well-designed prompts. Smaller models (e.g., TinyLlama:1.1b, Orca-mini:3b) struggled to generalize, often failing to produce syntactically valid outputs due to parameter and context limitations. Interestingly, the evaluation of 70B parameter LLMs (e.g., Llama2, Llama3.3, and Codellama:70b) showed that size alone does not ensure superiority: Codellama and Llama2 offered only marginal improvements over mid-sized models, and at significantly higher computational cost. Llama3.3:70b performed better in zero-shot settings, but further results were limited by hardware constraints. These findings suggest that beyond scale, alignment between the model’s training, prompt structure,

and the schema-specific nature of intent translation tasks is critical for reliable performance. Moreover, diminishing gains for some large models (e.g., QwQ-fusion) with more context imply a saturation effect, reinforcing the importance of task-specialized tuning over raw parameter count.

Notably, benchmarking results show that LLMs generally perform better on ONOS than on ODL in zero-shot (Ctx 0) scenarios, with most models achieving higher accuracy without context examples. However, as more context is provided (Ctx 1 and Ctx 3), this advantage shifts, and a greater number of models achieve higher accuracy on ODL. This is likely because ONOS’s schema is simpler and more closely aligned with common pretraining data, enabling better zero-shot performance, while in-context examples help LLMs adapt to ODL’s more complex structure and narrow the performance gap.

E. LLM BENCHMARKING RESULTS FOR CONFLICT DETECTION

To find how LLMs perform on conflict detection task related to network configuration, we benchmarked them on the proposed FlowConflict-ODL and FlowConflict-ONOS datasets.

Conflict Detection using FlowConflict-ODL Datasets: We now present the results of benchmarking the LLMs for the conflict detection task using the FlowConflict-ODL dataset. In Sec. IV-B, we mentioned how the dataset is prepared. In total, each LLM evaluated 50 pairs of flow rules. Among these, 46 rule pairs were non-conflicting, while 4 were conflicting. Therefore, the ideal outcomes are 4 true positives (TP) and 46 true negatives (TN). Table 19 presents the TP, TN, false positives (FP), and false negatives (FN) for 28 LLMs. We excluded 5 models (“Phi”, “Orca-mini”, “Qwen”, “tinyLlama”, “Deepseek-coder-V2”) from the results due to their inability to generate meaningful responses.

The results reveal significant variations in accuracy, precision, and error handling across models. Models such as QwQ-abliterated:32b, QwQ-fusion:32b demonstrated exemplary performance, achieving near-perfect results (TP = 4, FP around 7, TN around 39) with minimal FP or FN. QwQ fell slightly behind by achieving TP = 3. These results are consistent with the earlier findings, where the QwQ-family models consistently excelled in both natural language and JSON-based conflict detection tasks, indicating their strong generalization and task-specific capabilities. Anomalies arise with models like Mistral:7b, Wizardlm2:7b, and Yi:6b, which produced perfect True Positives (TP = 4) but

TABLE 19: Benchmarking of LLMs for conflict detection using FlowConflict-ODL dataset

Sl.	Model	TP	FP	FN	TN	Accuracy	Precision	Recall	F1-Score	FPR	Avg. Time
1	Marco-o1:7b	1	12	3	34	0.70	0.08	0.25	0.12	0.26	4.19
2	Mistral:7b	4	46	0	0	0.08	0.08	1.00	0.15	1.00	4.05
3	Mistral-nemo:12b	4	39	0	7	0.22	0.09	1.00	0.17	0.85	5.61
4	Deepseek-coder:1.3b	3	43	1	3	0.12	0.07	0.75	0.12	0.93	2.58
5	Starcoder:3b	0	0	4	46	0.92	0.00	0.00	0.00	0.00	3.85
6	Codegemma:7b	4	22	0	24	0.56	0.15	1.00	0.27	0.48	4.09
7	Starcoder2:3b	0	0	4	46	0.92	0.00	0.00	0.00	0.00	3.18
8	Openchat:7b	3	21	1	25	0.56	0.12	0.75	0.21	0.46	3.48
9	Phi3:8b	2	25	2	21	0.46	0.07	0.50	0.13	0.54	2.63
10	Dolphin-Mistral:7b	2	19	2	27	0.58	0.10	0.50	0.16	0.41	3.39
11	Wizardlm2:7b	4	46	0	0	0.08	0.08	1.00	0.15	1.00	4.09
12	Yi:6b	4	46	0	0	0.08	0.08	1.00	0.15	1.00	3.61
13	Zephyr:7b	2	26	2	20	0.44	0.07	0.50	0.12	0.57	3.70
14	Command-r:35b	2	11	2	35	0.74	0.15	0.50	0.24	0.24	9.08
15	llava-Llama3:8b	1	16	3	30	0.62	0.06	0.25	0.10	0.35	4.71
16	Codestral:22b	2	8	2	38	0.80	0.20	0.50	0.29	0.17	7.14
17	Codellama:34b	3	29	1	17	0.40	0.09	0.75	0.17	0.63	3.87
18	Codellama:7b	4	45	0	1	0.10	0.08	1.00	0.15	0.98	3.87
19	Llama2:7b	2	30	2	16	0.36	0.06	0.50	0.11	0.65	3.69
20	Llama3:8b	4	36	0	10	0.28	0.10	1.00	0.18	0.78	4.27
21	Llama3.1:8b	2	22	2	24	0.52	0.08	0.50	0.14	0.48	4.25
22	Llama3.2:3b	2	26	2	20	0.44	0.07	0.50	0.12	0.57	3.27
23	Owen2:7b	2	17	2	29	0.62	0.11	0.50	0.17	0.37	3.52
24	Owen2.5:7b	0	3	4	43	0.86	0.00	0.00	0.00	0.07	3.59
25	Gemma2:27b	2	7	2	39	0.82	0.22	0.50	0.31	0.15	7.72
26	QwQ-abliterated:32b	4	4	0	42	0.92	0.50	1.00	0.67	0.09	9.23
27	QwQ-fusion:32b	4	7	0	39	0.86	0.36	1.00	0.53	0.15	9.59
28	QwQ:32b	3	4	1	42	0.90	0.43	0.75	0.55	0.09	9.42

had excessively high false positives ($FP = 46$), suggesting that these models lacked the ability to correctly differentiate conflicting from non-conflicting cases. This pattern is inconsistent with earlier evaluations, where Mistral:7b and Wizardlm2:7b performed moderately well, indicating sensitivity to the input format (natural language vs. JSON). Conversely, models such as Starcoder:3b and Starcoder2:3b failed entirely ($TP = 0$, $FN = 4$), reflecting their inability to interpret the conflict detection task in the structured JSON format, consistent with their poor performance in earlier JSON-related evaluations. Larger models like Codestral:22b and Gemma2:27b displayed mixed results, with moderate TPs (2) but noticeable FPs (7–8). This suggests that while they can identify conflicts to some extent, they are prone to over-flagging non-conflicting cases. These results align with prior observations, where these models showed decent, but not exceptional, performance in JSON-based tasks.

The performance of Codellama:34b is notably underwhelming, especially given its large parameter size and computational demands. With a True Positive (TP) count of 3, False Positives (FP) of 29, and a True Negative (TN) of only 17, it performed worse than similar and mid-sized models like Codestral:22b, Codegemma:7b and Command-r:35b. This discrepancy indicates that Codellama:34b, despite its size and potential for handling complex tasks, struggles to balance sensitivity and precision in JSON-based conflict detection. A likely reason for this underperformance is the model's inability to effectively handle structured data, as seen in earlier results where it also lagged behind in JSON-based tasks. This could stem from inadequate pretraining or fine-tuning on tasks involving structured formats like JSON. Furthermore, its high FP count suggests that it frequently over-flagged non-conflicting cases as conflicts, reflecting poor understanding of the nuanced prompt instructions.

Highlights: In the FlowConflict-ODL conflict detection task, a strict field-by-field comparison prompt led to varied LLM performance. Only QwQ-abliterated and Gemma2:27b accurately detected direct conflicts with minimal false positives, while others over-predicted due to partial match misinterpretation or missing-field semantics errors. Reliable detection requires precise structural reasoning aligned with literal matching rules, achieved by few code-aligned or structurally sensitive models.

Conflict Detection using FlowConflict-ONOS Datasets.:

We now present the results of benchmarking the LLMs for the conflict detection task using the FlowConflict-ONOS dataset. In Sec. IV-B, we mentioned how the dataset is prepared. In total, each LLM evaluated 62 pairs of flow rules. Among these, 52 rules were non-conflicting, while 10 were conflicting. Therefore, the ideal outcomes are 10 true positives (TP) and 52 true negatives (TN). Table 20 presents the TP, TN, FP, and FN for 30 LLMs. We excluded 3 models (“Orca-mini”, “tinyLlama”, “Deepseek-coder-V2”) from the results due to their inability to generate meaningful responses.

The evaluation reveals significant performance variations among LLMs in detecting conflicting ONOS flow rules. QwQ-32b stands out as the only model achieving perfect accuracy (100%), with zero false positives or false negatives, making it the gold standard for this task. Close behind, QwQ-fusion-32b (98%) and QwQ-abliterated-32b (94%) also demonstrate exceptional performance, suggesting that the QwQ family is particularly well-suited for structured data conflict identification.

TABLE 20: Benchmarking of LLMs for conflict detection using FlowConflict-ONOS dataset

Sl.	Model	TP	FP	TN	FN	Accuracy	Precision	Recall	F1-Score	FPR	Avg. Time
1	Codegemma:7b	5	43	9	5	0.23	0.10	0.50	0.17	0.83	21.67
2	Codellama:34b	9	47	5	1	0.23	0.16	0.90	0.27	0.90	52.27
3	Codellama:7b	10	52	0	0	0.16	0.16	1.00	0.28	1.00	18.83
4	Codestral:22b	9	30	22	1	0.50	0.23	0.90	0.37	0.58	37.31
5	Command-r:35b	8	15	37	2	0.73	0.35	0.80	0.49	0.29	46.75
6	Deepseek-coder:1.3b	9	49	3	1	0.19	0.16	0.90	0.27	0.94	10.57
7	Dolphin-Mistral:7b	9	45	7	1	0.26	0.17	0.90	0.29	0.87	16.55
8	Gemma2:27b	8	3	49	2	0.92	0.73	0.80	0.76	0.06	38.19
9	Llama2:7b	10	52	0	0	0.16	0.16	1.00	0.28	1.00	17.72
10	Llama3.1:8b	3	22	30	7	0.53	0.12	0.30	0.17	0.42	20.38
11	Llama3.2:3b	5	26	26	5	0.50	0.16	0.50	0.24	0.50	16.50
12	Llama3:8b	6	36	16	4	0.35	0.14	0.60	0.23	0.69	20.95
13	llava-Llama3:8b	4	30	22	6	0.42	0.12	0.40	0.18	0.58	23.96
14	Marco-o1:7b	6	10	42	4	0.77	0.38	0.60	0.47	0.19	21.42
15	Mistral:7b	8	51	1	2	0.15	0.14	0.80	0.24	0.98	17.12
16	Mistral-nemo:12b	10	38	14	0	0.39	0.21	1.00	0.35	0.73	27.19
17	Openchat:7b	9	49	3	1	0.19	0.16	0.90	0.27	0.94	17.76
18	Phi:7b	7	43	9	3	0.26	0.14	0.70	0.23	0.83	11.87
19	Phi3:8b	9	48	4	1	0.21	0.16	0.90	0.27	0.92	13.25
20	Owen:4b	1	1	51	9	0.84	0.50	0.10	0.17	0.02	8.01
21	Owen2.5:7b	3	1	51	7	0.87	0.75	0.30	0.43	0.02	19.10
22	Owen2:7b	5	8	44	5	0.79	0.38	0.50	0.43	0.15	19.42
23	QwQ:32b	10	0	52	0	1.00	1.00	1.00	1.00	0.00	47.43
24	QwQ-abliterated:32b	9	3	49	1	0.94	0.75	0.90	0.82	0.06	47.73
25	QwQ-fusion:32b	10	1	51	0	0.98	0.91	1.00	0.95	0.02	47.53
26	Starcoder:3b	0	0	52	10	0.84	—	0.00	—	0.00	15.30
27	Starcoder2:3b	0	0	52	10	0.84	—	0.00	—	0.00	12.93
28	Wizardlm2:7b	8	51	1	2	0.15	0.14	0.80	0.24	0.98	20.16
29	Yi:6b	8	45	7	2	0.24	0.15	0.80	0.25	0.87	16.31
30	Zephyr:7b	9	35	17	1	0.42	0.20	0.90	0.33	0.67	18.45

Highlights: In the FlowConflict-ONOS conflict detection task, QwQ:32b uniquely achieved very high accuracy with no false positives. The strict, schema-based prompt required exact field-by-field flow rule comparison, revealing weaknesses in models using generalization or fuzzy matching. Precise conflict detection demands structural alignment between the LLM's behavior and the dataset's deterministic logic, which only highly structured-output models like QwQ:32b consistently achieved.

Llama3.3-70b (94%) and Gemma2-27b (92%) also perform well, but at significantly higher computational costs. Many models, especially some of the smaller ones or those not specifically designed for code-related tasks, struggled significantly. For instance, several models, including Llama2:70b and Llama2:7b, had a high recall (correctly identifying most actual conflicts) but also an extremely high FPR, meaning they flagged many non-conflicts as conflicts, rendering them practically unfit. Similarly, many models struggle with false positives, with Codellama:7b, Codegemma-7b, Dolphin-Mistral-7b, and Deepseek-coder-1.3b exceeding 87% false positive rate (FPR), making them unsuitable for practical use indicating limited contextual understanding of JSON flow rules. However, the Starcoder models, despite being code-focused, had very low recall. The same conclusion can be drawn here as was in intent translation- parameter size alone doesn't guarantee good performance, as some larger models performed poorly. The table reveals a clear speed-performance trade-off. While QwQ:32b delivers perfect performance, it's not the fastest model. Some smaller models, like those from the Phi family,

are quicker but have much lower accuracy. Given these findings, for conflict detection in ONOS flow rules, the QwQ family is the clear recommendation balancing accuracy and efficiency. Prioritizing models with high precision (to minimize false alarms) and balanced F1-scores is critical, though deployment should consider inference time as well. As for SDN controllers in general, the key lesson is to prioritize models that have demonstrated strong performance on similar rule-based tasks or code analysis. Careful evaluation with representative flow rules and consideration of the speed-performance trade-off are essential. A model with high recall but also a very high false positive rate is not practical.

F. EVALUATION OF 70 BILLION PARAMETER LLMs FOR CONFLICT DETECTION

To evaluate how very large language models perform in structural reasoning tasks, we benchmarked three representative 70-billion-parameter LLMs on the FlowConflict-ONOS dataset for conflict detection. As shown in Table 21, the results reveal a wide variance in effectiveness despite comparable scale. While Llama3.3:70b demonstrated strong overall performance—with 94% accuracy, low false positive rate (0.02), and the highest F1-score (0.78) among the three—Codellama:70b and Llama2:70b underperformed significantly. Llama2:70b, despite perfect recall, produced 52 false positives, suggesting it over-predicted conflicts and failed to adhere to the prompt's strict field-level comparison criteria. Codellama:70b performed more moderately but still struggled with both recall and precision. Several factors contribute to the underperformance of large LLMs in this context. Large LLMs, trained on extensive and diverse datasets, tend to rely on semantic reasoning and pattern recognition. This predisposition can lead to overgeneralization, causing the models to infer conflicts where none exist, thereby

TABLE 21: Benchmarking of 70 billion parameter LLMs for conflict detection using FlowConflict-ONOS dataset

Sl.	Model	TP	FP	TN	FN	Accuracy	Precision	Recall	F1-Score	FPR	Avg. Time
1	Codellama:70b	6	21	31	4	0.60	0.22	0.60	0.32	0.40	97.37
2	Llama2:70b	10	52	0	0	0.16	0.16	1.00	0.28	1.00	104.05
3	Llama3.3:70b	7	1	51	3	0.94	0.88	0.70	0.78	0.02	82.63

increasing false positives. The training objectives of general-purpose large LLMs often do not align with the requirements of structured data interpretation. Without specific fine-tuning on tasks involving strict schema adherence and rule-based logic, these models struggle to accurately process and evaluate structured inputs like ONOS flow rules. However, the substantial computational resources required to run 70B parameter models can limit their practicality, especially when smaller models achieve comparable or superior performance with significantly lower resource consumption.

G. END-TO-END IBN REALIZATION USING NetIntent

In this section, we show the implementation result of NetIntent. First we mention what LLM use choose for intent translation and conflict detection based on our LLM benchmark outcome. Then we report end-to-end delay of NetIntent for intent translation and intent activation on ODL and ONOS SDN controller.

1) SELECTED LLM FOR INTENT TRANSLATION AND CONFLICT DETECTION

Based on the results of Table 14 and 19, we choose QwQ:32b model for ODL for both intent translation and conflict detection tasks. As for ONOS, based on the results of Table 14 and 20, we choose Codestral:22b model for intent translation and QwQ:32b model for conflict detection tasks. The context example was set to 3 for intent translation for both the controllers.

2) IMPLEMENTATION OUTCOME IN ODL AND ONOS SDN CONTROLLERS

In Sec. VI-B, we described the experimental topology used to evaluate our IBN framework. Using this setup, NetIntent accepts high-level natural language intents from the user. For simulation, we provided various intents such as “*Deny packets originating from 10.0.0.1 destined for 10.0.0.4 using switch 1*” and “*Do slicing at node 3 and ensure TCP packets destined for port 80, addressed to 10.0.0.3, are forwarded via port 2 using queue 0*” to test the end-to-end IBN workflow. NetIntent first applies Algorithm 1 to translate the input intent into a structured JSON flow rule (based on target SDN controller, ODL or ONOS). This translated rule is then checked for potential conflicts against existing flow entries using Algorithm 2. If a conflict is detected, NetIntent attempts to resolve it; if resolution is not possible, the intent is not activated, and the user is notified. If no conflict is found, the system proceeds to install the rule via the southbound API—RESTCONF for ODL or FlowRuleService for

ONOS. Once installed, NetIntent verifies whether the flow is effective by inspecting its operational stats to confirm successful activation. The entire pipeline is fully automated, with the user interacting only at the intent level. Tables 22 and 23 present the measured durations from the start of intent translation to confirmed rule activation for both ODL and ONOS deployments. These timings include translation, conflict detection, installation, and operational verification phases. While intent assurance was not included in the timing results (as it operates continuously in a closed-loop fashion), it was active in the background following Algorithm 3 to ensure persistent verification of intent conformance.

VII. LIMITATIONS AND FUTURE WORK

Our benchmarks focus on representative natural language intents but do not encompass all possible configuration types or complex multi-intent scenarios. We also evaluated LLMs in their pretrained state, without task-specific fine-tuning, which may limit performance compared to customized models. Moreover, while NetIntent detects performance drift of intents, it does not address semantic drift [28] which is the presence of intents that are not specified by the user but are present in the network. Future research should explore fine-tuned models, as smaller, task-optimized models may perform well and be better suited for deployment in resource-constrained environments. Additionally, lightweight optimizations such as quantization, pruning, and knowledge distillation could further enhance the applicability of compact models. Beyond fine-tuning, customizing pretraining datasets and training strategies may improve model adaptability for IBN tasks and help reduce end-to-end latency. Investigating more efficient prompt designs could minimize translation errors and reduce false positives and negatives in conflict detection, thereby improving model reliability. Further efforts should examine LLMs’ reasoning capabilities to generate comprehensive explanations for decisions and output actions in IBN.

VIII. CONCLUSION

We benchmarked 33 open-source LLMs using our proposed IBNBench and found that their performance on isolated IBN tasks varies significantly. Crucially, task accuracy depends on prompt design, schema alignment, and in-context learning than on model size alone. Notably, models such as Gemma2-27B, QwQ-32B, QwQ-Fusion-32B, QwQ-Abliterated-32B, and Codestral-22B deliver consistent and high performance when guided by specialized prompting strategies. To enable LLM integration into a fully autonomous IBN pipeline, we introduced NetIntent, a unified, LLM-driven framework that

TABLE 22: End-to-end duration of NetIntent in ODL

Sl.	Intent	Intent Type	Existing Rule	Conflicting Rule	Translating LLM	Context	Detection LLM	E2E Time (s)
1	In switch 4, install a firewall to block traffic from 10.0.0.2 to 10.0.0.4.	Security	9	1	QwQ:32b	3	QwQ:32b	31.34
1	Drop all traffic from 10.0.0.9 on switch 2 while forwarding all other traffic normally.	Security	2	1	QwQ:32b	3	QwQ:32b	29.75
2	Using openflow switch 1, forward UDP traffic on port 80 to 10.0.0.3 via interface 2, queue 0.	QoS	8	1	QwQ:32b	3	QwQ:32b	24.02
4	If incoming traffic on interface 3 of node 3 is UDP to port 80, send via port 2, queue 1.	QoS	4	0	QwQ:32b	3	QwQ:32b	25.47
5	If port 2 on switch 3 receives TCP to port 80, send via interface 3, queue 0.	QoS	4	2	QwQ:32b	3	QwQ:32b	23.07
6	If port 2 on switch 3 receives UDP to port 80, pass via port 1, queue 0.	QoS	4	2	QwQ:32b	3	QwQ:32b	35.10
7	If port 2 on switch 3 receives TCP to port 80, pass via port 1, queue 0.	Forwarding	8	1	QwQ:32b	3	QwQ:32b	30.70
8	In switch 2, traffic from port 1 should pass through port 3.	Forwarding	2	1	QwQ:32b	3	QwQ:32b	20.10
9	Port 2 of switch 2 to 10.0.0.1 should use interface 4.	Forwarding	2	0	QwQ:32b	3	QwQ:32b	26.50
10	In switch 4, traffic from 10.0.0.1 to 10.0.0.4 should use output interface 4.	Forwarding	9	1	QwQ:32b	3	QwQ:32b	33.60

TABLE 23: End-to-end duration of NetIntent in ONOS

Sl.	Intent	Intent Type	Existing Rule	Conflicting Rule	Translating LLM	Context	Detection LLM	E2E Time (s)
1	In switch 4, install a firewall to block traffic from 10.0.0.2 to 10.0.0.4.	Security	9	1	Codenstral:22b	3	QwQ:32b	30.34
1	Drop all traffic from 10.0.0.9 on switch 2 while forwarding all other traffic normally.	Security	2	1	Codenstral:22b	3	QwQ:32b	25.13
2	Using openflow switch 1, forward UDP traffic on port 80 to 10.0.0.3 via interface 2, queue 0.	QoS	8	1	Codenstral:22b	3	QwQ:32b	21.02
4	If interface 3 on node 3 receives UDP to port 80, pass via port 2, queue 1.	QoS	6	0	Codenstral:22b	3	QwQ:32b	28.56
5	If port 2 on switch 3 receives UDP to port 80, pass via port 3, queue 0.	QoS	6	2	Codenstral:22b	3	QwQ:32b	23.07
6	If switch 3 receives UDP on port 2 to port 80, pass via port 1, queue 0.	QoS	6	2	Codenstral:22b	3	QwQ:32b	19.58
7	In node 1, traffic destined for 10.0.0.2 should use port 3.	Forwarding	8	1	Codenstral:22b	3	QwQ:32b	31.50
8	In switch 2, traffic from port 1 should pass through port 3.	Forwarding	2	1	Codenstral:22b	3	QwQ:32b	17.90
9	Traffic from port 2 of switch 2 to 10.0.0.1 should use interface 4.	Forwarding	2	0	Codenstral:22b	3	QwQ:32b	23.21
10	In switch 4, traffic from 10.0.0.1 to 10.0.0.4 should use output interface 4.	Forwarding	9	1	Codenstral:22b	3	QwQ:32b	29.20

automates the entire IBN lifecycle, including intent translation, policy activation, and assurance, across both ODL and ONOS SDN controllers. NetIntent coordinates LLM and non-LLM agents, enabling robust, natural language-driven intent realization with dynamic feedback and minimal human intervention. By releasing our open datasets, benchmarking results, and a practical NetIntent implementation, we establish a reproducible foundation for research on LLM-powered IBN. Our results demonstrate the feasibility of extensible, adaptive SDN automation, paving the way for future advances in intent-driven networking and LLM-native next-generation networking systems.

REFERENCES

- [1] D. Kreutz, F. M. V. Ramos, P. Veríssimo *et al.*, “Software defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
- [2] K. Park, S. Sung, H. Kim, and J.-i. Jung, “Technology trends and challenges in sdn and service assurance for end-to-end network slicing,” *Computer Networks*, p. 109908, 2023.
- [3] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B, C. Bhagat, S. Jain, J. Kaimal, S. Liang *et al.*, “B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan,” in *Proceedings of the ACM SIGCOMM Conference*, 2018, pp. 74–87.
- [4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven wan,” in *Proceedings of the ACM SIGCOMM Conference*, 2013, pp. 15–26.
- [5] O. Project, “Opendaylight: A linux foundation collaborative project,” 2015, accessed: 2025-01-15. [Online]. Available: <https://www.opendaylight.org/>
- [6] Jul 2024. [Online]. Available: <https://opennetworking.org/onos/>
- [7] M. Wheatley. (2015) At&t is using opendaylight big time, says exec. Accessed: 2025-06-17. [Online]. Available: <https://siliconangle.com/2015/08/07/att-using-odl-big-time-odsummit/>
- [8] (2024) Sdn archives - orange open source. Accessed: 2025-06-17. [Online]. Available: <https://opensource.orange.com/en/category/news/sdn/>
- [9] O. A. A. Johari Abdul Rahim, Rosdiadee Nordin, “Open-source software defined networking controllers: State-of-the-art, challenges and solutions for future network providers,” *Computers, Materials & Continua*, vol. 80, no. 1, pp. 747–800, 2024. [Online]. Available: <http://www.techscience.com/cmc/v80n1/57354>
- [10] OpenDaylight Project, “Opendaylight dlux web ui,” 2025, accessed: 2025-01-15. [Online]. Available: <https://test-odl-docs.readthedocs.io/en/latest/getting-started-guide/common-features/dlux.html>
- [11] E. Zeydan and Y. Turk, “Recent advances in intent-based networking: A survey,” in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE, 2020, pp. 1–5.
- [12] A. Leivadeas and M. Falkner, “A survey on intent-based networking,” *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 625–655, 2022.
- [13] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, “Intent-Based Networking - Concepts and Definitions,” RFC 9315, Oct. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9315>
- [14] T. Szigeti, D. Zacks, M. Falkner, and S. Arena, *Cisco digital network architecture: intent-based networking for the enterprise*. Cisco Press, 2018.
- [15] Nokia, “Digital operations center,” Nokia, Tech. Rep., 2024, accessed: 2025-07-10. [Online]. Available: <https://www.nokia.com/blog/harmonizing-the-use-of-intents-across-network-and-service-management/>
- [16] O. Project, “Openflowplugin developer documentation,” 2025, accessed: 2025-01-15. [Online]. Available: <https://docs.opendaylight.org/projects/openflowplugin/en/latest/devs/plugin.html>
- [17] C. Liu, X. Xie, X. Zhang, and Y. Cui, “Large language models for networking: Workflow, advances and challenges,” *IEEE Network*, 2024.
- [18] S. Long, J. Tan, B. Mao, F. Tang, Y. Li, M. Zhao, and N. Kato, “A survey on intelligent network operations and performance optimization based on large language models,” *IEEE Communications Surveys and Tutorials*, pp. 1–1, 2025.
- [19] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du, “Explainability for large language models: A survey,” *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 2, pp. 1–38, 2024.
- [20] 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.
- [21] R. Han, J. Wang, H. Sun, Z. Jiang, Q. Qi, Z. Zhuang, Y. Zhang, and J. Liao, “Network copilot: Intent-driven network configuration updating for service guarantee,” in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 2025, pp. 1–10.
- [22] A. Mekrache and A. 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.
- [23] C. Wang, M. Scazzariello, A. Farshin, S. Ferlini, D. Kostić, and M. Chiesa, “Netconfeval: Can llms facilitate network configuration?” *Proceedings of the ACM on Networking*, vol. 2, no. CoNEXT2, pp. 1–25, 2024.
- [24] 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.
- [25] N. Tu, S. Nam, and J. W.-K. Hong, “Intent-based network configuration using large language models,” *International Journal of Network Management*, vol. 35, no. 1, p. e2313, 2025.

- [26] F. Li, H. Lang, J. Zhang, J. Shen, and X. Wang, "Preconfig: A pre-trained model for automating network configuration," *arXiv preprint arXiv:2403.09369*, 2024.
- [27] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, and S. G. Rao, "Deploying natural language intents with lumi," in *Proceedings of the ACM SIGCOMM Conference Posters and Demos*, 2019, pp. 82–84.
- [28] S. Kou, C. Yang, and M. Gurusamy, "Safla: Semantic-aware full lifecycle assurance for intent-driven networks," *IEEE Transactions on Cognitive Communications and Networking*, pp. 1–1, 2025.
- [29] Z. Guo, F. Li, J. Shen, T. Xie, S. Jiang, and X. Wang, "Configreco: Network configuration recommendation with graph neural networks," *IEEE Network*, vol. 38, no. 1, pp. 7–14, 2023.
- [30] A. Alsudais and E. Keller, "Hey network, can you understand me?" in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2017, pp. 193–198.
- [31] H. Mahtout, M. Kiran, A. Mercian, and B. Mohammed, "Using machine learning for intent-based provisioning in high-speed science networks," in *Proceedings of the 3rd international workshop on systems and network telemetry and analytics*, 2020, pp. 27–30.
- [32] H. Yang, K. Zhan, Q. Yao, X. Zhao, J. Zhang, and Y. Lee, "Intent defined optical network with artificial intelligence-based automated operation and maintenance," *Science China Information Sciences*, vol. 63, pp. 1–12, 2020.
- [33] M. Kiran, E. Pouyoul, A. Mercian, B. Tierney, C. Guok, and I. Monga, "Enabling intent to configure scientific networks for high performance demands," *Future Generation Computer Systems*, vol. 79, pp. 205–214, 2018.
- [34] K. Dzeparoska, A. Tizghadam, and A. Leon-Garcia, "Intent assurance using llms guided by intent drift," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–7.
- [35] 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)*, 2023, pp. 545–550.
- [36] C. Creanga and L. P. Dinu, "Designing NLP systems that adapt to diverse worldviews," in *Proceedings of the 3rd Workshop on Perspectivist Approaches to NLP (NLPerspectives) @ LREC-COLING 2024*, G. Abercrombie, V. Basile, D. Bernadi, S. Dudu, S. Frenda, L. Havens, and S. Tonelli, Eds. Torino, Italia: ELRA and ICCL, May 2024, pp. 95–99. [Online]. Available: <https://aclanthology.org/2024.nlperspectives-1.10/>
- [37] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, B. Chang *et al.*, "A survey on in-context learning," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 1107–1128.
- [38] S. Pisharody, J. Natarajan, A. Chowdhary, A. Alshalan, and D. Huang, "Brew: A security policy analysis framework for distributed sdn-based cloud environments," *IEEE transactions on dependable and secure computing*, vol. 16, no. 6, pp. 1011–1025, 2017.
- [39] M. H. H. Khairi, S. H. S. Ariffin, N. M. A. Latiff, K. M. Yusof, M. K. Hassan, F. T. Al-Dhief, M. Hamdan, S. Khan, and M. Hamzah, "Detection and classification of conflict flows in sdn using machine learning algorithms," *IEEE Access*, vol. 9, pp. 76 024–76 037, 2021.
- [40] J. Zhang, J. Guo, C. Yang, X. Mi, L. Jiao, X. Zhu, L. Cao, and R. Li, "A conflict resolution scheme in intent-driven network," in *2021 IEEE/CIC International Conference on Communications in China (ICCC)*. IEEE, 2021, pp. 23–28.
- [41] J. Cui, S. Zhou, H. Zhong, Y. Xu, and K. Sha, "Transaction-based flow rule conflict detection and resolution in sdn," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–9.
- [42] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 49–54.
- [43] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 99–111.
- [44] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 469–483.
- [45] A. Leivadeas and M. Falkner, "Autonomous network assurance in intent based networking: Vision and challenges," in *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, 2023, pp. 1–10.
- [46] J. Wang, L. Zhang, Y. Yang, Z. Zhuang, Q. Qi, H. Sun, L. Lu, J. Feng, and J. Liao, "Network meets chatgpt: Intent autonomous management, control and operation," *Journal of Communications and Information Networks*, vol. 8, no. 3, pp. 239–255, 2023.
- [47] D. Dholakiya, T. Kshirsagar, and A. Nayak, "Survey of mininet challenges, opportunities, and application in software-defined network (sdn)," *Information and Communication Technology for Intelligent Systems: Proceedings of ICTIS 2020, Volume 2*, pp. 213–221, 2021.
- [48] "NetIntent," <https://github.com/Muhammadkamrul/NetIntent>, [Accessed 17-07-2025].
- [49] "Open flow documentation," <https://docs.opendaylight.org/projects/openflowplugin/en/latest/users/operation.html>, [Accessed 27-06-2025].
- [50] "Flow Rules - ONOS - Wiki — wiki.onosproject.org," <https://wiki.onosproject.org/display/ONOS/Flow+Rules>, [Accessed 13-07-2025].
- [51] X. Ye, S. Iyer, A. Celikyilmaz, V. Stoyanov, G. Durrett, and R. Pasunuru, "Complementary explanations for effective in-context learning," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 4469–4484. [Online]. Available: <https://aclanthology.org/2023.findings-acl.273/>
- [52] C. Si, D. Friedman, N. Joshi, S. Feng, D. Chen, and H. He, "Measuring inductive biases of in-context learning with underspecified demonstrations," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 11 289–11 310. [Online]. Available: <https://aclanthology.org/2023.acl-long.632/>
- [53] "FlowRuleStore (ONOS Java API (2.5.0)) — api.onosproject.org," <https://api.onosproject.org/2.5.0/apidocs/org/onosproject/net/flow/FlowRuleStore.html>, [Accessed 15-02-2025].
- [54] 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.
- [55] "Ollama llm models," <https://ollama.com>, accessed: 2025-01-15.

Appendix A EXAMPLE OF INTENT TRANSLATION

Intent: Forward TCP traffic on port 80 destined for 10.0.0.3 via interface 2, assigning it to queue 0 to prioritize.

```
ODL Flow Rule: {
  "flow-node-inventory:flow": [
    {
      "id": "1",
      "priority": 200,
      "table_id": 0,
      "flow-name": "tcp_port80_queue0_to_h3",
      "instructions": [
        {
          "instruction": [
            {
              "order": 0,
              "apply-actions": [
                {
                  "action": [
                    {
                      "order": 0,
                      "set-queue-action": {
                        "queue-id": 0
                      }
                    },
                    {
                      "order": 1,
                      "output-action": [
                        {
                          "output-node-connector": "2"
                        }
                      ]
                    }
                  ]
                }
              ],
              "match": [
                {
                  "ethernet-match": {
                    "type": 2048
                  },
                  "ip-match": {
                    "ip-protocol": 6
                  },
                  "ipv4-destination": "10.0.0.3/32",
                  "tcp-destination-port": 80
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}

ONOS Flow Rule: {
  "flows": [
    {
      "priority": 200,
      "timeout": 0,
      "isPermanent": true,
      "deviceId": "of:0000000000000001",
      "treatment": {
        "instructions": [
          {
            "type": "QUEUE",
            "queueId": 0
          },
          {
            "type": "OUTPUT",
            "port": "2"
          }
        ]
      },
      "selector": {
        "criteria": [
          {
            "type": "ETH_TYPE",
            "ethType": "0x800"
          },
          {
            "type": "IP_PROTO",
            "protocol": 6
          },
          {
            "type": "IPV4_DST",
            "ip": "10.0.0.3/32"
          },
          {
            "type": "TCP_DST",
            "tcpPort": 80
          }
        ]
      }
    }
  ]
}
```