RESEARCH-ARTICLE

NetConfEval: Can LLMs Facilitate Network Configuration?

CHANGJIE WANG, KTH Royal Institute of Technology, Stockholm, Stockholms, Sweden

MARIANO SCAZZARIELLO, KTH Royal Institute of Technology, Stockholm, Stockholms, Sweden

ALIREZA FARSHIN, NVIDIA, Santa Clara, CA, United States

SIMONE FERLIN, Red Hat, Inc., Raleigh, NC, United States

DEJAN KOSTIĆ, KTH Royal Institute of Technology, Stockholm, Stockholms, Sweden

MARCO CHIESA, KTH Royal Institute of Technology, Stockholm, Stockholms, Sweden

Open Access Support provided by:

Red Hat, Inc.

KTH Royal Institute of Technology

NVIDIA

Proceedings of the ACM on Networking, Volume 2, Issue CoNEXT2 (June 2024)
https://doi.org/10.1145/3656296
EISSN: 2834-5509

NetConfEval: Can LLMs Facilitate Network Configuration?

CHANGJIE WANG, KTH Royal Institute of Technology, Sweden
MARIANO SCAZZARIELLO, KTH Royal Institute of Technology, Sweden
ALIREZA FARSHIN[*], NVIDIA, Sweden
SIMONE FERLIN, Red Hat, Sweden
DEJAN KOSTIĆ, KTH Royal Institute of Technology, Sweden
MARCO CHIESA, KTH Royal Institute of Technology, Sweden

This paper explores opportunities to utilize Large Language Models (LLMs) to make network configuration human-friendly, simplifying the configuration of network devices & development of routing algorithms and minimizing errors. We design a set of benchmarks (NetConfEval) to examine the effectiveness of different models in facilitating and automating network configuration. More specifically, we focus on the scenarios where LLMs translate high-level policies, requirements, and descriptions (*i.e.,* specified in natural language) into low-level network configurations & Python code. NetConfEval considers four tasks that could potentially facilitate network configuration, such as (*i*) generating high-level requirements into a formal specification format, (*ii*) generating API/function calls from high-level requirements, (*iii*) developing routing algorithms based on high-level descriptions, and (*iv*) generating low-level configuration for existing and new protocols based on input documentation. Learning from the results of our study, we propose a set of principles to design LLM-based systems to configure networks. Finally, we present two GPT-4-based prototypes to (*i*) automatically configure P4-enabled devices from a set of high-level requirements and (*ii*) integrate LLMs into existing network synthesizers.

CCS Concepts: • **Networks** → **Network manageability**; • **Computing methodologies** → *Machine translation.*

Additional Key Words and Phrases: Large Language Models (LLMs), Network Configuration, Network Synthesizer, Code Generation, Routing Algorithms, Function Calling, RAG, Benchmark, P4

ACM Reference Format:
Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. NetConfEval: Can LLMs Facilitate Network Configuration?. *Proc. ACM Netw.* 2, CoNEXT2, Article 7 (June 2024), 25 pages. https://doi.org/10.1145/3656296

1 INTRODUCTION

Networks are the backbone of today's communication infrastructure, powering everything from simple online interactions to mission-critical services. Network operators wield significant control over the flow of data in a network, guiding it from one device to the next by carefully specifying a set of per-device configurations within the network infrastructure. These configurations – which can affect devices & services ranging from switches/routers, servers, network interfaces, network

[*] Work was done at RISE Research Institutes of Sweden.

Authors' Contact Information: Changjie Wang, KTH Royal Institute of Technology, Stockholm, Sweden; Mariano Scazzariello, KTH Royal Institute of Technology, Stockholm, Sweden; Alireza Farshin, NVIDIA, Stockholm, Sweden; Simone Ferlin, Red Hat, Stockholm, Sweden; Dejan Kostić, KTH Royal Institute of Technology, Stockholm, Sweden; Marco Chiesa, KTH Royal Institute of Technology, Stockholm, Sweden.

ACM 2834-5509/2024/6-ART7
https://doi.org/10.1145/3656296

functions, and even GPU clusters (used for training or inference of online services) – must be carefully configured to ensure the reliable transmission of information.

In the last decade, academia and industry adopted Software-Defined Networking (SDN) to simplify the configuration of networks compared to the previous traditional (monolithic) paradigm. Despite the benefits brought by SDN, network configuration entails frequent human intervention [15]. Manual configuration is, however, costly and difficult, *e.g.,* it requires expert developers who are familiar with large & complex software documentation and API interfaces, as well as knowledge about libraries, protocols, and their potential vulnerabilities.

Many efforts have attempted to simplify the process of compiling a high-level policy specified by a network operator into a set of per-device network configurations [8, 15, 16, 25, 40, 46] and to minimize errors by generating configurations with provable guarantees via verification [1, 6, 7, 17, 20, 28, 38]. Nevertheless, network configuration remains an arduous, complex, and expensive task for network operators. Moreover, the abstraction and composition of networks force these configuration tools to employ a self-defined specification or language for succinctly describing network requirements. For instance, SyNet [15] introduced a stratified Datalog query language to express routing protocols and network requirements. These approaches impose another challenge on network operators because they must acquire proficiency in a new domain-specific language that may not be widely used and could potentially have flaws.

Large Language Models (LLMs) open up new opportunities. Generative AI and LLMs (*e.g.,* OpenAI's GPT [12, 34], Google's PaLM [5, 14], and Meta's Llama [47, 48]) have recently shown great potential to generate coherent and contextually relevant content, answer questions, and even engage in meaningful conversations with users; all of these capabilities offer immense potential for applications in various industries. For instance, GitHub Copilot, Meta CodeLlama [41], and Amazon CodeWhisperer offer services using LLMs to perform coding tasks.

In this paper, we explore a scientific question of whether LLMs can be used to facilitate network configuration. LLMs enable new possibilities in *quickly* acquiring *vast* knowledge (*e.g.,* they can potentially learn IETF specification documents of protocols standards as well as best practices), which goes beyond the capabilities of humans and goes beyond traditional network configuration tools. We discuss various opportunities to simplify and potentially automate the configuration of network devices based on human language prompts/inputs.

A few recent works have highlighted the potential benefits of using Natural Language Processing (NLP) and LLMs to address networking problems (*e.g.,* configuration synthesis, verification, and translation). In particular, they have focused on (*i*) proposing a verified programming method for LLMs [30], (*ii*) using LLMs to analyze networking documents [43], and (*iii*) analyzing network topologies & communication graphs using LLMs [29]. These works, however, do not present sufficient quantitative evaluation for various LLMs and also fail to provide meaningful insights to the community regarding the use of LLMs for configuring networks; see §2 for details.

Leveraging LLMs for network configuration is challenging. While LLMs hold great potential for simplifying network management, there are a number of critical challenges that may hinder their widespread deployment. First, LLMs are notoriously unreliable, producing outputs that may be completely incorrect. Second, reducing inaccuracies produced by LLMs highly depends on the way the user prompts the LLM, a concept known as "prompt engineering" in the community. Third, operating or using LLMs is expensive. Training an LLM such as GPT-4 may cost millions of dollars [52]. Even the cost associated with using the available APIs offered by existing companies is non-negligible. In this work, we further observe that drawing conclusions based on statistically significant experiments may easily become overly expensive.

Evaluating LLMs to simplify network configuration. We systematically study emerging LLMs to facilitate the configuration of network devices and the development of routing algorithms from high-level requirements & descriptions specified in natural language. To do so, we design a set of benchmarks (`NetConfEval`) to evaluate the capability of existing LLMs for network configuration purposes. Having such a benchmark is crucial for tracking the fast-paced evolution of LLMs and their applicability for networking use cases, as done for other tasks (*e.g.,* `HumanEval` [13] for code generation, `CausalBench` [55] for causal reasoning, and `ReasonEval` [54] for mathematical reasoning). Within the scope of this work, we currently focus on the following tasks, which are commonly related to network configuration:

(1) Generating high-level requirements into a formal specification format;
(2) Translating high-level requirements into API/function calls, which is particularly interesting for SDN and automation protocols in modern network equipment;
(3) Writing code to implement routing algorithms based on high-level descriptions;
(4) Generating low-level configuration for existing/new protocols based on input documentation.

We note that while we focus on the above four tasks, we envision that `NetConfEval` will be expanded with additional use cases shared from the industrial and academic communities.

Our main findings show that some LLMs are mature enough to automate simple interactions between users and network configuration systems. More specifically, GPT-4 exhibits extremely high levels of accuracy in translating human-language intents into formal specifications that can be fed into existing network configuration systems. Smaller models also exhibit good levels of accuracy, however only when these are fine-tuned on the specific tasks that need to be solved, thus requiring expertise in the specific tools and protocols that one expects to use. GPT-4 also exhibits acceptable capabilities in generating basic routing algorithms for computing paths in a network.

Learning from our study, we propose a set of principles for building LLM-based systems to facilitate the configuration of the network. Finally, we present two *functional* GPT-4-based prototypes capable of (*i*) generating fully working P4 configurations (without fine-tuning) to enforce a class of path policies expressed as natural language requirements, and (*ii*) integrating with existing network synthesizers to provide a user-friendly interface. Although we only target switch/router configuration, the same principle could be applied to other network configurations (see §4) and tasks. For instance, LLMs can potentially simplify the cumbersome task of managing Kubernetes-based clusters, as these get larger and more distributed, or troubleshooting networks.

Lessons learned. Running experiments is expensive and it is easy to spend up to thousands of dollars even on a single use case evaluation.[1] We envision that these costs will become even more prohibitive for academia to afford as model sizes and their handled context have increased by a factor of 10× in the last year [35]. We also observed that finding the correct prompts is challenging and it highly affects the results. We confirm that techniques based on “step refinement” [44] are more effective also in tasks such as routing-based code generation. We observed that small models were ill-suited for code generation tasks, even those that were specifically fine-tuned on Python coding. We believe that fine-tuning models on network-related problems will not be sufficient as network operators often need to write new functionalities that cannot easily be envisioned when fine-tuning the model (*e.g.,* writing code based on new ideas from scientific papers, RFCs, etc.).

Contributions. In this paper, we:

- Design a set of benchmarks (called `NetConfEval`[2]) to evaluate LLMs for networking use cases;

[1] As an example, reading the entire FRRouting documentation using GPT-4-Turbo costs roughly $1 per run. As at least 100 runs are needed to obtain statistically significant results, the cost for already jumps to $100. Comparing 10 different prompts requires $1000.

[2] The code, prompts, and data are available on GitHub [49] and HuggingFace [50].

- Formulate a set of principles based on our benchmarking experiments to facilitate building LLM-based systems for configuration & development of networks.
- Finally, present two prototypes for real-world networking systems to demonstrate the feasibility of LLMs to configure networks; GPT-4-Turbo can generate P4 configurations based on high-level requirements in natural language; and can be integrated into existing network synthesizers.

2 EXISTING WORKS ON AI & LLMS FOR NETWORK CONFIGURATION

Using AI for networking has already been explored by previous works. Beurer-Kellner *et al.,* [10] proposed a Neural-Algorithmic-Reasoning-based synthesizer to generate fast and scalable network configurations. Ben-Houdi *et al.,* [24] presented potential use cases for taking advantage of NLP techniques in networking. Additionally, Nile [26] introduced an intent-based networking scheme where NLP is applied in translation to a simple and limited set of requirements expressed in basic natural language. However, with the advancement of LLMs, traditional NLP models seem to be less efficient for processing large context or handing complex tasks. LLMs have captivated researchers' interest in exploring their potential in the networking area. Recently, Nail [4] revealed the challenges of using LLMs in network configuration tasks, emphasizing the necessity for consistent human supervision. Moreover, it also suggested that user experience can be improved by integrating their system with emerging chatbots. Such integration would enable users to interact with the system to configure P4 programmable switches and also facilitate the detection of potential conflicts. A holistic framework integrating LLMs in network incident management is proposed in [21]. To apply the framework into practice, authors analyze the fundamental requirements to be considered, based on discussions with network operators. In [30], authors introduce a Verified Prompt Programming method aimed at eliminating the need for human supervision throughout the process. The method combines GPT-4 with verifiers that automatically correct errors and provide automatic consistent feedback to the LLMs until a correct answer is generated. The authors prove the method by showing GPT-4's capabilities in translating router configurations between two different vendors' syntaxes, and to implement a BGP no-transit policy on multiple routers. Nevertheless, they do not provide a quantitative analysis of GPT-4's effectiveness in accomplishing the task. In [29], authors explore LLMs' potential in generating high-quality code for graph manipulation in network traffic analysis and network lifecycle management. However, the paper only focuses on the graph analysis and manipulation tasks. Network management covers many more tasks such as intent translation, traffic engineering, and configuration generation, which are discussed thoroughly in our paper. PROSPER [43] utilizes LLMs to extract specifications and understand network protocols from their corresponding RFCs. Our paper takes a step further, focusing instead on creating protocol-specific configurations using newly acquired knowledge. Besides academic research, there is a growing trend in the industry to integrate LLMs into their products, mainly to streamline network troubleshooting and root cause analysis tasks [32, 53]. Yet, these proposals have not adopted LLMs for the automatic configuration of networks, which is one of the goals of this paper.[3]

3 CAN LLMS FACILITATE NETWORK CONFIGURATION?

This section presents and evaluates the opportunities that LLMs offer in automating network configuration. We mainly focus on four different use cases: (*i*) translation of high-level requirements (expressed in natural language) into formal, structured, machine-readable specifications; (*ii*) translation of high-level requirements into function/API calls; (*iii*) development of code to implement routing algorithms; and (*iv*) generation of detailed, device-compatible configurations for various routing protocols. More specifically, we have developed `NetConfEval`, the *first*

[3] An earlier version of our work briefly discussed these aspects [51].

model-agnostic network configuration benchmark for LLMs. NetConfEval holds the promise of establishing a new standard benchmarking for assessing both current/future LLMs relevant for network configuration. We have developed NetConfEval using LangChain [22], a framework for building LLM-based applications, to facilitate extending our benchmark with newer tasks and supporting newer models.

3.1 Translating High-Level Requirements to a Formal Specification

In this part, we discuss LLMs' ability to translate network operators' requirements into a formal specification. To make use of most state-of-the-art network configuration tools (*e.g.,* [15, 16]), network operators should input their network requirements in a pre-defined specification or language, which requires learning the desired format for various tools. However, LLMs could help reduce this burden by allowing network operators to interact in natural language, and then convert such input into a valid specification for different configuration tools. In this regard, we propose a detailed prototype of how LLMs could be integrated with existing configuration tools in §4.2.

For instance, the input information can be converted into a simple data structure to specify the reachability, waypoints, and load-balancing policies in a network; these are key requirements heavily used by the state-of-the-art [3, 9, 11, 15, 45]. Depending on the complexity of the network requirements & policies, a network operator may directly add/remove new entries in the formal specification format (*e.g.,* to consider link preferences and/or resilience to more efficiently configure the network). Listing 1 shows a sample input and its corresponding formal specification.

Listing 1. An example data structure (right) representing a formal specification for a sample input (left). *Reachability* refers to the ability of traffic originating from a switch to reach a destination host. *Waypoint* forces the traffic from a switch towards a host to traverse a predefined sequence of intermediate switches. *Load-Balancing* forces the traffic from a host towards another host to traverse multiple paths.

Network components:
- 4 switches: s1, s2, s3, s4
- 2 end-hosts: h1, h2

Requirements:
- All the switches can reach all the destination hosts.
- Traffic from s1 to h1 should travel across s2.
- The traffic from h1 to h2 is load balanced on 3 paths.

```
{
  "reachability": {
    "s1": ["h1", "h2"],
    "s2": ["h1", "h2"],
    "s3": ["h1", "h2"],
    "s4": ["h1", "h2"],
  },
  "waypoint": {
    ["s1", "h1"]: ["s2"],
  },
  "loadbalancing": {
    ["h1", "h2"]: 3,
  }
}
```

Experiments. To understand whether existing LLMs are sufficient for our purpose, we devised an experiment as follows: we (*i*) generated 3200 network requirements focusing on reachability, waypoint, and load-balancing, using Config2Spec [11] on a topology composed of 33 routers; (*ii*) randomly picked a certain number of requirements and sliced them with various batch sizes[4]; (*iii*) for each batch, we converted them into the expected formal specification format using a Python script; (*iv*) transformed them to natural language based on predefined templates; (*v*) asked an LLM to translate these requirements from natural language to the formal specification; and (*vi*) evaluated the efficiency of different LLMs by comparing the translated version of formal

[4] We initialized the random function with a specific seed to ensure consistent results across various models.

specification with the expected one. We did the evaluations on different combinations of policies (*i.e.,* reachability, reachability + waypoint, and reachability + waypoint + load-balancing). The batch size definition varies with the number of policies, *i.e.,* a batch size of 2 in the reachability + waypoint scenario indicates that the batch contains a reachability and a waypoint specification. In our analysis, we use various OpenAI (*i.e.,* GPT-3.5-Turbo, GPT-4, and GPT-4-Turbo) and Meta CodeLlama (*i.e.,* 7B-instruct and 13B-instruct) models. Additionally, we fine-tune GPT-3.5-Turbo[5] and CodeLlama-7B-Instruct models using the OpenAI dashboard and QLoRA, respectively. To do so, we created a dataset similar to the one used for the evaluation but with slightly different templates and then fine-tuned the models for 3 epochs. Figure 1 shows the results of our analysis. GPT-4 performs similarly to GPT-4-Turbo; we do not show the results for better visibility in the figures.

Takeaways. The results of our analysis demonstrate that:

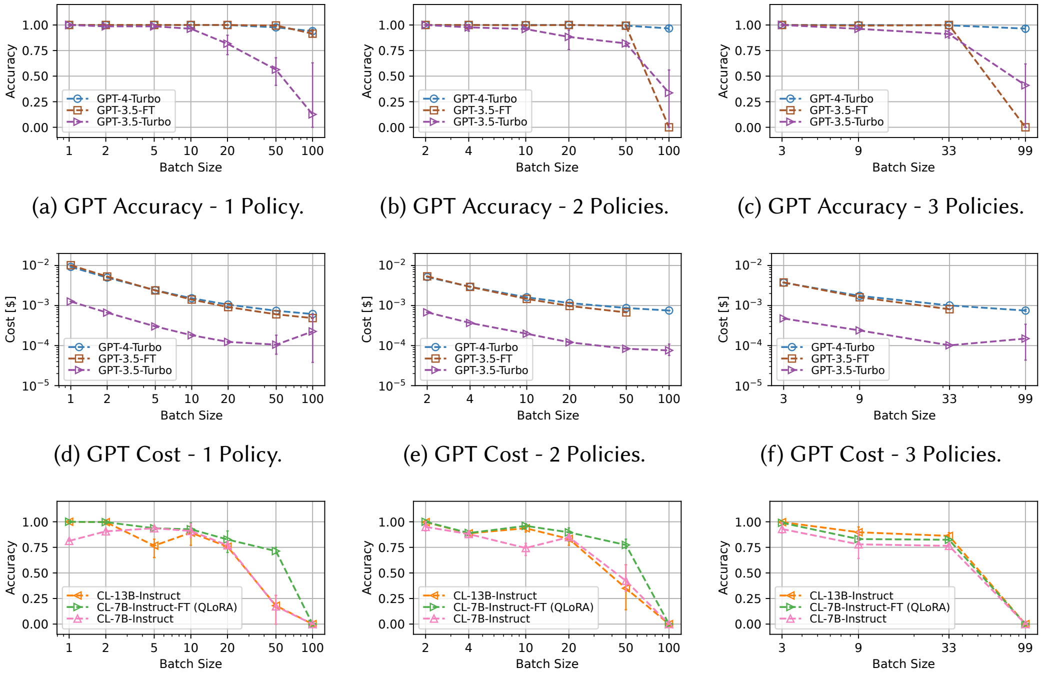
- **Selecting the appropriate batch size is key for cost-effective & accurate translations.** Since each inference request should contain preliminary instructions within the prompts[6], batching the translations could reduce the per-translation cost (both in terms of money and time). However, our results show that the accuracy of translations is worsened with larger batch sizes (especially for non-GPT-4 models). Therefore, it is important to carefully select a suitable batch size for each model to ensure the right trade-off between accuracy and cost (*i.e.,* ensuring cost-effectiveness and guaranteeing correctness). For instance, translating 20 requirements in one batch with GPT-4-Turbo is around 10× cheaper than translating 20 requirements one by one, while still achieving 100% accuracy (see Figure 1a and 1d).
- **Context window matters.** Our results show that the translation accuracy decreases as we increase the batch size. We speculate that this reduction in accuracy may be related to reaching the context length (*i.e.,* 4096 maximum input/output tokens for all models except GPT-4-Turbo supports 128k input tokens). More specifically, in most of the experiments, we noticed that the generated LLMs outputs are always truncated when the batch size gets closer to 100.
- **Fine-tuning improves accuracy.** Fine-tuning LLMs for a specific purpose could optimize their accuracy. For example, while GPT-3.5-Turbo apparently performs worse than GPT-4-Turbo, Figures 1a, 1b, and 1c show that a fine-tuned version of GPT-3.5-Turbo achieves similar accuracy to GPT-4-Turbo, but with a higher cost since OpenAI sets a higher per-token price for fine-tuned models. Figures 1g, 1h, and 1i show a similar takeaway for CodeLlama models, where fine-tuning CodeLlama-7B-Instruct model using QLoRA can achieve better accuracy than the original model & sometimes better than 13B-Instruct model.
- **GPT-4 beats the majority of existing models in our experiments.** Our results show that GPT models generally can achieve higher accuracy than their open-source counterparts (*e.g.,* CodeLlama). We also experiment with other open-source models (*e.g.,* Mistral-7B-Instruct and Llama-2-Chat) & Google Bard[7], and they generated less accurate translations.

3.1.1 Conflicts/Ambiguity. The ambiguity of human language and unfamiliarity with specific classes of problems may result in misinterpretations. Moreover, when multiple entities (*e.g.,* customers, managers, or operators) specify their requirements individually, their combination may contain contradictory requirements. Even when a single network operator is involved, contradictory network requirements can still occur, especially when the number of requirements is large. These conflicts can result in severe configuration failures before operators discover them. Previous network

[5] At the time of writing, it is not possible to fine-tune GPT-4 using OpenAI APIs.

[6] Prompts are conversation-wide instructions to the LLMs.

[7] We used the chat interface since we did not have access to the official API. We did not include these results, since we were not able to automate the process and repeat the experiments to increase statistical relevance.



(a) GPT Accuracy - 1 Policy. (b) GPT Accuracy - 2 Policies. (c) GPT Accuracy - 3 Policies.

(d) GPT Cost - 1 Policy. (e) GPT Cost - 2 Policies. (f) GPT Cost - 3 Policies.

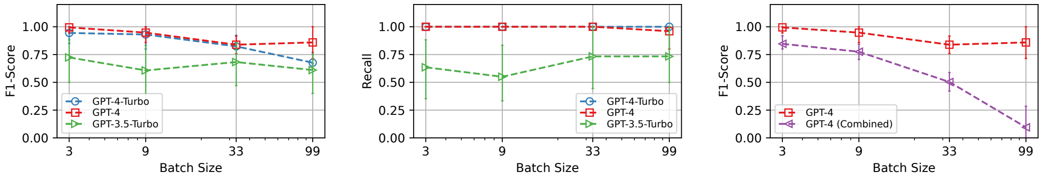
(g) CodeLlama Accuracy - 1 Policy. (h) CodeLlama Accuracy - 2 Policies. (i) CodeLlama Accuracy - 3 Policies.

Fig. 1. It is important to find the appropriate batch size when translating high-level requirements into a formal specification format. GPT-4-Turbo achieve higher accuracy than GPT-3.5-Turbo and CodeLlama. We run CodeLlama on the Leonardo supercomputer equipped with NVIDIA custom Ampere GPU 64 GB.

synthesizers have often overlooked this aspect, and they are typically paired with formal tools capable of detecting conflicts.

Simple conflicts. A common case for a *simple* conflict is when two requirements explicitly include contradictory information. For instance, a requirement specifies s1 to reach h2 while another requirement prevents s1 from reaching h2. To evaluate LLMs' performance in conflict detection, we design a set of experiments based on the ones described in §3.1. We randomly selected one requirement from each batch, generated a conflicting requirement (*e.g.,* the conflicting requirement of "h1 can reach h2" is "h1 *cannot* reach h2"), and inserted them back into the batch. We evaluate the effectiveness of LLMs in detecting simple conflicts in two scenarios. First, we consider conflict detection as a separate step and explicitly ask an LLM to search for a conflict and report it. Second, we ask the LLM to perform conflict detection during the translation of requirements into a formal specification format. We refer to the latter scenario as "Combined". Figures 2a and 2b show the results of various GPT models when performing conflict detection. These results show that GPT-4 and GPT-4-Turbo reach almost 100% recall[8] for different numbers of input requirements. These results suggest that such models are always capable of detecting conflicts when a batch contains a conflicting requirement (*i.e.,* they do not report a false negative). Figure 2c demonstrates that conflict detection is much more accurate when done in isolation. As opposed to GPT-4 models, our results demonstrate a poor recall and F1-Score for GPT-3.5-Turbo model.

[8] The recall metric reports the ratio of true positives (*i.e.,* true positives divided by the true positives and false negatives).



(a) F1-Score. (b) Recall. (c) F1-Score w/ & w/o combined steps.

Fig. 2. GPT-4 can successfully detect simple conflicts in the provided high-level requirements.

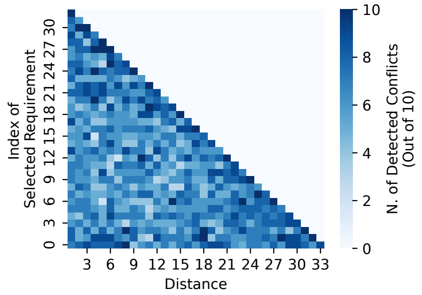


Fig. 3. The impact of distance on GPT-3.5-Turbo when detecting simple conflicts.

In order to determine whether this performance degradation is related to the smaller context window size of GPT-3.5-Turbo, we designed a new experiment to measure the impact of the position of a conflicting requirement in a batch (*i.e.,* to understand whether adding a conflicting requirement at the beginning, middle, or at the end could affect the accuracy of conflict detection). More specifically, we select a few batches with 33 requirements. For each requirement in the batch, we iterated through all the possible positions (indices), where we could insert a conflicting requirement. Figure 3 shows the number of conflicts detected out of 10 runs. One can observe that GPT-3.5-Turbo may be better at detecting conflicting requirements at the end of the batch (see the relatively darker squares at the hypotenuse of the heatmap). Finally, we compare the performance of GPT-4 when performing conflict detection separately and combined with translation (see Figure 2c).

Complex conflicts. Identifying *complex* conflicts between requirements can be more challenging; hence, without fine-tuning, LLMs may struggle to directly detect these contradictions. For instance, an example of such complex conflicts is when a requirement specifies s1 to reach h2 through s2 while another requirement prevents s2 from reaching h2. We observed that most of the time GPT-4 translates these types of conflicts into *Reachability* and *Waypoint* specifications *without* reporting any conflicts, which is not desirable. Moreover, another type of complex conflict may occur when a large number of requirements is segmented into small batches to enhance accuracy (see § 3.1), which could introduce inter-batch conflicts. To address this issue, we propose conducting intra-batch conflict detection before translating the requirements. If no conflict is identified within the batch, the translation results can be merged into the formal specification. Once the translation is completed, it is possible to use Satisfiability Modulo Theories (SMT) solvers to ensure there exists *a solution* for a given formal specification. In case of detecting any contradictions, an LLM can interpret them and provide feedback to network operators, which remains as our future work.

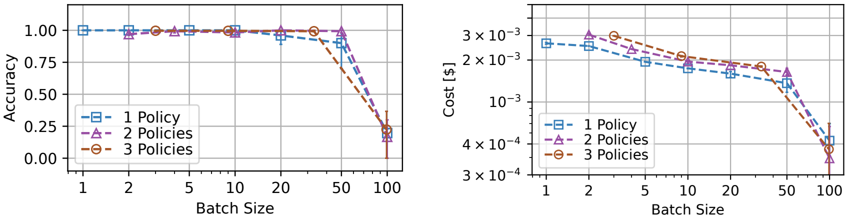
Takeaways. The results of our analysis demonstrate that:

- **Breaking tasks helps.** Comparing the accuracy of conflict detection when (*i*) performed as a separate task and (*ii*) when performed during translation, we observe that separating the conflict detection & translation results in better accuracy (*i.e.,* higher F1-Score). This finding motivates the necessity of splitting complex tasks into multiple simpler steps and solving them separately.
- **Simple conflicts can be detected.** GPT-4 and GPT-4-Turbo models are capable of successfully detecting all those simple conflicting requirements that we presented to them.

- **Detected conflicts could be false positives.** GPT-4 and GPT-4-Turbo sometimes report false positives (*i.e.,* they detect a conflict when there is none). A concrete false positive example is "For traffic from Rotterdam to 100.0.4.0/24, it is required to pass through Basel, but also to be load-balanced across 3 paths which might not include Basel". LLMs tend to overinterpret the conflict by, *e.g.,* considering load-balance conflicting with waypoints. It is, however, possible to minimize false positives by providing examples for possible conflicts in the input prompts.

3.2 Translating High-Level Requirements to Functions/API Calls

This subsection discusses the ability of LLMs to translate natural language requirements into corresponding function/API calls, which is a common task in network configuration since many networks employ SDN, where a software controller can manage the underlying network via direct API calls (*e.g.,* installing some configurations into a router). SDN is not the only application, as many modern network devices implement network automation protocols such as NETCONF, RESTCONF, gRPC, or Ansible [23]. A practical LLM-based system should be able to select the appropriate function/API call among multiple functions. Unfortunately, at the time of writing, GPT-4-Turbo is the only model that natively supports multiple function calls together in a batch (namely, Parallel Function Calling [33]). Therefore, we start with evaluating the performance of GPT-4-Turbo for this specific use case. To do so, we consider a library that supports three policies: (*i*) `add_reachability(switch, host)`, (*ii*) `add_waypoint(switch, host, [waypoints])`, and (*iii*) `add_loadbalancing(switch, host)`. We ask the LLM to select the appropriate function for a given input requirement. We use a similar procedure as §3.1, except asking the LLM to translate the input requirement into function calls rather than a formal specification format. Figure 4 shows the result of GPT-4-Turbo; the different number of policies specifies the policy types in the batch and corresponding prompts (*i.e.,* 1 only contains reachability requirement; 2 contains reachability & waypoint requirements; and 3 contains all three types).

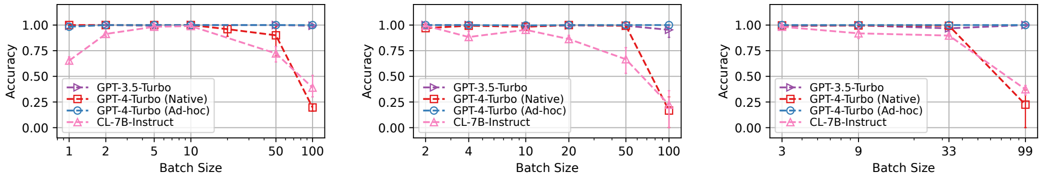


(a) Accuracy.

(b) Cost per requirement.

Fig. 4. GPT-4-Turbo can successfully select the appropriate function when translating high-level requirements. Using more functions increases the cost.

As previously mentioned, the other LLMs do not natively support Parallel Function Calling. To ensure a fair comparison, we devised a specific prompt that mimics the behavior of the native method. In particular, we provide the LLM with the signature of the function, along with a brief textual explanation of the function behavior and its parameters. We then ask the LLM to convert human input to the corresponding function calls. After the translation, we extract the functions & parameters' values from the result and we enable individual function execution. This approach allows us to conduct the same type of experiment as with native function calling. Figure 5 shows the result of the tests conducted using GPT-4-Turbo, GPT-3.5-Turbo, and CodeLlama-7B-Instruct. GPT-4-Turbo and GPT-3.5-Turbo achieve almost 100% accuracy, even with larger batch sizes, outperforming the native function calling supported by OpenAI. CodeLlama exhibits good levels of accuracy even with large batches, which are surprisingly good for a small model.



(a) Accuracy - 1 Policy.

(b) Accuracy - 2 Policies.

(c) Accuracy - 3 Policies.

Fig. 5. With ad-hoc function calling, GPT-3.5-Turbo outperforms the native GPT-4-Turbo function calling.

Takeaways. The results of our analysis demonstrate that:

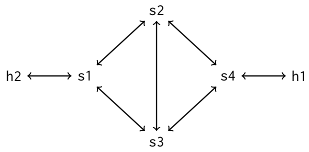
- **Calling the appropriate function/API is feasible.** The use of either native or ad-hoc function calling allows GPT-4-Turbo to successfully select the correct function with almost 100% accuracy, especially when the batch size is 5, 10, and 20. Even small models like CodeLlama demonstrate a good level of accuracy using ad-hoc function calling.
- **Native function calling has shortcomings.** With larger batch sizes, the performance decreases due to the limit of GPT-4-Turbo's output context window. We witnessed the output was truncated. Moreover, GPT-4-Turbo may also occasionally fail to translate the specifications into the correct function calls with the smallest batch sizes (*i.e.,* 2 and 3). In this situation, the LLM fails to recognize that two distinct specifications (*e.g.,* reachability + waypoint) should be converted into two different function calls (*e.g.,* `add_reachability` and `add_waypoint`). Instead, it merges them into a single function call. We expect that this mismatch will be solved in future updates of this OpenAI's functionality.
- **Ad-hoc function calling performs better than native function calling.** Our ad-hoc function calling method achieves higher accuracy compared to the native function calling implemented by GPT-4-Turbo. We speculate that OpenAI pre-trained GPT-4 to identify function calls and generate responses in a JSON format. To enhance performance, it appears developers may have reduced the context window size, which subsequently hinders the LLM ability to process larger inputs (*i.e.,* the model consistently fails with larger batch sizes). In contrast, a straightforward translation approach, such as the one used in our ad-hoc method, introduces less overhead compared to structured JSON outputs. This allows us to fully leverage LLMs' capabilities, resulting in better performance especially when the number of function calls in each batch grows.

3.3 Developing Routing Algorithms

Traffic engineering is a critical yet complex problem in network management, particularly in large networks. It guarantees efficient data transmission and prevents congestion. In SDN architectures, the controller is the central component that orchestrates data flow across the network. The controller should leverage relevant routing algorithms to enforce network policies. These algorithms play a crucial role in routing, prioritization, and various other network-related aspects as specified by network operators. The design of these algorithms is indeed a complex task since it involves understanding network topologies, traffic patterns, and specific requirements set by network operators. Our preliminary experiments highlight that GPT-4-Turbo is already *unable* to *directly* calculate the routing paths, using a simple shortest path policy, within a network composed of 10 devices. Therefore, rather than interacting with LLMs to directly translate requirements into routing paths, we leverage LLMs' capabilities in code generation, prompting them to write code for a given routing problem rather than generating routing paths. The performance of the generated code should remain unaffected by the topology size, as long as the code is correctly generated and the requirements/functionalities remain unaltered.

Experiment. We only evaluate LLMs' capability in generating Python code for routing purposes.[9] Our experiment asks the models to create functions that compute routing paths based on specific network requirements. As in §3.1, we consider the shortest path, reachability, waypoint, and load-balancing. The desired function should accept a network topology and formal requirements as input and produce routing paths in a predefined data format.[10] Listing 2 shows an example of the routing paths for Listing 1. While LLMs can generate code from scratch, their output is not always error-free, as already highlighted by previous work [29]. To ensure the correctness of the generated code, we implemented a verification system that checks the syntax and logical correctness of the code using pre-defined test cases. After generating the code, we verified that all the tests were successfully passed. If so, the process terminates; otherwise, the LLM re-generates the code. To study the impact of prompting LLMs for code generation, we performed the experiment by asking GPT-4-Turbo to generate the code by providing (i) only essential information (*i.e.,* input and output formats and the code's purpose) and (ii) detailed algorithmic instructions to guide the generation process (see an example in Appendix A). Additionally, motivated by prior work [29, 30], we wanted to analytically evaluate the impact of feedback in code correction when the LLM generates incorrect code. Our feedback is derived from Python syntax errors & the outcome of the test cases, and includes the input along with the actual and expected output. We re-prompt the model by re-explaining the task and incorporating both the previously generated code and the feedback message.

Listing 2. An example topology (left) and routing information acting as the high-level configuration (right). This data structure shows the shortest path between every pair of end hosts.



```
{
  "h1": {
    "h2": ["s4", "s2", "s1"],
  },
  "h2": {
    "h1": ["s1", "s2", "s4"],
  },
}
```

Takeaways. Our experiments (see Figure 6 for GPT-4-Turbo results) indicate that:

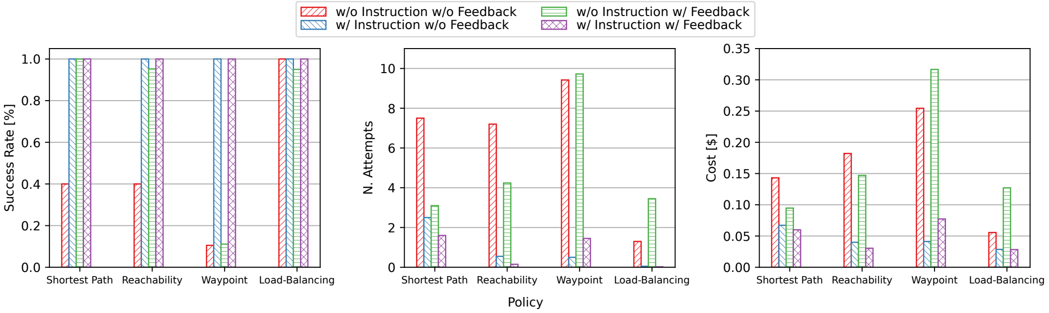
- **GPT-4 is the only model capable of successfully generating code.** As already highlighted in §3.1, GPT-4 models are the only models capable of code generation for network purposes (GPT-4 results can be found in Appendix A). For completeness, we conducted the same experiment using GPT-3.5-Turbo, Google Bard, and smaller open-source models specifically devised for the purpose (*i.e.,* CodeLlama, Phi, Mistral-7B). None of these models were able to generate correct code (not even once in any of the tested scenarios), so we do not report the results in the paper.
- **Providing detailed instructions is helpful in generating the correct code.** Our experiments highlight that when GPT-4-Turbo is prompted only with essential information (red bar), it is able to correctly generate code (Figure 6a) but with a higher number of attempts, especially for the

[9] Previous research on GPT-4 demonstrated that the LLM has strong Python code generation abilities [37].

[10] The code generation task for different policies is different. For the shortest path policy, GPT-4-Turbo is prompted to generate code from scratch. For reachability, we provide a code base implementing the shortest path policy. The new code should ensure that switches along the path can reach the end host. In the case of the waypoint policy, we use a base code implementing the shortest path and reachability, and the newly generated function should ensure that the path includes defined waypoints. Lastly, for the load-balancing policy, we input the base code implementing the shortest path policy. The resulting function should select the k-top shortest paths to allow load balancing.

shortest path, reachability, and waypoint policies (Figure 6b). Conversely, by providing detailed instructions, the model is able to *always* generate correct code with fewer or no retries (blue bar). While this approach requires extra human input, it is feasible to train or fine-tune another LLM to automatically produce detailed algorithmic instructions.

- **Feedback upon failure helps the model to generate correct code quickly.** In complex tasks (*i.e.,* shortest path and waypoint generation), providing proper feedback upon failure helps GPT-4-Turbo to generate correct code with fewer attempts (green and purple lines). However, for less complex tasks (*i.e.,* reachability and load-balancing), the LLM often solves the problem in a single attempt. In these cases, simply asking the model to "regenerate" the code with no feedback instructions may resolve the issues in the previously generated code while reducing the cost due to fewer tokens. Thus, the need for feedback should be based on the level of complexity.



(a) Success Percentage. (b) Number of Attempts. (c) Cost.

Fig. 6. Providing detailed instructions and feedback improves code generation using GPT-4-Turbo.

3.4 Generating Low-level Configurations

This section explores the problem of transforming high-level requirements into detailed, low-level configurations suitable for installation on network devices. Configuring networks is a cumbersome task, even for humans, due to the need to deal with a plethora of different networking protocols, often with vendor-specific syntaxes. We argue that LLMs can play a crucial role in this field, as they have the capability to process and comprehend extensive documentation (*e.g.,* RFCs, frameworks documentation) more efficiently than one or multiple humans [43]. Additionally, they can proactively adapt to software updates and newly identified vulnerabilities by autonomously updating configurations without the need for human input. Network configuration is done at different levels and scales, from configuring network interfaces (*e.g.,* specifying IP addresses) to routing paths on the Internet (*e.g.,* setting up BGP and/or OSPF); in this work, we mainly focus on routing protocol configurations. Prior research [30] has shown that LLMs can effectively translate network configurations across different vendor syntaxes. We go a step further and explore the possibility of generating configurations from scratch for (*i*) established protocols (*i.e.,* RIP, OSPF, and BGP) and (*ii*) emerging, undocumented protocols (*i.e.,* RIFT [39]).

Experiment. To understand how capable LLMs are in performing this task, we devised the following experiment. We first handpicked four network scenarios publicly available in the Kathará network emulator repository [18]. We chose Kathará due to its ease-of-use and its widespread adoption for education. The selection encompasses the most widespread protocols and consists of two OSPF networks (one single-area network and one multi-area network), a RIP network, and a BGP network featuring a basic peering between two routers. All these scenarios leverage FRRouting [19] as the routing suite. On average, the configurations for each device in each scenario consist of about 10 lines. We rely on this publicly accessible repository as it provides complete and accurate configurations for each network, which we can utilize later as the ground truth. For every scenario, we determined the specific configuration goals needed to successfully implement the solution. We then manually crafted prompts that detail the steps to achieve these goals. We asked the LLM (*i.e.,* GPT-4-Turbo in this case) to translate the goals into actual FRRouting configurations. Our goal is to determine whether the existing LLM knowledge suffices for correct translation and, if not, how much improvement is achieved with updated/extended knowledge. We are also interested in understanding *how* providing this extra knowledge impacts the translation. To do so, we repeated the same experiment in four different settings: (*i*) using existing knowledge (*i.e.,* no additional information); (*ii*) inputting the full FRRouting latest documentation in the prompt; (*iii*) interacting twice, by first inputting the table of contents of the documentation and asking for the section number of a specific protocol (*e.g.,* section number 3.3 refers the to BGP documentation), and second, inputting the corresponding section body text; and (*iv*) using Retrieval Augmented Generation (RAG)[11] [27] on the full FRRouting documentation. We are also interested in exploring the adaptability of LLMs in integrating new information without disrupting their existing knowledge base. This aspect is especially relevant when considering the implementation of new routing protocols or significant software updates that disruptively alter the syntax of a specific software. To achieve this, we included an additional network scenario, implementing a small Fat-Tree datacenter network, and we wrote a made-up FRRouting documentation for the RIFT protocol. This way, we aim to force the LLM to rely on this new & non-pre-existing knowledge, as the LLM would be unable to access information about the protocol from its pre-trained knowledge. For assessing the accuracy of each translation, we deploy a container equipped with FRRouting. This enables the loading of the generated configurations in the real daemon. Moreover, to make the comparison between the expected & generated configurations fairer, we use the `write terminal` command within FRRouting. This command reformats the configuration so that the lines are arranged in a consistent order, eliminating differences caused by the different positioning of configuration segments. Regarding the RIFT evaluation, due to the absence of an FRRouting implementation, we developed a manual formatter similar to the `write terminal` command, ensuring a consistent output. The formatted configurations are then compared using Python's `difflib` library, which can return a similarity score for a pair of input documents that ranges between 0 and 1. We repeat the experiment 5 times, showing the results in Figure 7. In the figure, we report the similarity ratio between the expected & generated configurations for the successful runs (*i.e.,* runs where no error occurred during the loading of the generated configuration; those with errors are omitted). The number of successful runs is shown with a label on top of the corresponding bar.

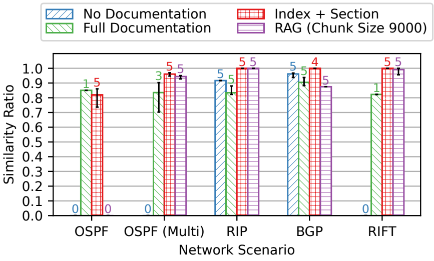


Fig. 7. Performance of GPT-4-Turbo in generating low-level configurations.

Takeaways. The results of our analysis demonstrate that:

- **Context window matters, again.** Similar to the findings in §3.1, we observe that LLMs are sensitive to the size of the context window. In fact, GPT-4-Turbo struggles to provide correct configurations when the full documentation is used as input (green bar). In this case, the volume of information to be processed is large, preventing the LLM from accurately pinpointing the relevant

[11] RAG splits input documents into smaller chunks, and appends the user prompt with the appropriate chunk via embedding similarity to enable LLMs to use external knowledge.

section needed to generate the correct commands. For example, in both OSPF scenarios, GPT-4 was only able to successfully generate error-free configurations in 1 and 3 runs, respectively. On the other hand, the two-step interaction (red bar) provides correct results in all scenarios. By providing a smaller document, GPT-4-Turbo can correctly identify the appropriate commands.

- **RAG is helpful in the configuration generation.** RAG is a well-known method for enhancing the knowledge of LLMs without the need for fine-tuning or additional training. We observed that the technique is also particularly effective in performing the configuration generation task. Indeed, the performance of RAG is also affected by the selected chunk size. In Figure 7, we only report the outcomes for a chunk size of 9000 tokens, as it exhibits the best performance among all the evaluated sizes. As it is possible to notice, RAG (purple bar) matches the performance of the two-step interaction (red bar) in almost all the scenarios. The only exception is the single-area OSPF scenario, where RAG consistently fails to produce successful configurations. Examining the logs, we noticed that the retrieved chunks were not relevant for OSPF configuration generation, leading to the same accuracy as the case when no documentation was provided (blue bar).
- **GPT-4-Turbo can take advantage of new & non-pre-existing knowledge without fine-tuning.** Focusing on the RIFT scenario, we notice that the LLM is able to correctly generate the configurations, based on our made-up documentation, using the two-step interaction method. When no documentation is provided, the model outputs fabricated information including commands that are a mixture of its pre-trained knowledge. A similar reasoning applies when providing the full documentation or using RAG, where GPT-4-Turbo struggles in identifying the commands due to the large context to process.

4 BUILDING LLM-BASED SYSTEMS FOR CONFIGURING NETWORKS

The last section showed that existing LLMs, especially GPT-4 and GPT-4-Turbo, are capable of performing network-configuration-related tasks. More specifically, our micro-benchmarking results from §3 can be summarized into the following principles that could help network developers design LLM-based systems for network configuration:

P1 Split complex tasks into smaller subtasks. Our findings suggest that a complex system should allow dividing tasks into smaller subtasks for two main reasons: (*i*) LLMs demonstrate higher accuracy when prompted with fine-grained tasks (see §3.1.1 and §3.4); and (*ii*) different tasks can be performed via different LLMs, depending on their performance, cost, and features (*e.g.,* GPT's function calling, see §3.2). Moreover, certain problems may *not* be effectively solved by LLMs (see §3.1.1); therefore, it should be possible to replace LLM engines with manually developed code, external tools, or external knowledge sources.

P2 Support task-specific verifiers. LLMs are prone to errors and hallucinations, particularly with large inputs or complex tasks. We emphasize, along with previous works [21, 29, 30], the importance of task-specific verifiers to guarantee generating error-free outputs. As shown in §3.3, combining LLMs with test cases improves code generation by minimizing failed attempts (and thus reducing overall generation costs). We also suggest that providing feedback, in case of failure, is instrumental for building a reliable and cost-effective LLM-based system. Nevertheless, a generic system should support several approaches (*e.g.,* formal verification) to validate the correctness of the generated output.

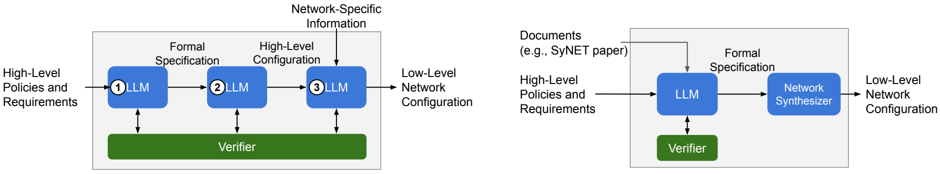
P3 Keep humans *still* in the loop. Although we demonstrated that it is possible to simplify certain network configuration tasks with the help of LLMs, current models are not entirely autonomous, and operators still need to carefully build prompts that take out the best performance from an LLM. §3.3 demonstrated the importance of prompting the LLMs with detailed, task-specific instructions when solving complex tasks. Therefore, we think that an LLM-based system should support human interactions, both in the form of prompting or manual feedback, at least as an optional feature. The OCE-centric design [21] closely aligns with our idea.

Relying on **P1—P3**, we present two real-world prototypes using LLMs to configure networks.

4.1 Using LLMs to Configure P4-Enabled Networks

This section presents a potential prototype of an LLM-based network configuration framework for a P4-based network. The system receives high-level policies and requirements (*i.e.,* specified in natural language) from network operators and translates them into low-level device-specific configurations. Following our proposed principles, we use a multi-stage pipeline for performing such a translation in multiple less-complex & fine-grained steps (see Figure 8a).

In particular, we divide the translation into three steps; each may be performed by a different LLM, depending on the needs. In this prototype, we focus on general-purpose LLMs (*i.e.,* GPT-4-Turbo), but a network operator can potentially use different (expert/specialized) LLMs for each task, *i.e.,* pre-trained or fine-tuned for a specific domain (*e.g.,* the fine-tuning example shown in §3.1). Besides, each LLM is paired with a verifier component that ensures the correctness of the output and provides feedback in case it detects errors/flaws (according to **P2**). We assume that network operators already have the machinery to deploy and/or run the generated low-level configuration on the network devices; therefore, we do not consider this step.


Network-Specific Information
Formal Specification
High-Level Configuration
High-Level Policies and Requirements
① LLM
② LLM
③ LLM
Low-Level Network Configuration
Verifier
Documents (e.g., SyNET paper)
Formal Specification
High-Level Policies and Requirements
LLM
Network Synthesizer
Low-Level Network Configuration
Verifier

(a) A stand-alone LLM-based system. (b) Integrating LLMs into existing synthesizers.

Fig. 8. Potential prototypes for using LLMs to configure networks, where the system translates high-level policies & requirements to low-level network configurations.

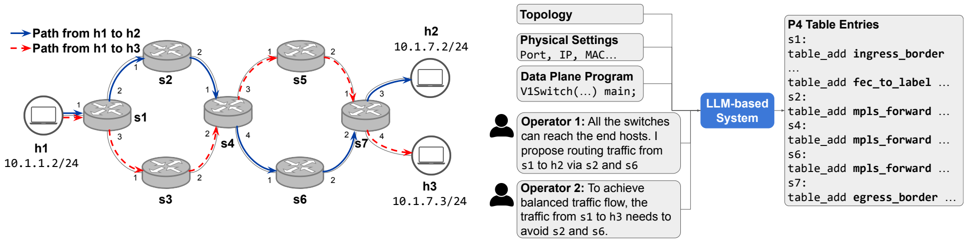
① **Generating formal specification.** The goal of this step is to convert the network operators' input in natural language into a machine-readable format. In our prototype, we employ the same formal specification format introduced in §3.1 as the output for this step (see Listing 1). It would be possible to include other information (*e.g.,* notifications/emails from a network monitoring service) in other formats (*e.g.,* the image/graph of the network topology). In this step, the verifier component checks the format and syntax of the generated output, and performs a conflict detection (see §3.1.1). Our evaluations in §3.1 have shown LLMs' capability of performing the translation with no error, *i.e.,* GPT-4-Turbo can achieve nearly 100% accuracy when the batch size is less than 20. Therefore, network operators could either trust the LLM's output or check the correctness of the output manually (according to **P3**).

② **Generating high-level configurations.** Before generating low-level network configurations, the system translates the formal specification into a high-level configuration, in the form of routing paths. In §3.3, we demonstrated GPT-4-Turbo's ability to generate network-related code. For our prototype, we employed the same algorithms to compute the routing paths within a network. In this step, a verifier can be used to (*i*) check the syntax, compilability/runnability, and correctness of the generated output and (*ii*) provide feedback to the LLM to correct the output in case of semantic errors. If the generated code is correct, we save and reuse it for future requests, since this is a costly and error-prone process. For instance, if an operator changes a specific requirement (*e.g.,* removes the connectivity between two nodes) without altering the type of requirements (*e.g.,* introducing traffic priorities), then we can re-use the already generated code and only provide the new formal specification as input. If an operator introduces a new type of requirement (*e.g.*, traffic priorities), an LLM would receive a description of the new requirement, along with the previous version of the code and various task-specific test cases. This enables the LLM to adapt the previous code to meet the new requirement, *rather than* developing it from scratch (according to **P1**). Note that it is also possible to perform this step without LLM by providing a manually developed code.

③ **Generating low-level configurations.** As the final step, the system generates low-level network configurations, as we discussed in §3.4. Similar to Step ②, the output of this step could be a script to generate a low-level configuration based on the routing information, and the verifier component can check the syntax, compilability/runnability, and correctness of the generated output and provide feedback if necessary. For instance, one can use existing verification tools (*e.g.*, SwitchV [2] to validate P4 control plane configurations) to ensure the correctness of the output. In the majority of the cases, the configuration, commands, and protocol specification may be known by a general-purpose LLM to generate the low-level network configuration. However, as shown in §3.4, sometimes LLMs can produce inaccurate results when no specific documentation is provided. Moreover, there are cases where networks use proprietary/undocumented devices and/or protocols, which are not included in LLMs pretrained knowledge (see the RIFT use case in §3.4). In the case of this prototype, we implement P4 switches using bmv2 [36], and the documentation for this software is already available in the existing knowledge base.

In the following examples, we demonstrate the effectiveness of our proposed prototype in configuring a P4-based network in implementing Multi-Protocol Label Switching (MPLS) routing and Equal-Cost Multi-Path (ECMP) load-balancing. We show the details of the latter example in Appendix B. In both examples, we assume that the P4 data plane program is already available[12] and focus on configuring P4 table entries. However, it is potentially possible to fine-tune an LLM to generate P4 data plane programs from scratch.

4.1.1 MPLS Routing for P4-Enabled Switches. Figure 9a shows the evaluated network topology, where three end-hosts (h1-h3) communicate with each other via seven switches (*i.e.*, s1-s7). We assume network devices are already configured with MAC and IP addresses. Next, we explain the details of each step to generate P4 table entries. Figure 9b shows the overview of our prototype.



Operator 1: All the switches can reach the end hosts. I propose routing traffic from s1 to h2 via s2 and s6

Operator 2: To achieve balanced traffic flow, the traffic from s1 to h3 needs to avoid s2 and s6.

```
s1:
table_add ingress_border
...
table_add fec_to_label ...
s2:
table_add mpls_forward ...
s4:
table_add mpls_forward ...
s6:
table_add mpls_forward ...
s7:
table_add egress_border ...
```

(a) Network topology.

(b) Input/Output of the system.

Fig. 9. An overview of the P4 prototype. The red and blue links show the two different routing paths.

Step ① Initially, we assume Operator 1 states two requirements: (*i*) full reachability and (*ii*) traffic engineering via waypoints. Our system uses GPT-4-Turbo to convert them into formal specifications with a format that includes reachability and waypoint, similar to Listing 1.

[12] GPT-4-Turbo is currently unable to generate entire P4 programs.

Step ② The LLM-based system receives the output of the previous step and the network topology to generate forwarding paths. We use the same data structure presented in Listing 2 for the high-level network configuration to specify the routing information. In this example, we use a Python script developed by GPT-4-Turbo to calculate the forwarding paths from the input information. In this example, the output of the script contains six forwarding paths. The path from h1 to h2, highlighted with blue in Figure 9a, satisfies the requirement of the waypoint.

Step ③ Our system sequentially provides (*i*) the already-deployed P4 program in the switches, (*ii*) a description of the expected script (*i.e.,* input and output data structure, the functionality of the script) to GPT-4-Turbo. To generate a successful script, this step still needs some human intervention. Then the system will run the generated script with (*i*) the topology, (*ii*) switches & hosts configurations (*e.g.,* MAC and IP addresses), and (*iii*) output of the previous step (*i.e.,* forwarding paths) to generate the P4 table entries.

In this example, our LLM-based system generates P4 table entries to configure switches for all the forwarding paths between hosts. For instance, for the blue path, the final P4 program performs the following tasks: s1 checks whether the incoming packet originates from h1; if so, it adds a MPLS header to the packet and forwards it to s2. Then, s2, s4, s6, and s7 perform MPLS forwarding based on the attached label. Finally, s7 removes the MPLS label and transmits the packet to h2. To evaluate the correctness & runnability of the generated configurations, we emulate the same network using the Kathará network emulator [42]. Our testbed uses the output configuration and automatically installs the generated table rules on the P4-enabled switches running in Kathará. To verify the correctness of our system, we manually test the requirements via ping and tcpdump; however, the verification can potentially be automated.

Introducing a new requirement. All forwarding paths, except the blue one, are selected simply based on the shortest path policy. As the second step, we assume Operator 2 introduces a new requirement to improve link utilization and distribute the load more efficiently. To support the new requirement category, the operator adds a new entry called "avoidance" to the specification format. Instead of generating everything from scratch, our system re-uses the already-existing configurations to enforce the new requirement. In particular, in Step ②, our system modifies the previously developed Python script via GPT-4-Turbo to take into account the newly introduced requirement; this step is done similarly to §3.3 by providing detailed instructions and feedback. The new script calculates new forwarding paths based on the newly-introduced requirement, which forces the traffic from h1 to h3 to go through the red path shown in Figure 9a. All other steps are performed in the same way as before.

4.2 Integrating LLMs with Existing Tools

The last subsection showed the possibility of building tailored systems from scratch using LLMs to configure networks. However, some operators may have already invested in developing proprietary tools to configure networks, which may not be included in the pre-trained knowledge of existing LLMs. While fine-tuning LLMs is possible, we explore the possibility of relying on documentation of proprietary tools to integrate LLMs into the existing network configuration pipeline.

As an example, we focus on facilitating the usage of network synthesizers that typically rely on specialized languages/specifications to define requirements and policies, imposing a learning burden on network operators. We suggest attaching an LLM to network synthesizers to provide a more intuitive, user-friendly interface, which can automatically translate high-level requirements specified in natural human language into languages/specifications used by network synthesizers. Figure 8b shows our proposed design for such a system, where an LLM can (*i*) understand & learn the specialized language or specification used by the network synthesizer; and then (*ii*) translate the network operators' requirements expressed in natural language, into the synthesizer's formal language. For instance, SyNET [15] utilizes the stratified Datalog query language to articulate the behavior of routers and their protocols. Our proposed system enables network operators to simply articulate their network requirements in natural language and then rely on an LLM to do the translation, rather than becoming proficient in stratified Datalog. Later, the translated output can be provided as input to the SyNET synthesizer to generate the appropriate configurations, thereby bridging the gap between human-friendly communication and technical specifications.

In our example, ① we ask GPT-4-Turbo to formulate a few network requirements in natural language, extracted from examples available in the SyNET repository [31]; ② we provide to GPT-4-Turbo (i) the SyNET paper [15] and the generated network requirements from the last step and (ii) ask the LLM to generate the constraints supported by SyNET (based on the provided inputs); ③ we compare original constraints in the examples with the generated one.

Our experiments indicate that GPT-4-Turbo can successfully grasp the idea of the SyNET paper and convert the network requirements into the desired specifications. For example, the requirements "*Traffic classified as* `R2_N1` *should be routed from* `R1` *to* `R3`*, and then from* `R3` *to* `R2`*, exclusively using the OSPF protocol*" are interpreted and translated into the low-level specifications `Fwd(R2_N1, R1, R3, ospf)` and `Fwd(R2_N1, R3, R2, ospf)`, respectively. However, we observed that GPT-4-Turbo could generate some constraints differently from the expected ones. This is due to a mismatch between the SyNET paper and its actual software implementation (see Appendix C). This again confirms the importance of providing LLMs with up-to-date technical documentation (see §3.4). Furthermore, the translation process could require some human involvement if the generated configurations contain errors (see **P3**).

5 CONCLUSION

ML-based approaches have been investigated for automatically generating code from natural language. The rise of LLMs, such as GPT-4, has provided a means to develop AI-assisted tools. While multiple benchmarks & datasets have been proposed to evaluate LLMs capabilities for different tasks such as programming, the same phenomenon has not yet taken place in the networking community. In this paper, we presented a benchmark suite to efficiently examine various LLMs capabilities to improve the tasks of network configuration. Additionally, we propose two *functional* prototypes to deploy LLMs for network configuration. Our main takeaway is that LLMs can dramatically simplify and automate complex network management tasks. We hope our work motivates more research on employing AI techniques on network management tasks. Future iterations of our benchmark could (i) enhance complexity by incorporating additional policies, implementing more sophisticated and distributed routing algorithms, and creating advanced configuration generation tasks; and (ii) explore the impact of different task decomposition strategies or applying LLMs in network policy mining. Furthermore, it would be beneficial to organize a study involving small network operators and systematically evaluate the advantages and improvements in network management achieved through the utilization of LLM-based systems.

6 ACKNOWLEDGEMENTS

We would like to thank our shepherd and the anonymous reviewers for their insightful comments and suggestions on this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889). This work has been partially supported by Vinnova (the Sweden's Innovation Agency), the Swedish Research Council (agreement No. 2021-04212), and KTH Digital Futures. We acknowledge EuroHPC Joint Undertaking for awarding us access to the Leonardo supercomputer located at the CINECA data center in Bologna, Italy.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[2] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: Automated SDN Switch Validation with P4 Models. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 365–379. https://doi.org/10.1145/3544216.3544220

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[4] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. 2023. NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches. *IEEE Communications Magazine* (2023), 1–7. https://doi.org/10.1109/MCOM.001.2300313

[5] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. PaLM 2 Technical Report. arXiv:2305.10403 [cs.CL]

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network Configuration Synthesis with Abstract Topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 437–451. https://doi.org/10.1145/3062341.3062367

[9] Ryan Andrew Beckett. 2018. *Network Control Plane Synthesis and Verification.* Ph. D. Dissertation. Princeton University. http://arks.princeton.edu/ark:/88435/dsp01d217qs28v

[10] Luca Beurer-Kellner, Martin Vechev, Laurent Vanbever, and Petar Veličković. 2022. Learning to Configure Computer Networks with Neural Algorithmic Reasoning. In *Advances in Neural Information Processing Systems 35*, Sanmi Koyejo, Shakir Mohamed, Alekh Agarwal, Danielle Belgrave, Kyunghyun Cho, and Alice Oh (Eds.). Curran, Red Hook, NY, 730 – 742. 36th Annual Conference on Neural Information Processing Systems (NeurIPS 2022); Conference Location: New Orleans, LA, USA; Conference Date: November 28 - December 9, 2022; Poster presentation on December 1, 2022..

[11] Rudiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 969–984. https://www.usenix.org/conference/nsdi20/presentation/birkner

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford,

Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[15] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 261–281.

[16] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 579–594. https://www.usenix.org/conference/nsdi18/presentation/el-hassany

[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[18] Kathará Framework. 2023. Kathara-Labs: Collection of Kathará Network Scenarios. https://github.com/KatharaFramework/Kathara-Labs Accessed 2023-11-29.

[19] FRRouting. 2023. FRRouting. https://frrouting.org/ Accessed 2023-11-29.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[21] Pouya Hamadanian, Behnaz Arzani, Sadjad Fouladi, Siva Kesava Reddy Kakarla, Rodrigo Fonseca, Denizcan Billor, Ahmad Cheema, Edet Nkposong, and Ranveer Chandra. 2023. A Holistic View of AI-driven Network Incident Management. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 180–188. https://doi.org/10.1145/3626111.3628176

[22] Chase Harrison. 2022. LangChain. https://github.com/langchain-ai/langchain Accessed 2023-11-29.

[23] Lorin Hochstein and Rene Moser. 2017. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* O'Reilly Media, Inc., Sebastopol, California.

[24] Zied Ben Houidi and Dario Rossi. 2022. Neural language models for network configuration: Opportunities and reality check. *Computer Communications* 193 (sep 2022), 118–125. https://doi.org/10.1016/j.comcom.2022.06.035

[25] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu

[26] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. 2018. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks* (Budapest, Hungary) *(SelfDN 2018)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3229584.3229590

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL]

[28] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[29] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 196–204. https://doi.org/10.1145/3626111.3628183

[30] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What Do LLMs Need to Synthesize Correct Router Configurations?. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/3626111.3628194

[31] Networked Systems Group (NSG) - ETH Zürich. 2017. GitHub - SyNET. https://github.com/nsg-ethz/synet Accessed 2023-12-02.

[32] Nokia. 2023. SR Linux and ChatGPT combine for network AIOps. https://www.nokia.com/blog/sr-linux-and-chatgpt-combine-for-network-aiops Accessed 2023-11-29.

[33] OpenAI. 2023. Function calling. https://platform.openai.com/docs/guides/function-calling/parallel-function-calling Accessed 2023-11-29.

[34] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[35] OpenAI. 2023. Models. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo Accessed 2023-11-29.

[36] p4lang. 2023. Behavioral Model (bmv2). https://github.com/p4lang/behavioral-model Accessed 2023-11-29.

[37] Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. AI-assisted coding: Experiments with GPT-4. arXiv:2304.13187 [cs.AI]

[38] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 953–967. https://www.usenix.org/conference/nsdi20/presentation/prabhu

[39] Tony Przygienda, Alankar Sharma, Pascal Thubert, Bruno Rijsman, Dmitry Afanasiev, and Jordan Head. 2023. *RIFT: Routing in Fat Trees*. Internet-Draft draft-ietf-rift-rift-21. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-rift-rift/21/ Work in Progress.

[40] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. 2023. Practical Intent-driven Routing Configuration Synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 629–644. https://www.usenix.org/conference/nsdi23/presentation/ramanathan

[41] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[42] Mariano Scazzariello, Lorenzo Ariemma, and Tommaso Caiazzi. 2020. Kathará: A Lightweight Network Emulation System. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Budapest, Hungary, 1–2. https://doi.org/10.1109/NOMS47738.2020.9110351

[43] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 41–47. https://doi.org/10.1145/3626111.3628205

[44] Kumar Shridhar, Koustuv Sinha, Andrew Cohen, Tianlu Wang, Ping Yu, Ram Pasunuru, Mrinmaya Sachan, Jason Weston, and Asli Celikyilmaz. 2023. The ART of LLM Refinement: Ask, Refine, and Trust. arXiv:2311.07961 [cs.CL]

[45] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables in Multi-Tenant Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 572–585. https://doi.org/10.1145/3009837.3009845

[46] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. https://doi.org/10.1145/3341302.3342088

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[48] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[49] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. `NetConfEval` GitHub Repository. https://github.com/NetConfEval/NetConfEval Accessed 2024-04-23.

[50] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. `NetConfEval` HuggingFace Repository. https://huggingface.co/datasets/NetConfEval/NetConfEval Accessed 2024-04-23.

[51] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. 2023. Making Network Configuration Human Friendly. arXiv:2309.06342 [cs.NI]

[52] Will Knight. 2023. OpenAI's CEO Says the Age of Giant AI Models Is Already Over. https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/ Accessed 2023-12-05.

[53] Shirley Wu. 2023. Move Naturally and Rule the Network with ChatGPT. https://blogs.juniper.net/en-us/enterprise-cloud-and-transformation/move-naturally-and-rule-the-network-with-chatgpt Accessed 2023-11-29.

[54] Shijie Xia, Xuefeng Li, Yixin Liu, Tongshuang Wu, and Pengfei Liu. 2024. Evaluating Mathematical Reasoning Beyond Accuracy. arXiv:2404.05692 [cs.CL]

[55] Yu Zhou, Xingyu Wu, Beicheng Huang, Jibin Wu, Liang Feng, and Kay Chen Tan. 2024. CausalBench: A Comprehensive Benchmark for Causal Learning Capability of Large Language Models. arXiv:2404.06349 [cs.LG]

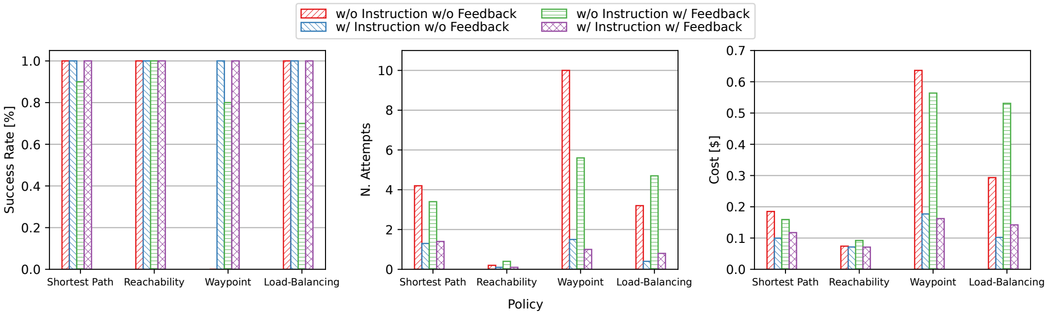
A SUPPLEMENTARY MATERIAL FOR CODE GENERATION

This section provides supplementary material for code generation experiments; see Listing 3, Listing 4, and Figure 10.

Listing 3. An example of algorithmic instructions to generate a code for finding the shortest paths.

Instructions for finding the shortest path:

(1) Construct a graph from the topology;
(2) Identify the unidirectional host pairs;
(3) Find all possible paths for each host pair, and rank the paths according to their length;
(4) Pick the path that strictly satisfies all the requirements related to the switches – if no path is found, set the path to empty;
(5) Return final routing paths for all host pairs.



(a) Success Percentage. (b) Number of Attempts. (c) Cost.

Fig. 10. The results of code generation using GPT-4.

Listing 4. An example of a successfully generated code for the shortest path policy.

```
def compute_routing_paths(topo, requirements=None):
    import itertools
    import networkx as nx
    # Construct a network graph from the topology
    G = nx.Graph()
    for device, connections in topo.items():
        for port, lan in connections.items():
            G.add_edge(device, lan)

    # Identify the unidirectional host pairs based on the network topology
    hosts = [device for device in topo.keys() if device.startswith('h')]
    host_pairs = list(itertools.permutations(hosts, 2))

    # For each host pair, find all possible paths
    paths = {}
    for host1, host2 in host_pairs:
        all_paths = list(nx.all_simple_paths(G, source=host1, target=host2))
        all_paths.sort(key=len)  # sort paths from shortest to longest
        # For each pair, pick the shortest path satisfying the requirements
        for path in all_paths:
            switches_in_path = [node for node in path if node.startswith('s')]

            # Check reachability requirements
            if not all(host2 in requirements.get('reachability', {}).get(switch, []) for switch in
            ↪  switches_in_path):
                continue

            shortest_path = path
            break
        else:
            shortest_path = []  # no path found

        shortest_path = [node for node in shortest_path if node.startswith('h') or
        ↪  node.startswith('s')]

        if host1 not in paths:
            paths[host1] = {}
        paths[host1][host2] = shortest_path

    return paths
```

B ECMP LOAD BALANCE FOR P4-ENABLED SWITCHES

In this scenario, we show an example of ECMP load balancing via P4-enabled switches based on our proposed network generator. It receives the same information as input, and the data plane program implements ECMP (see Figure 11).

Step ① We assume a new operator states the following requirements: h1 should send traffic to h2 through 4 different paths. Our LLM-based system converts them into a formal specification.

Step ② The system receives the formal specification and the network topology to generate multiple forwarding paths. Our prototype interacts with GPT-4-Turbo to generate the corresponding load balancing algorithm, as described in §3.3.

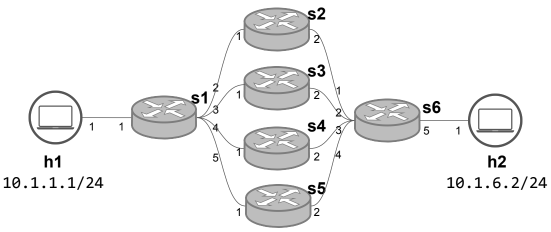


Fig. 11. ECMP Network topology.

Step ③ The multiple forwarding paths generated by Step ② are fed into GPT-4-Turbo, along with other required information. The LLM will then generate the corresponding P4 table entries. In `s1`, the flow 5-tuple identifier will be hashed and the packet will be sent to the appropriate output port.

C MISMATCH BETWEEN SYNET PAPER AND REAL IMPLEMENTATION

The requirements "*Traffic classified as R2_N1 should be routed from* `R1` *to* `R3`*, and then from* `R3` *to* `R2`*, exclusively using the OSPF protocol*" will sometimes be translated by GPT-4-Turbo into low-level specifications `Path('R2_N1', 'R1', ['R1', 'R3', 'R2'])`, which is different from the expected answer `Fwd(R2_N1, R1, R3, ospf), Fwd(R2_N1, R3, R2, ospf)`. However, this is due to a mismatch between the SyNET paper and its actual software implementation. The paper implies that a higher-level constraint `Path(TC, R1, [R1, R2,..., Rn])` could be specified as conjunction over the `Fwd(TC, Router, NextHop)` constraint, and translation from the `Path` constraint to `Fwd` constraint could be supported in the backend to make it more user-friendly (*i.e.,* users can write their requirements in a high-level way). However, the backend translation is not realized and the `Path` constraint is not supported in the real implementation. A similar problem arises when we ask explicitly for `Fwd` predicates. The translation answer will sometimes be `Fwd('R2_N1', 'R1', 'R3'), Fwd('R2_N1', 'R3', 'R2')`, as the implementation supports `Path(TC, Fwd(TC, Router, NextHop, Protocol))` format but the paper does not mention it. However, the wrong results generated above follow the format of the paper and are logically translated correctly.

Received December 2023; revised January 2024; accepted March 2024