PINC (P4 and Intent for Network Configuration): A Prototype Using Large Language Models

Alyas (Illia) Nasiri
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
ina14@sfu.ca

Kunwar Deep Singh Bakshi
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
kdb5@sfu.ca

Gwen Yang
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
gya21@sfu.ca

Kangyi Zhang
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
kza73@sfu.ca

Ziyan Chen
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
chen.ziyan2@northeastern.edu

Manya Sharma
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
msa261@sfu.ca

Ouldooz Baghban Karimi
School of Computing Science
Simon Fraser University
Surrey, BC, Canada
ouldooz@sfu.ca

***Abstract*—Enabled by recent advances in code generation using large language models (LLMs), the increasing data-plane programmability, and expanded compute capabilities at the network edge, we experiment intent-driven network management automation by leveraging an LLM fine-tuned for $P4_{16}$ data-plane programming coupled with a verification pipeline. Our verification pipeline includes early rejection, logic and functional checks, and runtime performance validation. Our contributions are threefold. First, we present the design of PINC, an intent-driven automation framework that maps high-level user intents to verifiable $P4_{16}$ data-plane configurations. Second, we develop a minimal prototype and evaluate the performance of a fine-tuned LLM, augmented with an intent classifier and a few-shot prompting wrapper, on $P4_{16}$ code-generation tasks representative of data-plane configuration workflows. We reach up to 98% compilation rate of the generated code, and 48-100% success in solution generation. Third, we release our curated dataset of 405 cleaned, license-compliant, and annotated $P4_{16}$ programs to support future research on LLM-driven data plane programming. Our work takes a step towards generating coherent AI-driven network behaviors by combining generation and validation for reliable intent-based configuration with minimal human supervision.**

I. Introduction

The growing complexity of networks has underscored the need for network management automation. The Intent-based Networking (IBN) framework [1] tries to address this challenge through mechanisms for translation of high-level, natural-language intents to low-level configurations. However, the difficulty of translating intents into correct and complete configurations has hindered the widespread adoption of IBN.

Recent advances in Large Language Models (LLMs) have demonstrated promising capabilities in generating code from high-level natural-language instructions. In the meantime, networks have evolved toward programmability across the control and data planes. Software-Defined Networking (SDN) and Network Function Virtualization (NFV) have enabled programmable control-planes while P4 (Programming Protocol-Independent Packet Processors) [9], among similar solutions, has brought programmability, and the possibility of automation, to the data plane. Together, these developments show a promising path for reinvigorating the research on IBN.

We consider automating P4-enabled generalized switches across access networks with LLMs and IBN framework. While language models show promising early code generation results, the quality of the output depends on the data they are trained on. There are available LLMs trained on various programming languages. However, the quality of the generated code is reported low [10], especially on $P4_{16}$ code, due to training-data scarcity. Correctness of the generated P4 configuration codes, verifiability of the configurations, target constraints, performance testing, and complete end-to-end automation of the tasks still remain challenging problems.

Our contributions are three-fold: First, we propose a system (PINC) which uses IBN framework with LLMs to generate P4, automating network configuration. Second, using our publicly available dataset, we fine-tune and use an LLM and couple it with a wrapper that utilizes an intent classifier and a few-shot prompting strategy, and build a prototype that shows promising results as a minimum viable solution for an automated network management tool. Third, to fine-tune the engine of our system as well as close dataset availability gap for the community, we release our developed dataset of cleaned, license compliant, annotated P4 codes that could be used to fine-tune language models generating P4 code. This supports adoption and future research on LLM-driven data plane programming[1]. Our experiments with the PINC prototype generated P4 code shows up

This work is supported by an NSERC Discovery Grant, and Google exploreCSR 2024 program for broadening participation in research.

ISBN 978-3-903176-82-9

[1]PINC engine code and our fine-tuning code are available on: PINC GitHub: *https://github.com/ouldoozbk/PINC/*. Dataset and fine-tuned model weights are available on SFU Connect Hugging Face: *https://huggingface.co/SFU-CONNECT*

TABLE I
LLM BASED NETWORK CONFIGURATION SOLUTIONS

Solution	Application	P4	I	D	HV	Scope & Limitations
LLNet [2]	Intent to Code SLM	*	✓	–	✓	Evaluations on three scenarios: firewall, rate limiter, load profiling
P4 dataplane [3]	LLM/SLM for P4	✓	–	–	✓	Code generated on modified training problem statement (P4tutorials)
LLM-NetCFG [4]	Config Generation	–	✓	–	✓	Evaluated on classification of intents; observed hallucinations
P4TestGen [5]	Test oracle for P4	✓	–	–	–	Automatic test generation for P4 targets
NetConfEval [6]	Benchmark	✓	✓	–	✓	Exploratory prototype emphasizing unreliability of LLMs
VPP [7]	Config Generation	–	–	–	✓	Network configuration generation and translation among devices
IBN-LLM [8]	IBN using LLMs	–	✓	–	✓	Configurations of network functions; potential hallucinations
PINC	LLM/SLM for P4	✓	✓	✓	*	Limited verification in the current prototype

Column **P4** indicates whether the P4 code generation quality evaluated in the paper. Column **I** indicates whether the study uses Intent-based Networking framework. Column **D** indicates whether the dataset is publicly available. Column **HV** indicates if the study uses Human Validation as a step in the process. ✓: Yes, **–**: No, *****: Available as an option

to 91% compilability. Among the compilable code, 48-100% passed all formal-verification test cases with 100 generated samples per problem, indicating a possible solution.

This makes PINC a step toward generative AI network management, where an LLM-based system produces verified data-plane configurations from user's high-level intent and network state.

We review related work in section II. We introduce our system design and implemented prototype in sections III and IV. We present our initial results in section V. We discuss threats to validity in section VI and conclude our work and discuss future directions in section VII.

II. BACKGROUND

P4 [11] aims to achieve reconfigurability, as well as protocol and target independence [9] in an attempt to facilitate SDN control plane communication with programmable switches. Generating the P4 code is influenced by control plane protocols [9] as well as network applications directly controlling the programmable switches through application policies [12].

LLMs' success in natural language understanding has inspired approaches that translate high-level intents into network functions to reduce manual configuration. Attempts to configure network elements using LLMs [7], and implementation of different types of network functions and algorithms such as NetLLM [13], NADA [14] are examples of LLM usage for network management. Other works, such as NetConfEval [6] try to evaluate such LLM-based solutions and provide a set of principles to integrate LLMs in network management.

Promising results with LLMs in other networking tasks have led to the reconsideration of Intent-based solutions [15] with P4, leveraging the promise of LLMs [2]. Challenges of generating P4 code using LLMs include limited data available for training. While solutions have been proposed [3], no dataset is publicly made available. Verification of the generated code also largely remains a challenge [5]. Another problem in intent to P4 code is virtually unlimited alternative natural language expressions for the same high-level intents. Prompt optimization and partitioning works for network management [16], [17] propose solutions, applicable to the IBN context.

Previous work evaluated the ability to generate P4 code using a benchmark of modified tutorial and snippet-completion tasks [3]. Their evaluation used the `P4c` compiler for testing compilation of generated code, and unit testing for 11 of 50 task categories to assess functional correctness. Their results showed that fine-tuned small models (1B-3B parameters) pre-trained in P4 can outperform larger commercial models like ChatGPT-4 on P4 code completion tasks, achieving compilation success rates of 60% compared to 33% for general-purpose LLMs. However, their evaluation depends on manually constructed tasks and hand-written unit tests; functional testing is limited to a subset of tasks rather than general intents. Table I summarizes related experimental research.

III. SYSTEM ARCHITECTURE

PINC consists of three main components: *Intent Intake*, *Code Generation*, and *Verification*, as illustrated in Figure 1. The goal of the system is to translate a high-level natural language network intent into verified code that can be deployed in programmable data planes.

An intent is a declarative, high-level statement describing a desired network function, for example, *"Enable source MAC filtering"*. Because such intents do not specify low-level implementation details, PINC expands them into a richer representation before generation. The system incorporates network rules, device and interface information, and current configuration state so that the generated code is both task-relevant and consistent with the existing environment.

A. Intent Intake

Intent Intake bridges the gap between user intent and the detailed information required for $P4_{16}$ generation. Starting from a natural language description, this module constructs a structured prompt that combines the requested network behavior with relevant network context, including configuration schemas and current state. The output is a context-rich prompt that serves as input to the code generation engine.

B. Code Generation

Code Generation is the core engine of PINC. Given the structured output from Intent Intake, the generation model produces $P4_{16}$ code intended to implement the requested behavior while respecting the current network environment. This stage is responsible only for producing candidate code and does not itself validate correctness.

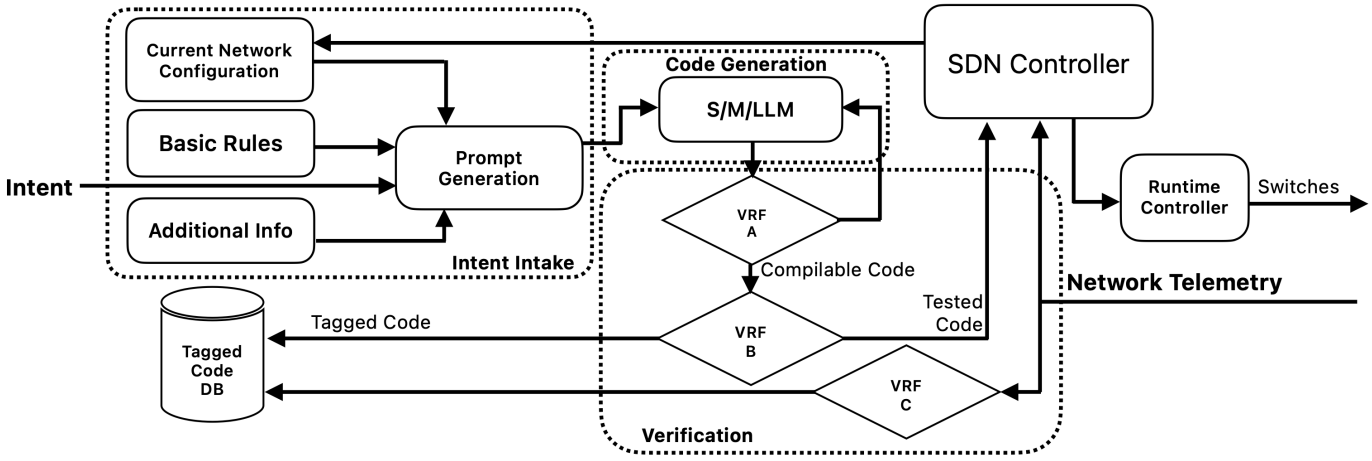

Fig. 1. System Architecture, including three main modules: (1) Intent Intake, (2) Code Generation, and (3) Verification. The verification pipeline includes VRF A (Early Rejection), VRF B (Logical and Functional Verification), and VRF C (Runtime and Performance Verification).

C. Verification

Generated code must be validated before deployment. PINC therefore includes a multi-stage verification pipeline. VRF A performs early rejection by checking syntax and compilation. VRF B evaluates functional and logical correctness through automated testing. VRF C, which is part of the full PINC design but not fully implemented in the prototype, is intended to assess runtime and performance behavior after deployment. Together, these stages filter candidate programs and improve reliability before execution in the network.

IV. Prototype Implementation

To evaluate the feasibility of PINC, we built a minimum viable prototype implementing the first two stages of the system pipeline: intent-to-code generation and pre-deployment verification. The prototype includes a prompt-construction wrapper, an LLM-based $P4_{16}$ generation engine, a lightweight intent classifier for example selection, and two verification stages: compilation checking and functional testing.

A. Intent Intake

In the prototype, Intent Intake is implemented as an LLM wrapper that converts a user's high-level intent into a structured prompt for code generation. The wrapper begins with the user's natural language request and augments it with additional context, including current network configuration data and relevant YANG schemas describing configuration structure. These inputs are combined into a single prompt consumed by the downstream generation model.

1) Prompt Generation: The wrapper constructs prompts by combining four elements: (1) the user intent, (2) current network parameters (YANG), (3) current configuration (JSON), and (4) optional in-context examples. This structured prompting allows the generation model to condition on both the requested behavior and the surrounding network context.

2) Prompting Strategy: To improve generation quality, we used few-shot prompting in the wrapper. The prompt includes concrete examples mapping network intents and configuration context to valid $P4_{16}$ implementations. These examples provide patterns for syntax, structure, and task formulation, helping the model produce more compilable and task-appropriate code. This follows the in-context learning paradigm [18]. [2]

3) Classification: We also implemented an intent classification module to dynamically select few-shot examples at inference time. The goal was to test whether choosing examples that match the user's intent category improves code generation quality.

Our classifier is a Linear SVM trained on TF-IDF features. To create a training set, we used Gemini 2.5 Pro [19] to generate a synthetic dataset of 958 English utterances describing networking tasks derived from the intent categories in our dataset (Section IV-B2). The data spans 12 categories, including *Basic Tunneling*, *ECN*, *Firewall*, and *QoS*, and is approximately balanced across classes. We split the dataset 75/12.5/12.5 into train/validation/test. During inference, the predicted category is used to select the corresponding few-shot examples. This improved the compilation rate of generated $P4_{16}$ code from 76% and 74% to 98% and 80% respectively for testers A and B.

B. Code Generation

The code generation engine takes the structured prompt from the Intent Intake and produces candidate $P4_{16}$ programs. Our prototype implements this stage using an instruction-tuned language model.

1) Choice of LLM: We selected DeepSeek-Coder-V2-Lite-Instruct as the main generation engine for the prototype. Large language models can achieve strong code generation performance, but very large models are not desirable in realistic network settings. Prior work shows that efficient deployment

[2]More information in the Appendix available on: *https://osf.io/86cqb/*

of very large models on edge or constrained hardware often requires model partitioning, specialized kernels, or aggressive quantization [20], [21]. By contrast, medium-sized models in the 1B–20B range offer a more practical tradeoff between generation quality and deployability [22].

This motivated our choice of DeepSeek-Coder-V2-Lite-Instruct with 16B parameters. The model is sufficiently capable for domain-specific code generation while remaining deployable on accelerators such as an NVIDIA A100 40GB. This makes it suitable for low-latency, privacy-preserving, and locally hosted $P4_{16}$ generation.

2) P4 Dataset: Our dataset consists of 405 programs scraped from public GitHub repositories. We resolved intra-repository dependencies, de-duplicated to avoid repetitive examples, filtered by permissive license, restricted to $P4_{16}$ version, and validated via compilation. [3] Each example contains cleaned $P4_{16}$ source code, the intent, the original raw source code prior to dependency resolution and comment preprocessing, inferred license information with its method of inference, and repository metadata such as project description, and file path. Each $P4_{16}$ program is also annotated with a high-level task description using an LLM. Refer to the Appendix for details of the collection and filtering pipeline.[4]

3) Fine-tuning PINC Engine: We instruction-tuned [23] DeepSeek-Coder-V2-Lite-Instruct [24] on the collected $P4_{16}$ dataset, reducing test perplexity from 1.784961 to 1.501949.

Training examples were constructed as prompt-code pairs in a causal left-to-right format aligned with the model's native chat template. Each example contained three semantic parts: a fixed system message describing the task, a user message containing the high-level annotation, and an assistant prefix indicating the start of code generation. We introduced special tokens `<P4>` and `</P4>` to mark the beginning and end of code output. These tokens were added to the tokenizer vocabulary, and the model's embeddings were resized accordingly.

We capped the combined prompt and code length at 4096 tokens. Most examples fit within this limit. For longer examples, the code was split into multiple chunks while reusing the same prompt prefix for each chunk.

C. Verification

1) VRF A: VRF A provides an initial validation stage based on iterative compilation feedback. After the model generates a candidate output, the system extracts only the $P4_{16}$ code and discards additional explanatory text. The extracted code is then compiled using `P4c` with `--target bmv2 --arch v1model` to check both syntactic and semantic validity against the $P4_{16}$ language and BMv2 reference architecture.

If compilation fails, the compiler errors are fed back into the prompt together with the previously generated code, and the model is given another attempt. This loop continues up to a fixed maximum number of attempts. VRF A therefore acts as an early rejection and refinement stage, ensuring that only compilable candidate programs proceed further in the pipeline.

[3]The dataset contains public code with permissible licenses.

[4]More information in the Appendix available on: *https://osf.io/86cqb/*

TABLE II
COMPILATION RATES FOR ZERO-SHOT (ZS), FEW-SHOT (FS), AND CHAIN-OF-THOUGHT (COT) PROMPTING

	Test (A)			Test (B)		
	ZS	CoT	FS	ZS	CoT	FS
Base Model	0%	24%	36%	0%	6%	10%
Fine-Tuned Model	2%	54%	76%	6%	66%	74%
Base + Classifier	–	–	41%	–	–	65%
FTM + Classifier	–	–	98%	–	–	80%

TABLE III
PASS@K (N=100) FOR FINE-TUNED AND BASE MODEL USING FEW-SHOT (FS) PROMPTING

	Test (A)		Test (B)	
	Pass@1	Pass@10	Pass@1	Pass@10
Base Model	34.51%	60.43%	45.76%	79.90%
Fine-Tuned Model	84.24%	95.98%	91.41%	99.89%
Base + Classifier	24.02%	43.87%	24.88%	43.90%
FTM + Classifier	48.78%	73.17%	91.70%	100.0%

2) VRF B: VRF B evaluates the functional correctness of compilable $P4_{16}$ programs. In our prototype, this stage uses P4TestGen [5] together with BMv2. P4TestGen automatically generates test cases by exploring execution paths of the candidate $P4_{16}$ program and using SMT-based reasoning to derive packets that trigger specific behaviors. BMv2 then executes these tests in a software switch environment.

This allows the system to move beyond compilation success and assess whether a candidate program behaves correctly under generated test scenarios. VRF B forms the basis for the pass@k evaluation [25] used in our experiments.

3) VRF C: VRF C is part of the full PINC architecture but was not implemented in the current prototype. Its role is to evaluate runtime and performance properties after deployment, such as latency and resource behavior in deployment.

V. EXPERIMENTS AND RESULTS

We evaluated the prototype along two dimensions: (1) compilation success of generated $P4_{16}$ programs and (2) functional correctness measured using pass@k. Results are reported under zero-shot (ZS), chain-of-thought (CoT), or few-shot (FS) prompting, using either the base model, the fine-tuned model, or classifier-assisted prompt selection.

A. Compilation Rate

We measured how often generated code compiled under different prompting conditions. Table II compares zero-shot (ZS), chain-of-thought (CoT), and few-shot (FS) prompting for the base and fine-tuned models, with and without the classifier.

Across both test settings, few-shot prompting substantially outperformed zero-shot prompting, and fine-tuning provided an additional large gain. For example, the fine-tuned model improved from 2% to 76% on Test A and from 6% to 74% on Test B when moving from zero-shot to few-shot prompting. Adding the classifier further improved few-shot compilation rates, reaching 98% on Test A and 80% on Test B.

TABLE IV
COMPILATION RATES FOR VARIOUS BASE MODELS USING ZERO-SHOT (ZS) AND FEW-SHOT (FS) PROMPTING

	Test (A)		Test (B)	
	ZS	FS	ZS	FS
Qwen 3 Max	45%	78%	24%	85%
Llama 4 Maverick	71%	86%	65%	83%
GPT-5 Mini	41%	71%	56%	87%

TABLE V
PASS@1 (N=10) FOR BASE, FINE-TUNED, AND ALTERNATIVE MODELS USING FEW-SHOT (FS) PROMPTING

	Test (A)	Test (B)
Base Model	36.09%	48.04%
Fine-Tuned Model	81.21%	58.04%
Qwen 3 Max	43.90%	13.41%
Llama 4 Maverick	68.29%	58.78%
GPT-5 Mini	7.31%	9.24%

These results suggest that all three factors matter: domain fine-tuning, example-based prompting, and intent-aware example selection. Zero-shot prompting alone is not sufficient for reliable $P4_{16}$ generation in this low-resource domain.

B. Pass@k Functional Evaluation

Compilation alone does not guarantee correct packet-processing behavior, so we next evaluated functional correctness using pass@k. For each prompt, we generated multiple compilable candidates, used P4TestGen to create test suites, and executed those tests in BMv2. Table III reports pass@1 and pass@10 for the base model, fine-tuned model, and classifier-assisted variants.

The fine-tuned model consistently outperformed the base model. In Test A, pass@1 increased from 34.51% to 84.24%, and pass@10 rose from 60.43% to 95.98%. In Test B, the fine-tuned model achieved 91.41% pass@1 and 99.89% pass@10, indicating that correct programs were generated very reliably within a small number of samples.

The classifier-assisted setting showed mixed behavior. In Test B, the fine-tuned model with classifier slightly improved over the non-classifier version at pass@1 and reached 100% pass@10. In Test A, however, classifier-assisted pass@k was lower. This suggests that intent-based example selection can help, but its effectiveness depends on prompt composition and category-example alignment.

C. Alternative LLMs

To test whether our prompt wrapper and few-shot strategy generalize across model families, we evaluated several alternative base models under the same few-shot inference setup. Table IV reports compilation rates across two prompt settings.

Few-shot prompting improved compilation rates for all three models. Llama 4 Maverick achieved the strongest overall performance, reaching 86% and 83% in the few-shot setting across Tests A and B. These results confirm the wrapper and prompt structure benefits beyond a single model family.

We also evaluated pass@1 for these alternative models using few-shot prompting and 10 generations per prompt. Table V summarizes the results. Llama 4 Maverick again performed best among the alternative models, substantially outperforming Qwen 3 Max and GPT-5 Mini. This suggests that the interaction between model family and prompt format matters: some models are better able to leverage the structured context and few-shot demonstrations used by our wrapper.

D. Discussion

Overall, the results show that fine-tuning on domain-specific $P4_{16}$ data improves the performance, with few-shot prompting providing an additional substantial gain. The verification pipeline is also essential: many generated programs that compile are not necessarily functionally correct, which justifies the use of pass@k evaluation through VRF B.

Our fine-tuned model's performance is consistent with prior work on P4-specific code generation. Dumitru et al. reported compilation rates between 66.7% and 80% for smaller models fine-tuned on P4 code [3]. Our fine-tuned DeepSeek-Coder-V2-Lite-Instruct model achieved competitive compilation and strong pass@k performance on this domain-specific task. While direct comparison with general-purpose code benchmarks such as HumanEval or MultiPL-E should be made cautiously, these results suggest that targeted fine-tuning plus verification can make $P4_{16}$ generation substantially more reliable even in a low-resource programming language setting. Our work aligns with AI-driven networking by coupling generation with validation to ensure reliable intent-based network configuration.

VI. LIMITATIONS AND THREATS TO VALIDITY

Our current validation pipeline consists of compilation and passing P4TestGen generated tests on the code. This leaves a possibility of producing syntactically valid and internally consistent P4 code that does not align with the input intent. To alleviate this, we are currently expanding VRF B through code-intent verification through classifying blocks of generated code and matching them with classified intents.

A second limitation arises from the use of few-shot prompting. Although our initial checks did not identify direct copying of the examples, there remains a risk that the model interpolates or partially replicates few-shot snippets in ways that superficially resemble correct solutions. This could inflate pass@k scores, since the model may rely on formulaic or memorized patterns rather than truly interpreting the intent. A more rigorous assessment would require a larger and more diverse dataset with test cases for each prompt that can detect imitation, enforce intent alignment, and provide statistically meaningful evidence that the model is genuinely performing intent translation rather than reproducing templates.

Finally, in constructing our dataset, used for training and testing our system, we used annotations of publicly available code as intents. While these annotations capture the natural-language description of each code's functionality and maintain a declarative form, they may not reflect the appropriate level

of abstraction required for all target use cases. Collecting human-specified intents and corresponding generated code in a working setup, such as using the presented prototype, is an avenue towards enhancing this in the future.

VII. Conclusion and Future Work

We presented PINC (P4 and Intent for Network Configuration), and, through a working prototype, demonstrated the feasibility of an LLM-based approach for intent-based automation of network dataplane configuration. Our prototype achieved a compilation success rate of up to 98% and a solution generation success rate ranging from 48% to 100%. We release both the prototype and the dataset used for instruction tuning to support community use, refinement, and further exploration. This work establishes a foundational baseline and identifies three key directions for future development: (1) strengthening generation capabilities through online and improved model training, prompt engineering, and constraint handling; (2) integrating runtime performance verification and feedback mechanisms to enable closed-loop validation; and (3) evaluating the system across diverse network topologies and traffic, incorporating comprehensive benchmarking and support for application-aware multi-agent deployments.

References

[1] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-based networking - concepts and definitions," 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9315/

[2] A. Angi, A. Sacco, and G. Marchetto, "Llnet: An intent-driven approach to instructing softwarized network devices using a small language model," *IEEE Transactions on Network and Service Management*, vol. 22, no. 4, pp. 3403–3418, 2025. [Online]. Available: https://doi.org/10.1109/TNSM.2025.3570017

[3] M.-V. Dumitru, V.-A. Bădoiu, A. M. Gherghescu, and C. Raiciu, "Generating p4 dataplanes using llms," in *2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*, 2024, pp. 31–36. [Online]. Available: https://doi.org/10.1109/HPSR62440.2024.10635926

[4] O. G. Lira, O. M. Caicedo, and N. L. S. da Fonseca, "Large language models for zero touch network configuration management," *IEEE Communications Magazine*, vol. 63, no. 7, pp. 146–153, 2025. [Online]. Available: https://doi.org/10.1109/MCOM.001.2400368

[5] F. Ruffy, J. Liu, P. Kotikalapudi, V. Havel, H. Tavante, R. Sherwood, V. Dubina, V. Peschanenko, A. Sivaraman, and N. Foster, "P4testgen: An extensible test oracle for p4," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 136–151. [Online]. Available: https://doi.org/10.1145/3603269.3604834

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

[7] R. Mondal, A. Tang, R. Beckett, T. Millstein, and G. Varghese, "What do llms need to synthesize correct router configurations?" in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, ser. HotNets '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 189–195. [Online]. Available: https://doi.org/10.1145/3626111.3628194

[8] L. Dinh, S. Cherrared, X. Huang, and F. Guillemin, "Towards end-to-end network intent management with large language models," 2025. [Online]. Available: https://arxiv.org/abs/2504.13589

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[10] Anthropic, "Introducing the next generation of claude," Mar 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family

[11] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3086704

[12] C. Györgyi, S. Laki, and S. Schmid, "Toward highly reliable programmable data planes: Verification of p4 code generation," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 1–5. [Online]. Available: https://doi.org/10.1109/NetSoft57336.2023.10175397

[13] D. Wu, X. Wang, Y. Qiao, Z. Wang, J. Jiang, S. Cui, and F. Wang, "Netllm: Adapting large language models for networking," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 661–678. [Online]. Available: https://doi.org/10.1145/3651890.3672268

[14] Z. He, A. Gottipati, L. Qiu, X. Luo, K. Xu, Y. Yang, and F. Y. Yan, "Designing network algorithms via large language models," in *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, ser. HotNets '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 205–212. [Online]. Available: https://doi.org/10.1145/3696348.3696868

[15] M. Riftadi and F. Kuipers, "P4i/o: Intent-based networking with p4," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 438–443.

[16] V. Komanduri, S. Estropia, S. Alessio, G. Yerdelen, T. Ferreira, G. P. Roldan, Z. Dong, and R. Rojas-Cessa, "Optimizing llm prompts for automation of network management: A user's perspective," in *2025 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2025, pp. 0958–0963.

[17] V. Komanduri, S. Alessio, S. Estropia, G. Yerdelen, T. Ferreira, M. Gunti, Z. Dong, and R. Rojas-Cessa, "Partitioning prompts for higher efficacy in network design with large language model," in *2025 IEEE 26th International Conference on High Performance Switching and Routing (HPSR)*, 2025, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPSR64165.2025.11038889

[18] T. B. et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[19] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen, L. Marris, S. Petulla, C. Gaffney, and A. Aharoni, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," 2025. [Online]. Available: https://arxiv.org/abs/2507.06261

[20] X. Nie, L. Dong, H. Zhang, J. Xiao, and G. Sun, "Elutq: Efficient lut-aware quantization for deploying large language models on edge devices," *arXiv preprint arXiv:2510.19482*, 2025.

[21] M. Zhang, J. Cao, X. Shen, and Z. Cui, "Edgeshard: Efficient llm inference via collaborative edge computing," *arXiv preprint arXiv:2407.21325*, 2024.

[22] Z. Lu, X. Li, D. Cai, R. Yi, F. Liu, W. Liu, J. Luan, X. Zhang, N. Lane, and M. Xu, "Demystifying small language models for edge deployment," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

[23] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[24] DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, W. Zeng, X. Bi, Z. Gu, H. Xu, D. Dai, K. Dong, L. Zhang, Y. Piao, Z. Gou, Z. Xie, Z. Hao, B. Wang, J. Song, D. Chen, X. Xie, K. Guan, Y. You, A. Liu, Q. Du, W. Gao, X. Lu, Q. Chen, Y. Wang, C. Deng, J. Li, C. Zhao, C. Ruan, F. Luo, and W. Liang, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2406.11931

[25] M. C. et al, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374