Extending Test-driven development to Softwarized Networks and Intent Based Networking

1st Davide Berardi
Dept. of Engineering and Sciences
Universitas Mercatorum, Rome, Italy
davide.berardi@unimercatorum.it

2nd Mattia Fontana
Dept. of Engineering and Sciences
Universitas Mercatorum, Rome, Italy
mattia.fontana@studenti.unimercatorum.it

3rd Barbara Martini
Dept. of Engineering and Sciences
Universitas Mercatorum, Rome, Italy
barbara.martini@unimercatorum.it

***Abstract*—The field of Intent Based Networking (IBN) has recently focused on the use of systems powered by Generative Artificial Intelligence and Large Language Models (LLMs) to generate and configure the network. While being a powerful tool to assist Network Administrators, these methods are far from perfect. They tend to hallucinate and generate bad or sub-optimal configurations. To avoid these problems, we propose the integration of Test-Driven development to the world of Softwarized Networks, as a feedback for automated assistants such as LLM-based configuration generator, leveraging technologies such as Atomic Predicates (AP) and Retrieval Augmented Generation (RAG). In this paper, we describe the usage of an assistant over a softwarized network, making it work in conjunction with Network Policy Enforcement, generating an effective Test-Driven Development for Softwarized Network. Our findings highlight the potential of these combined approaches, mutually benefit each other to reach the goal of a powerful co-pilot for Complex Network Configuration.**

***Index Terms*—network softwarization, software engineering, test driven development**

I. Introduction

The topic of Test-Driven-Development is one of the most important in modern Software Engineering [1]. By providing how the final product must behave trough automatic tests the developers can easily implement software without the difficult task to analyze if a previously resolved bug has been reintroduced (regression). While this is a solid topic which set the basis for more advanced techniques in the field of DevOps such as Continuous Integration (CI), Continuous Development (CD), and Continuous Testing (CT), it is not well adopted and well adaptable to traditional Networks [2]. Test-Driven-Development also plays a role in the modern "vibe-coding" practices. The approach of "vibe-coding" is an informal way to refer to the introduction of a Large Language Model in the loop of development. That is, a model is in charge of generating code or configuration, while the user is in charge of evaluating the quality of the code and correct the LLM. This latter approach, applied to networking is called "co-pilot". A co-pilot is an AI-based-system in charge of generating configuration by starting from an high level intent [3]. On a broader term, this approach falls under the category of Intent Based Networking (IBN), which focus on the definition of network specification from an high level point of view, abstracing the lower layer. In this "vibe-coding" approach, Test-Driven-Development can be in charge of automatically correcting the model, refusing programs with errors or badly written configuration without the human interaction. To achieve these goals, we leveraged different technologies such as Atomic Predicates and Retrieval Augmented Generation. The former is used to ensure that the network is compliant with the specifications enforced by the administrator. The latter is used to increase the performance of LLMs. The combination of these two approaches could lead to a significant improvement in the configuration generation scenario. The goal of this paper is to integrate Test-Driven-Development with a co-pilot specialized in network configurations. Which could lead to an extremely fast prototyping of network configurations, starting from natural language-expressed requirements. Developing this approach we identified the following research questions we want to tackle: (Q1) How much a co-pilot specialized in network configurations can benefit from the usage of Test-Driven-Development? (Q2) Which are the architectural building blocks to create a System which can put a co-pilot in the loop of network configuration, enabling the LLM to DevOps techniques such as CI/CD/CT? (Q3) Which are the main tests that can be done easily on a network architecture which can automatically give feedbacks to the AI-engine? In summary, replying to the questions above, we present the following contributions: (A) An analysis on how Test-Driven-Development can be used in conjunction with AI-based Intent Based Networking systems. (B) A reference implementation of a framework which can be used to enable the interactions between the AI-model and the Softwarized network. (C) A reference system which can be used to integrate specialized AI agents and Test-Driven-Development with an existing Software-Based network without any modification to the network itself.

II. Background and State of the Art

This work is rooted on two different topics: Intent Based Networking and Test Driven Development. The former topic is one of the most analyzed in relation to 5G, beyond 5G and 6G networks [4]–[6]. This kind of approach roots on the fact that networks are growing in complexity and capabilities. Unfortunately this could lead to technical difficulties in configuration by Network Architects. To overcome these limitations, efforts were made on the easiness of configuration. For instance, high-level interfaces were proposed to implement flexible requirements easily. With the explosion in popularity of Large Language Models, Generative Artificial Intelligence, and Natural Language Processing, this approach evolved. That is, the previously cited high-level interfaces can be translated to low-level primitives by leveraging the translation capabilities offered by LLMs. In this case, the API is completely transparent to the user, by translating the requirements expressed in natural language to low-level specifications and configurations. Recent works such as [7], [8], are based on this approach. A, so called, Co-pilot architecture is presented, in which a LLM ChatBot is in charge of configuring the system. The LLM is specialized using a technique called Retrieval Augmentation Generation (RAG). This techniques extends the capability of the LLM using a specific knowledge base, for instance the user or developer manuals / datasheets of the various components. Unfortunately this approach is not sufficient to overcome all the limitations. One of the main limitations is the analysis of corner cases. That is, if one case is not well explained in the RAG knowledge base, the results will be sub-optimal. This is also due to the fact that the knowledge base is usually imported in the model using Natural Language and unstructured documents [9]. While being outside of the scope of the work presented in this paper, is one of the most present limitation in current AI systems. The other main limitation of these approach is the requirement of an human-in-the-loop [10] when dealing with configuration validation. While being on-topic for the Industry 5.0 manifesto, it is an error prone task and difficult to do without the right tools. That is, the configurations are not easy to analyze and test without a testbed or a validation from an extremely specialized human. Allucinations are present in modern AI models and therefore the generated code could lead us to disasters and problems. Consider a configuration script for a 5G modem which integrates in it a clean-up subroutine. If this routine is generated from a hallucination it could remove files not related with the experiment. The approach we tackle in this work is to implement tests on the network to automatically evaluate undesiderable situations. Integrating policies in the loop of evaluation can give to the generator part of the system more degrees of freedom in the creation of the configuration, without the needing of generating an extremely safe and specific configuration every time. To implement this kind of approach, we need to have a system which can give to the network administrators, architects, and developers the possibility to recognize bad configurations. The definition of a "wrong" or "bad" configuration is not easy to give. From the system point of view, a "bad" configuration is a configuration which is not instantiable due to physical or hardware related constraints. From the point of view of the network administrator it could be a configuration which disables the communication between two nodes which must communicate every time. In literature there are some different approaches to overcome these problems. One of the most popular one is the policy based networking. For instance, Berardi et al. [2] employ Atomic Predicates to evaluate "operative" network concepts such as Reachability, To-waypoint, or Un-reachability of nodes. That is, the Network Administrator then can create a map of

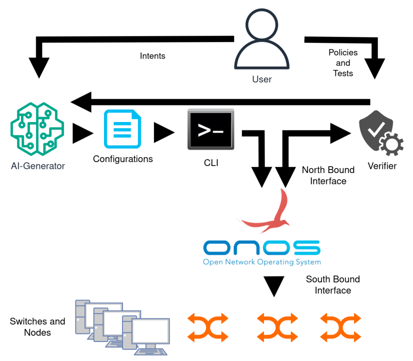

Fig. 1. Architectural design and interactions between the components. The Network Operating System, in this case, is pictured as ONOS, but can be easily exchanged with other NOS or controllers such as Ryu or Floodlight.

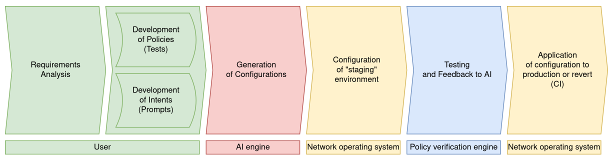
Requirements Analysis
Development of Policies (Tests)
Development of Intents (Prompts)
Generation of Configurations
Configuration of "staging" environment
Testing and Feedback to AI
Application of configuration to production or revert (CI)
User
AI engine
Network operating system
Policy verification engine
Network operating system

Fig. 2. Flow of execution of a network CI pipeline. In the former part of the image, the sequential steps are presented. On the latter part of the image, the subject in charge of that step is presented such as User, AI engine, the Network Operating System, and the Policy Verification Engine. Note that only the first and last steps requires human interaction, while the other are completely automated in a "zero-touch" flavour [14].

systems that are required to communicate using a specific flow of informations. If the system do not behave in such a way, the atomic predicates will fail to validate, leading to a "bad" configuration. Other projects such as [11] employs Headers Space Analysis, which can be used to achive the same goal. While theoretically focused on the same problem, the performances of the two systems are completely different. As stated in papers such as [12], the Atomic Predicates clearly win in terms of speed of verification. One of the related works on the field is the work by Esposito et al. [13], in which a different approach is analyzed by leveraging Behavior-Driven Development instead of Test-Driven Development.

III. System Architecture

The System we present to automatize Intent Based Networking is composed of four main building blocks. These, among the other vital parts for the proper operation of the network such as Switches, are presented in Figure 1 and are: (A) The Network Operating System, the manager of the entire network. Leveraging Softwarization technologies such as Software-Defined-Networking is easy for the Network Operating System to reconfigure the entire network on the fly, managing switches trough programs injected by the North-Bound interface. (B) The AI-Generator, an AI-based system which is in charge of generating configurations and inject them in the Network Operating System. (C) The Verifier, which load policies and can receive reconfiguration events from the Network Operating System. Upon a reception of such an event, the Verifier will query the Network Operating System for the network topology and switch configurations. After this step, the internal Solver engine will verify if the policies are satisfied by the configuration. (D) The User, which work is reduced to only express the requirements in Natural Language and generate policies in a format which can be processed by the Verifier; in the last step the user is required to evaluate the results and give a feedback to the system, accepting or rejecting the configuration. This sort of "human-in-the-loop" approach is then confined to the only mandatory parts. That is: the goal of this design is to avoid most of the manual operations performed by the user, posing the automation of the Intent Based configuration at its extreme. **Policies Design.** The policies are based on the technique of Atomic Predicates [2]. Abstracting the underlying details, we can use them to efficiently express the $Reachability(A,B)$ function, which express the possibility to send packets from the node A to the node B. To attach these node to a layer, we need to translate the rules of the layer into the atomic predicates, for instance VLANs for layer 2, routing for layer 3, firewalling for layer 4 and so on. Using this reachability building block we can express three more functions: Unreachability can be simply expressed as the negation of reachability between two nodes: $\neg Reachability(A,B)$. Full reachability express the ability of two nodes to talk to eachother in both ways: $FullReachability(A,B)=Reachability(A,B)\wedge Reachability(B,A)$. To-Waypoint express the constraint that a traffic originating from node A for the node B, must pass from M every time. Given $A\neq B, B\neq M, A\neq M$ and $Reachability(A,M)\wedge Reachability(M,B)$ then we can express To-Waypoint as: $\forall_{m\in Nodes-(M)}\neg Reachability(A,m)\vee\neg Reachability(m,B)$. These policies are written in a language that express the logical relationships between the nodes. This approach can be assisted by leveraging the Retrieval Augmented Generation capability of LLMs by instructing the LLM with the operative manual and examples of the language. With this approach, the LLM can be used also to generate the tests. **Pipeline and Interaction Flow.** Figure 2 depicts the operational workflow and the usage of this approach. Initially the User must analyze the requirements of the network in terms of network topology, bandwidth requirements, security policies, etc. From this he can process them to generate the prompts and tests. This part is dependant on the human expertise and it is where the human design is really needed. This part can benefit from the employments of LLM systems to generate both prompts and policies. In [15] is specified how to implement an Automated Prompt Engineering (APE) methodology, which can be used in this case to automatically converge to a satisfactory prompt which can be then sent to the next step. After this step, the AI-engine takes the control and start to generate configurations, these are automatically sent to the NOS trough a technique called "function-calling". This technique gives to the LLM the possibility to execute commands or interact directly with Application Programming Interfaces such as the ones exposed by the NOS. Obviously, it would be an enormous risk in most of the cases to directly apply the configurations generated by the AI model directly in production, even if the system is not a critical one. To avoid this problematics, the NOS applies the configuration in a "staging" environment. In this case we propose a system which can behave similarly to what happens in Blue-Green Deployments [16], a popular way to avoid the downtime of an infrastructure under update. If the benefits of Blue-Green Deployments are not required, this step could be implemented also with a virtual staging or a more constrainted environment, to avoid costs related to the duplication of the systems. After the configuration of the system, the tests runs and obtain a set of boolean results. These results are sent to the user and the AI system, to give feedback on the configuration, the user is then prompted if he wants to refine the configuration based on the results. After this phase, the NOS can then switch the "staging" deployment with the "production" deployment or leave the situation as is. For instance, let us suppose to have a big event in a place in which a number of end user terminals will be present. We can think about a soccer match or a concert in a stadium. Also, we plan to have two different services, one for the attendant and one for the ticket verifications. Abstracting from the physical layer problematics, we focus on the analysis of the requirements, generating the following intent: *Generate a configuration for the network which can be used to interact directly with internet by the users. We plan to have* 50.000 *people which will use the internet connection to post videos online. Also, the internal ticketing system must be reachable every time by the ticket verification assistants.* While not extremely specific, this intent can be a base building block to analyze the accuracy of the generation. As clearly evinctable from the example, the AI-system need to know precise information on the network design such as how to reach the internal ticketing system or how to identify the ticket verification assistant terminals. This is done trough a technique called Retrival Augmented Generation, as described in Section II and in previous works such as [8] In our case the generation could be augmented by the definition of the roles of the network nodes, by providing some internal documentation on the inventory. The policies can be expressed easily with the following rules, specified as described in Section III: First of all, we specified that all the user nodes must reach the internet trough a $Gateway$. $\forall_{N\ \in\ EndUsersNodes}FullReachability(N,Gateway)$ Also, Ticket checker nodes must always be able to reach the Ticket-validation system. Simplifying the scenario, we can assume to have a single server $TicketServer$. $\forall_{N\in TicketNodes}FullReachability(N,TicketServer)$. Finally, we can specify some security policies such as the fact that no End User can interact with the Ticket-validation system.$\forall_{N\in EndUsersNodes}\neg Reachability(N,TicketServer)$. To hold this property we must also declare that no node is present both as an End Users' Node and a Ticket-validation node: $EndUsersNode\ \cap\ TicketNodes=\varnothing$. After providing these policies to the Verifier component, we leave the computation to the system, which generates a configuration and install it in the staging environment. The feedback gets sent to the AI model and evaluated, leading to a configuration which can be applied or not, depending on the user feedback.

IV. Implementation and Testbed

The implementation of a reference design as described in III, took different software in consideration. First of all, we selected ONOS as the controller and Operating System of the network. We did not develop the solution vertically basing on this software completely but we created a modular design, with plugins and components that can be attached to ONOS API. In this way the controller choice is not mandatory and the effort to enable the use of a different one just relies on the creation of a specific plugin to implement the API dialogue. The second component is the integration of a LLM agent as a ONOS application to implement the AI engine. This system is built as a Python program. To develop a robust autonomous AI agent with tool-calling capabilities we employed SmolAgents. This framework facilitates seamless integration between the language model's reasoning capabilities and external tool interfaces, providing a standardized protocol for tool discovery, selection, and execution. The agent's cognitive architecture implements a deliberative reasoning paradigm, where each decision cycle involves perception of the current state, goal formulation, plan generation, and action execution. This approach enables the agent to maintain coherence across multi-step tasks while adapting to dynamic environmental conditions and unexpected outcomes. The agent system incorporates a diverse ecosystem of specialized tools: intent management utilities for processing user requests andorchestrating appropriate responses, user interaction management tools that handle bidirectional communication flows, RESTful API integration tools that execute HTTP calls to controller endpoints for backend system integration, user confirmation tools that request explicit authorization before executing; network topology discovery tools that query the ONOS controller to retrieve current network topology information, network information management tools for processing, analyzing network state data and available endpoints; intelligent strategy recommendation tools that analyze user intent and suggest optimal ONOS strategies and endpoints for task execution, API payload generation tools that construct structured data payloads for RESTful API calls. The reasoning process is governed by a system prompt that establishes the agent's operational parameters, behavioral guidelines, and decision-making criteria. The experimental evaluation encompasses three OpenAI GPT models representing different performance-cost trade-offs within the GPT-4 family. The evaluation suite includes GPT-4o ($2.50 per 1M tokens), GPT-4.1-mini ($0.40/1M), and GPT-4.1-nano ($0.10/1M). Model selection criteria considered the balance between reasoning capabilities, tool-calling proficiency, and economic efficiency to provide comprehensive coverage of deployment scenarios ranging from high-performance applications to cost-sensitive implementations. The Policy enforcement part was implemented using TechNETium [2]. A software which leverages Atomic predicates and Reachability-based language to evaluate the network reachability in an extremely fast fashion. An example of policy specification is the following code excerpt:

```
1 bdd-reachability of:22b of:21e
2 bdd-unreachability of:21a of:19c
3 bdd-towaypoint of:20a of:21b of:22c
```

In this case, in line 1, we describe how the port of a virtual OpenFlow switch `of:22b` must reach the port `of:21e`. Similarly, in line 2, we express that port `of:21a` and port `of:19c` cannot communicate. Finally, at line 3, we express that all the traffic originating from port `of:20a` for port `of:21b` must pass trough port `of:22c`.

Experimental Evaluation. To validate the agent's capabilities and ensure robust performance across diverse network management scenarios, a comprehensive testbed was developed encompassing three primary use cases that represent fundamental network control operations. The testbed evaluation framework consists of systematically designed test cases that assess the agent's ability to interpret natural language instructions and translate them into appropriate ONOS controller actions. The first test case (T1) focuses on traffic blocking functionality, where the agent must process natural language requests to "block HTTP traffic between specific devices" and correctly identify the appropriate ONOS flows API endpoint for implementing fine-grained traffic filtering rules between designated network devices. The second test case (T2) evaluates traffic allowance capabilities, requiring the agent to interpret permissive traffic policies such as "allow all traffic between devices" and map these requests to the ONOS intents API for establishing bidirectional connectivity between specified network endpoints. The third test case (T3) assesses advanced routing functionality, challenging the agent to process complex path specification requests like "route traffic through specific path" and generate appropriate Intent Based configurations that enforce traffic traversal through designated intermediate nodes. The evaluation framework employs a multi-dimensional metric system designed to capture both functional correctness and operational efficiency. (A) Success Rate is a composite metric evaluating end-to-end task completion effectiveness, comprising five critical components such as: API call execution capability, successful API response reception, correct endpoint selection, inclusion of task-specific implementation details, and results of the tests evaluated by the Policy Validation system. (B) Consistency Score: A reliability metric assessing behavioral reproducibility across multiple test executions, evaluated through three dimensions: (1) Success rate consistency measuring the stability of task completion performance, (2) API call consistency evaluating the uniformity of interaction patterns with external services, and (3) Result structure consistency assessing the standardization of output formats and data structures. Finally we evaluated (C) Cost Efficiency: A comprehensive performance-per-dollar metric calculated as the sum of success rate, consistency score, and API success rate divided by the model's token cost, enabling economic optimization of model selection for production deployments. The experimental evaluation, presented in Table I reveals significant performance variations across the three GPT models, with distinct trade-offs between capability and cost-effectiveness. Overall success rates demonstrated moderate

Model	gpt-4o	gpt-4.1-mini	gpt-4.1-nano
Consistency T1	92.59%	92.59%	100%
Consistency T2	92.59%	100%	100%
Consistency T3	100%	100%	100%
Success Rate	44.44%	44.44%	33.33%
Consistency	95.06%	97.53%	100.00%
API Success	66.67%	66.67%	55.56%
Test Success	49.63%	59.53%	56.32%
Average Execution Time	112s	66s	45s
Cost Efficiency ($)	82.47	521.60	1,888.89

TABLE I
EXPERIMENTAL EVALUATIONS, AVERAGE OF TEST RESULTS REPEATED 10 TIMES PER MODEL

performance across all models, with GPT-4o and GPT-4.1-mini achieving identical success rates, while GPT-4.1-nano achieved lower success rate, indicating a performance degradation in the most cost-optimized variant. Consistency analysis revealed exceptionally high behavioral reproducibility, with GPT-4.1-nano achieving perfect consistency across all test cases, GPT-4.1-mini demonstrating 97.53% consistency with perfect scores on T2 and T3, and GPT-4o showing 95.06% consistency with the lowest individual test performance on T1. API success rates followed similar patterns to overall success rates, with GPT-4o and GPT-4.1-mini achieving 66.67% API success rates compared to GPT-4.1-nano's 55.56%. Execution time analysis revealed an inverse relationship between model cost and response latency. GPT-4.1-nano demonstrated the fastest average execution time, followed by GPT-4.1-mini, while GPT-4o required higher average execution time, suggesting computational overhead in higher-tier models. The "Test Success" metric evaluates the complete task execution performance through the four core aspects: the capacity to execute API calls, the retrieval of valid server responses, the accuracy during the selection of API endpoints through mapping between human intents and corresponding network controller interfaces, the integration of task-specific implementation details, such as port 80 specification for HTTP traffic blocking or routing through dedicated intermediate devices. The Test system reveal itself to be particularly useful for these case, giving feedback to the user nearly in the 50% of the cases, with slightly variations between the three models. If these configurations were installed in the real system they would produce an-invalid configuration, which would lead to system malfunctioning or traffic blocks. The cost efficiency analysis produced remarkable results, with GPT-4.1-nano achieving the highest efficiency score of 1,888.89 performance points per dollar, followed by GPT-4.1-mini, and GPT-4o. This 23-fold efficiency advantage of GPT-4.1-nano over GPT-4o demonstrates the significant economic benefits of model optimization for production SDN management deployments, despite the moderate reduction in absolute performance metrics. Individual test case performance patterns showed consistent behavior across blocking (T1), allowing (T2), and routing (T3) operations, with routing tasks demonstrating the highest consistency scores across all models, suggesting that path specification tasks may be more amenable to reliable agent-based automation than traffic filtering operations.

V. CONCLUSIONS

In this work we presented an integration between an Agent-based LLM system and a Test-driven-development approach. This paper focus on the integration of the two parts and the development of an AI agent which can dialogate with a SDN controller such as ONOS. Referring to the research questions posed in Section I: (Q1) obtains an answer in Section IV, in which we analyzed the configurations, trough composed performance metrics, also including test-driven feedbacks; to reply to (Q2) we dedicated the entire Section III, proposing a reference design to implement Test-driven-development in Intent Based Networks; finally, (Q3) get answered in Section IV. The results, which takes in consideration different models, confirm the fact that a Test-Driven-Development can introduce benefits in terms of performance and precision for the field of Intent Based Networking based on LLMs and co-pilots.

Acknowledgements. This research was financially supported by Universitas Mercatorum under the 2024 FIN/RIC program (projects 23-FIN/RIC and 26-FIN/RIC).

REFERENCES

[1] S. Jalil *et al.*, "Chatgpt and software testing education: Promises & perils," in *2023 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pp. 4130–4137, IEEE, 2023.

[2] D. Berardi *et al.*, "Technetium: Atomic predicates and model driven development to verify security network policies," in *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–6, IEEE, 2020.

[3] C. Hegedűs *et al.*, "Co-pilots for arrowhead-based cyber-physical system of systems engineering," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pp. 1–6, IEEE, 2024.

[4] E. Zeydan *et al.*, "Recent advances in intent-based networking: A survey," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pp. 1–5, IEEE, 2020.

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

[6] Y. Wei *et al.*, "Intent-based networks for 6g: Insights and challenges," *Digital Communications and Networks*, vol. 6, no. 3, pp. 270–280, 2020.

[7] C. Hegedűs *et al.*, "Co-pilots for arrowhead-based cyber-physical system of systems engineering," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pp. 1–6, 2024.

[8] M. Fontana *et al.*, "Leveraging llm-powered intelligent chatbots for intent-based networking in 5g modem reconfiguration," in *11th IEEE International Conference on Network Softwarization (NETSOFT)*, pp. 1–5, IEEE, 2025.

[9] M. Pondel *et al.*, "Ai tools for knowledge management–knowledge base creation via llm and rag for ai assistant," in *European Conference on Artificial Intelligence*, pp. 3–15, Springer, 2024.

[10] I. Drori *et al.*, "Human-in-the-loop ai reviewing: feasibility, opportunities, and risks," *Journal of the Association for Information Systems*, vol. 25, no. 1, pp. 98–109, 2024.

[11] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 99–111, 2013.

[12] H. Yang *et al.*, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2015.

[13] F. Esposito *et al.*, "A behavior-driven approach to intent specification for software-defined infrastructure management," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 1–6, 2018.

[14] B. Martini *et al.*, "Intent-based zero-touch service chaining layer for software-defined edge cloud networks," *Computer Networks*, vol. 212, p. 109034, 2022.

[15] Y. Zhou *et al.*, "Large language models are human-level prompt engineers," in *The Eleventh International Conference on Learning Representations*, 2022.

[16] B. Yang *et al.*, "Survey and evaluation of blue-green deployment techniques in cloud native environments," in *Service-Oriented Computing–ICSOC 2019 Workshops*, pp. 69–81, Springer, 2020.