A Multi-Agent Framework for Natural Language-Driven Network Simulation

1st Alberto Salvatore Colletto
AlphaWaves S.r.l.
Torino, Italy
a.colletto@awaves.it

2nd Paolo Bonelli Bassano
AlphaWaves S.r.l.
Torino, Italy
p.bassano@awaves.it

3rd Alessio Viticchié
AlphaWaves S.r.l.
Torino, Italy
a.viticchie@awaves.it

4th Alessandro Aliberti
Politecnico di Torino
Torino, Italy
alessandro.aliberti@polito.it

***Abstract*—This paper presents a framework that combines large language models (LLMs) with a multi-agent system (MAS) to automate the translation of natural language instructions into executable network simulations. Designed to improve the usability of traditional simulators, the proposed system enables users, regardless of technical background, to interact with Mininet through intuitive language prompts. The MAS architecture assigns specific tasks to agents, including input parsing, topology analysis, code generation, execution, and result interpretation. A retrieval-augmented generation (RAG) module boosts contextual understanding by accessing authoritative documentation. The framework was tested on benchmark scenarios of increasing complexity, such as connectivity checks, routing optimization, and DDoS mitigation. Results show that LLMs are effective at interpreting intent and refining prompts, while code generation, especially for complex tasks, remains a challenge. Larger models performed better in accuracy and robustness, whereas smaller ones struggled with error handling and refinement. These findings highlight both the promise and current limitations of agentic AI in network simulation, positioning the system as a foundation for more intelligent and accessible tools in education, research, and infrastructure management.**

***Index Terms*—Agentic AI, Large Language Models, Network Simulation.**

I. Introduction

Network simulation is essential for designing, analyzing, and securing modern IT infrastructures. It enables safe exploration of "what-if" scenarios without impacting live systems. Tools like NS-3, OMNeT++, and Mininet support this through discrete-event simulations [1]. However, these tools often lack usability and accessibility, especially for novices [2]. Creating network topologies and configurations typically requires programming expertise, making the process time-consuming, error-prone, and unsuitable for non-expert users. This limits experimentation for educators, small enterprises, and researchers.

To address these issues, this paper introduces an AI-driven framework that integrates large language models (LLMs) into a multi-agent system (MAS). The system converts natural language prompts into executable simulations, with agents managing input parsing, topology generation, code execution, and result analysis. This modular and scalable design simplifies workflows and eliminates manual scripting. Recent work confirms the viability of LLMs for modeling both configuration and behavioral logic in simulations [3], supporting adaptive and dynamic environments.

The goal is to democratize access to network simulation by reducing technical barriers. Leveraging LLMs and MAS architecture, the framework enables IT administrators, educators, and small enterprises to conduct network analysis and security testing. Improving accessibility in this field can accelerate innovation, enhance cybersecurity, and support integration of emerging technologies.

The remainder of the paper is organized as follows: Section II reviews related work on network simulation tools, LLMs, and MAS. Section III describes the system architecture and methodology. Section IV presents use cases demonstrating the system's capabilities, and Section V reports experimental results and performance evaluation. Section VI discusses the findings, their implications, and limitations. Finally, Section VII outlines future research directions and concludes the paper.

II. Background

In recent years, advances in Artificial Intelligence (AI) have transformed a wide array of technological domains. Among the most impactful developments are LLMs and agentic AI frameworks [4]. LLMs, built on transformer architectures and trained on large textual corpora, have achieved impressive results in language understanding, reasoning, and generation [5]–[7]. Successive innovations, from BERT and GPT-3 to more advanced models like GPT-4 and LLaMA 4, have brought LLMs closer to human-level contextual comprehension [8]. These capabilities have enabled LLMs to move beyond general-purpose applications into specialized domains. In networking, they are now used for automated configuration, documentation synthesis, log analysis, and natural language interfaces for network operations [9], marking a step toward zero-touch network management [10].

In parallel, agentic AI frameworks have emerged as modular systems composed of autonomous agents that collaborate on complex tasks such as reasoning, planning, and execution [11]. When paired with LLMs, these frameworks are increasingly applied to infrastructure monitoring, cybersecurity, and network automation [12], making them particularly suited for distributed, fault-tolerant, and coordinated network operations.

Network simulators such as Mininet, NS-3, and GNS3 remain foundational for research and education. They allow for the modeling and testing of topologies, protocols, and

979-8-3503-9281-4/25/$31.00 ©2025 IEEE

routing strategies in controlled environments. Mininet stands out for lightweight SDN-based emulation [13], while NS-3 offers fine-grained protocol control at the cost of usability [14], and GNS3 provides high-fidelity hardware emulation with significant resource demands [15]. Despite their strengths, these tools face persistent limitations in scalability, real-time dynamics, and user accessibility [16].

The convergence of AI and network simulation is opening new possibilities for optimizing protocol performance, detecting anomalies, and enhancing resilience [17]. LLMs offer an intuitive interface between natural language and technical simulation tasks, though challenges remain in areas like generalization, data quality, and real-time adaptability [18]. In this context, MAS offer a promising architectural approach for managing the complexity of large-scale, dynamic networks [19]. AI-driven automation is also being used to improve network configuration and management, from predictive routing to policy enforcement. Yet, explainability, trust, and cascading failure handling remain open research challenges [20].

This paper builds on recent AI-networking intersections by proposing an agent-based framework that combines LLMs and code generation models to automate multi-step network simulations in Mininet. By integrating reasoning, planning, and execution, the system supports complex behavior emulation and offers a flexible tool for education and prototyping. While effective in simplified contexts, scalability to industrial-grade scenarios remains a limitation. Nonetheless, this work contributes to ongoing efforts to embed natural language interaction and autonomous systems into the future of network simulation and analysis.

III. Methodology

This study proposes a MAS-based approach to automatically translate natural language instructions into executable network simulations. The system consists of specialized agents that collaboratively i) process user inputs, ii) analyze network topologies, iii) generate configuration commands, iv) execute simulations, and v) deliver feedback.

The architecture follows a modular workflow where each simulation stage, from interpretation to execution, is managed by purpose-specific agents. Through sequential coordination, natural language directives are transformed into functional simulation tasks. The process unfolds as follows:

1) The user submits a natural language prompt describing the desired network behavior.
2) A context management agent extracts relevant information and maintains coherence across interactions.
3) A prompt refinement agent enhances the input for clarity and completeness.
4) A coordination layer analyzes the refined prompt and current topology to define an execution pipeline.
5) The pipeline, enriched prompt, and context are forwarded to a multi-agent execution system.
6) The system generates and validates a Python script via the Mininet API, executes the simulation, and evaluates results. Agents iteratively assess outputs, detect issues, and refine code as needed until the task is successfully completed.
7) A conversation agent interprets the results and returns a concise summary to the user.

If the simulation doesn't meet expectations, agents autonomously re-initiate the process, regenerating scripts and rerunning simulations. Throughout, they exchange metadata, traces, and diagnostics to guide refinement. This adaptive loop ensures dynamic error recovery, better alignment with user intent, and higher result quality, particularly under ambiguous or evolving prompts. The following subsections detail each stage of this process.

A. First Stage: Enhancing User Prompt

This section describes the multi-agent workflow enabling the system to interpret and respond to user requests in a network simulation environment. The process is divided into three key stages, each managed by a specialized group of agents: i) *User Interaction and Information Extraction*, ii) *Request Enhancement and Correction*, and iii) *Analysis and Command Generation*.

Each stage handles specific tasks, from interpreting input to generating executable Mininet commands. A shared memory module ensures all agents maintain a consistent view of the user's intent and the current network state. Figure 1 summarizes this process, illustrating how the user's initial natural language prompt is progressively transformed into a structured sequence of validated hints, used afterward on MAS Second stage.

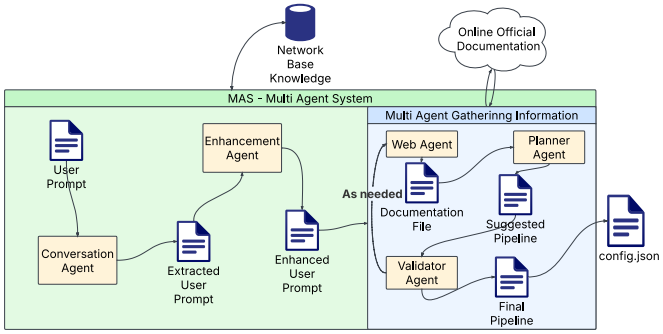

Fig. 1. Multi-agent prompt processing and pipeline generation.

1) User Interaction and Information Extraction: The process starts when the user submits a natural language prompt requesting a network task, such as checking connectivity, simulating faults, or testing specific conditions. For example: "Check connectivity between all hosts using ping. Report which connections work and which do not." The conversation agent extracts key information, such as involved network elements and the requested operation, and stores it in a shared memory module. This memory acts as a global reference, ensuring consistency across agents. All agents access this shared context to maintain a coherent understanding of the user's intent and the network state.

2) Request Enhancement and Correction: Once the user input is captured, it is passed to the *enhancement agent*, which improves the prompt by correcting typos, clarifying ambiguities, and adding missing context. This step ensures the system accurately understands the user's intent. Using NLP techniques, the agent identifies issues and enriches the input. For example, it can correct misspellings like "pign" to "ping" or add details if the prompt omits which hosts are involved in a connectivity check.

3) Analysis and Command Generation: After enhancement, the refined prompt is sent to a team of *analysis agents* that interpret it in the context of the current network topology. These agents determine the specific actions needed to fulfill the request by identifying and generating appropriate Mininet commands aligned with the user's goals and network constraints. For example, to check connectivity, the agents generate ping commands between all hosts. They ensure commands are valid, syntactically correct, and compatible with the network structure by consulting the shared memory and external sources like documentation or forums. The resulting commands are compiled into an executable pipeline, which is validated for correctness. If issues are found, such as conflicts or incorrect sequencing, the agents revise the pipeline. The validated output is stored as "hints" to guide execution and support future interactions.

B. Second Stage: From Code Generation to User Feedback

Building on the validated pipeline, the system translates it into executable code and delivers feedback based on simulation results. This phase involves three key steps, each managed by a dedicated agent: i) *Code Generation and Validation*, ii) *Code Execution*, and iii) *Result Analysis and Feedback*.

The system converts high-level plans into a Python script using the Mininet API, validates and runs the code, then analyzes the output to provide a clear summary to the user. These steps are tightly integrated for robustness and accuracy. Each agent builds on the previous output, and the workflow is designed to be fault-tolerant and adaptive. If errors occur, the system diagnoses and revisits earlier steps to ensure reliability. Figure 2 illustrates the complete workflow, from enhanced input to actionable feedback. The following subsections detail each stage.

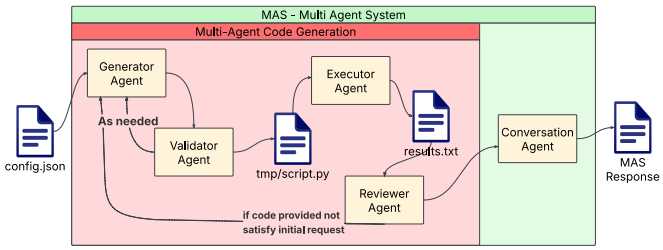

Fig. 2. Multi-agent code generation, execution, and feedback pipeline.

1) Code Generation and Validation: The validated pipeline is passed to the *code generation agent*, which creates an executable Python script using the Mininet API. The script sets up the network topology, runs the required commands (e.g., ping tests), and captures results, aligning with the user's intent and Mininet's syntax. The script is then reviewed by the code validation agent, which checks for syntax errors, logical flaws, and compatibility with Mininet. If issues are found, the agent corrects them, adjusting configurations or fixing code logic, to ensure smooth execution. Once validated, the script proceeds to the execution stage.

2) Code Execution: The *execution agent* runs the validated Python script in the Mininet environment, simulating the network behavior as requested. It captures all outputs, logs, results, and any errors, which are then used in the next stage to assess whether the task was successfully completed.

3) Result Analysis and Feedback Generation: After execution, the result *analysis agent* interprets the simulation output to assess whether the requested behavior was achieved. For example, in a connectivity check, it evaluates ping results to identify successful and failed connections. If the results meet expectations, the task is marked as complete. If not, the agent may recommend restarting the process, adjusting the configuration, or re-engaging code generation to resolve issues. Finally, the *conversation agent* summarizes the outcome for the user, explaining which connections worked, highlighting errors, and suggesting next steps when needed.

C. Iterative Process and System Adaptability

The agents operate in a modular, iterative fashion, enabling re-entrant workflows. If issues arise, especially during result analysis, the system can return to earlier stages like prompt enhancement or code correction. This feedback loop allows dynamic refinement, improving success rates for ambiguous or incomplete requests. A shared memory module maintains context across all stages, storing network topology, user goals, and execution history. This ensures coherence and goal alignment throughout the process.To improve code generation accuracy, agents access a RAG module built on official Mininet documentation and related sources. This helps reduce errors from hallucinated API calls and supports correct handling of complex or uncommon commands.

While the system performs well in many scenarios, it faces challenges with complex topologies or multi-step tasks, where agents may struggle to fully interpret or implement user goals. These limitations point to future improvements in coordination, memory use, and domain-specific support. Overall, the framework shows strong potential for automating network simulation via natural language, with iterative refinement and modular design offering a solid base for future scalability and reliability.

IV. Evaluation Scenarios

To assess the performance, robustness, and practical applicability of the proposed agent-based framework, we developed a series of benchmark scenarios with progressively increasing complexity, from basic connectivity checks to advanced security incident response. These tests aim to evaluate the

system's ability to interpret natural language instructions, generate correct Mininet commands, and iteratively refine its actions based on simulation feedback.

All scenarios were executed using a shared extended star topology, offering enough complexity to support diverse networking tasks while remaining manageable for controlled analysis. The topology includes five hosts and two OpenFlow-enabled switches. A visual representation is shown in Figure 3.

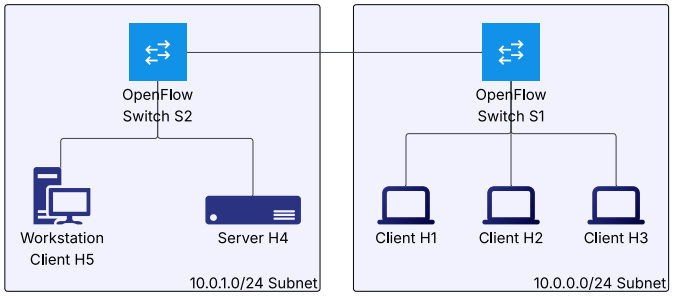

Fig. 3. Benchmark network topology used for simulation scenarios.

Components:

- *Switches*:
  - *s1*: Central OpenFlow switch
  - *s2*: Secondary OpenFlow switch
- *Hosts*:
  - *h1 (10.0.0.1)*, *h2 (10.0.0.2)*, *h3 (10.0.0.3)*: Standard clients
  - *h4 (10.0.1.4)*: Server, designated target for DDoS simulations
  - *h5 (10.0.1.5)*: Remote host accessible only via *s2*

This topology supports full-mesh ping tests, subnet-based routing policies, and realistic simulations of traffic bottlenecks and attack surfaces. Table I presents the benchmark scenarios designed to evaluate specific system capabilities. Each case includes a natural language prompt, task objective, difficulty level, and expected behavior, allowing for a structured assessment of the framework across varying levels of complexity.
To systematically evaluate the system's capabilities and limitations, each scenario was executed under controlled conditions, focusing on how effectively the framework could interpret user intent, generate executable code, and adapt to feedback. The detailed analysis of these behaviors is presented in the following section.

V. Model Performance and Behavioral Analysis

Each benchmark scenario was executed using a selection of state-of-the-art LLMs to evaluate the impact of different language and code generation models on system performance. Language-related tasks, such as intent recognition, prompt refinement, and inter-agent communication, were handled by general-purpose instruction-following models, while code generation was managed by domain-specific models optimized for programming. This dual-model setup allowed for targeted analysis of each component in the simulation pipeline.

TABLE I
Benchmark Scenarios and Task Descriptions

Case 1: Host-to-Host Connectivity Check	Objective:	Validate the system's ability to interpret a basic connectivity testing request and generate a full connectivity matrix.
	Difficulty:	Easy
	Prompt:	*Verify connectivity between all hosts in the current topology using ping. Report which connections work and which do not.*
	Expected Behavior:	The agentic system is expected to identify all host nodes in the network, iterate over all unique host pairs, generate and execute the corresponding Mininet ping commands, and compile a comprehensive report summarizing packet loss and latency or each pair.
Case 2: Routing Optimization Between Subnets	Objective:	Evaluate the system's ability to handle mid-level planning tasks involving analysis and reconfiguration of routing paths based on performance metrics.
	Difficulty:	Medium
	Prompt:	*Implement more efficient routing between the 10.0.1.0/24 and 10.0.0.0/24 subnets to minimize latency.*
	Expected Behavior:	The system should identify which hosts belong to each subnet, analyze default routing paths, propose and apply optimized static or dynamic routes (e.g., using cost metrics or OSPF simulation), and perform latency measurements before and after optimization.
Case 3: Simulated DDoS Detection and Mitigation	Objective:	Test the system's ability to reason about malicious traffic, identify attack sources, and implement mitigation techniques while preserving legitimate access.
	Difficulty:	Hard
	Prompt:	*A DDoS attack is underway targeting server h4. Identify the nature of the attack and implement mitigation measures.*
	Expected Behavior:	The system should detect abnormal traffic targeting h4, identify its origin (e.g., via traffic monitoring or rate anomalies), and implement appropriate countermeasures such as interface-based rate limiting, flow table filtering, or temporary isolation of malicious hosts. Legitimate clients should retain access to h4.

The following models were used within the agent framework:

- *Language Models:* LLaMA 3.3 70B, Deepseek r1 distill LLaMA 70B, LLaMA 4 Scout 17B, OpenAI GPT-4o
- *Code Generation Models:* Qwen-Coder 2.5 (14B, 32B), CodeLLaMA (13B, 34B), Deepseek-Coder v2 16B

Each scenario was run multiple times under consistent conditions, alternating model configurations. Performance was evaluated based on accuracy, stability, reasoning, and task completion success.

A. Scenario 1: Host-to-Host Connectivity Check

This scenario required agents to reason over all host pairs and issue the appropriate `ping` commands via the Mininet Python API to build a full connectivity matrix and assess response times. All language models successfully extracted relevant host information and understood the user's intent. GPT-4o and LLaMA 4 Scout produced clear, structured commands with minimal ambiguity. Deepseek r1 occasionally introduced structural inconsistencies, while LLaMA 3.3 was consistent but less flexible with vague inputs. In code generation, Qwen-Coder 32B was the most reliable, producing correct, Mininet-compliant scripts with minimal hallucination. CodeLLaMA

34B also performed well but occasionally misinterpreted edge-case syntax. Despite higher initial error rates, smaller models like Deepseek-Coder v2 16B and Qwen-Coder 14B showed strong self-correction, especially when guided by runtime error feedback. Overall, most models handled this task effectively. While smaller models were less accurate initially, their ability to adapt helped maintain a high success rate.

B. Scenario 2: Routing Optimization Between Subnets

In this scenario, agents were tasked with analyzing a network topology containing distinct subnets and implementing more efficient routing to reduce latency. This required a solid understanding of topology structure, routing semantics, and the ability to validate improvements using simulated performance metrics. The language models performed similarly, accurately identifying subnet boundaries and the intent to optimize routing. As in the previous case, GPT-4o and LLaMA 4 Scout showed strong fluency in prompt decomposition, while LLaMA 3.3 remained reliable but slightly rigid. Deepseek r1 again showed occasional structural inconsistencies but generally understood the task. The code generation models varied more significantly. Qwen-Coder 32B was the most reliable, consistently generating valid scripts with proper routing commands and latency checks. In contrast, Deepseek-Coder v2 16B and CodeLLaMA 13B often produced invalid route commands or omitted key reconfiguration steps. A recurring issue in smaller code models, like Qwen-Coder 14B and Deepseek-Coder v2 16B, was falling into repetitive correction loops when facing errors, repeatedly patching code without resolving core issues. This behavior, while not universal, was frequent enough to limit their success. Overall, this scenario underscored the added complexity of routing tasks. Larger models showed stronger reasoning and reliability, while smaller ones struggled with stability and effective error recovery.

C. Scenario 3: Simulated DDoS Detection and Mitigation

This high-complexity scenario required the system to identify and mitigate a simulated DDoS attack targeting host `h4`. The task involved interpreting traffic behavior, detecting anomalies, and applying mitigation strategies such as traffic filtering, rate limiting, or flow control, while maintaining legitimate access. Language models performed consistently across variants. As before, GPT-4o and LLaMA 4 Scout offered structured, context-aware prompt decomposition, while LLaMA 3.3 and Deepseek distilled were reliable but less flexible with vague or loosely defined goals. Greater variability emerged in code generation models. Only the larger models, Qwen-Coder 32B, CodeLLaMA 34B, and Deepseek-Coder v2 16B, produced functional code simulating attacks and applying basic mitigation. However, some models failed to return user-readable output or ignored peripheral network components, resulting in incomplete or partial solutions. In several cases, agents detected the attack but failed to apply effective countermeasures. Smaller code models, such as Qwen-Coder 14B and CodeLLaMA 13B, consistently failed. They often entered infinite self-correction loops or terminated early without generating usable results. These failures suggest that multi-stage reasoning and adaptive planning exceed their capabilities. In general, larger models showed promising alignment with the task objectives, but often lacked comprehensive coverage or transparency. This scenario highlights the need for better long-horizon planning and feedback integration in agentic code generation workflows.

D. Comparative Summary and Limitations

Across all scenarios, a clear pattern emerged in the difficulty split between language-based and code-based tasks, as reported in Tables III and II. These tables report model performance across the three benchmark scenarios. Each Case column corresponds to a specific task: Case 1, Case 2, and Case 3. The scores range from 0 to 1 and reflect the degree of task completion, where 1.0 indicates full success.

Linguistic components, such as interpreting prompts, extracting entities from conversation history, and refining requests, were handled effectively by nearly all instruction-tuned language models (see Table II). These tasks align well with the current strengths of large language models, supported by years of progress in conversational and prompt-following benchmarks.

TABLE II
PERFORMANCE OF LANGUAGE LLMS IN PROMPT INTERPRETATION.

Model	Case 1	Case 2	Case 3	Intent	Flex	Adapt	Limitations
GPT-4o	1.0	1.0	1.0	Accurate	High	High	None
LLaMA4 17B	1.0	0.9	0.9	Accurate	Medium	High	Rigid
LLaMA3 70B	0.9	0.8	0.8	Reliable	Medium	Medium	Less flexible
Deepseek-r1 (LLaMA distill)	0.8	0.7	0.7	Moderate	Low	Medium	Structural issues

In contrast, the code generation and integration phase remains significantly more challenging, ad detailed in Table III. This is due in part to model capacity, smaller models struggle with logic consistency, error handling, and long-range dependencies needed to translate high-level goals into valid Mininet code. More critically, network simulation itself introduces procedural complexity. Tasks like routing reconfiguration or DDoS mitigation require staged execution, observation of intermediate results, and adaptive corrections, demands that go beyond static code generation and require strategic planning and state management, which smaller or non-specialized models often lack. Larger models showed more promise but still produced solutions that were often partial, brittle, or lacked deep reasoning.

Despite these challenges, the system is already suitable for educational purposes, small-scale simulations, and basic configuration tasks. It provides an interactive environment for exploring network behavior, learning Mininet syntax, and automating simple tests. However, for production-grade or industrial use, the system isn't yet mature enough to ensure reliability and robustness under complex conditions. These

TABLE III
PERFORMANCE OF CODING LLMS IN NETWORK SIMULATION TASKS.

Model	Case 1	Case 2	Case 3	Accuracy	Errors	Adapt	Limitations
Qwen 32B	1.0	1.0	0.8	Good	Moderate	High	Occasional hallucinations
CodeLLaMA 34B	1.0	0.7	0.6	Good	Moderate	Medium	Syntax errors
Deepseek coder 16B	0.8	0.5	0.5	Fair	Limited	Medium	Repetitive fixes
Qwen 14B	0.7	0.3	0.2	Partial	Weak	Low	Fails complex tasks
CodeLLaMA 13B	0.6	0.4	0.1	Partial	Weak	Low	Early termination

limitations point to future development areas, including multi-turn planning, persistent memory, simulation output interpretation, and better inter-agent coordination.

VI. DISCUSSION

While the proposed system shows strong potential in translating natural language instructions into network simulations, several enhancements are needed to improve robustness and handle more complex scenarios. Experimental results highlight that the main challenges lie not in language interpretation, where current models perform well, but in code generation, sequential execution, and context-aware reasoning.

Code generation remains a key bottleneck, especially for tasks involving multi-step logic. Future work will explore fine-tuning models on domain-specific datasets (e.g., annotated Mininet scripts) and incorporating planning agents to break down complex tasks into coherent action sequences. Enhancing feedback loops is also critical. The current system lacks mechanisms to interpret and respond to unexpected simulation outcomes. Adding structured error handling and recovery: blending rule-based and adaptive methods can improve resilience. Introducing real-time monitoring and reactive execution would enable dynamic scenario handling, essential for cases involving anomalies, routing changes, or fault mitigation. On the knowledge side, the existing RAG module is limited to Mininet documentation. Expanding it to include broader networking standards, protocols, and historical configurations will enrich the system's contextual understanding and autonomy.

Finally, expanding simulation realism is vital. While current benchmarks use simplified topologies, real-world use requires greater scale and complexity. Future development will explore integration with platforms like Containernet or NS-3 to support realistic testing and performance evaluation. These improvements aim to bridge the gap between experimental promise and real-world applicability, moving toward a reliable, autonomous tool for both educational and operational network management.

VII. CONCLUSION

This work presents a novel system that uses agentic AI and LLMs to translate natural language into executable network simulations. The approach enables intuitive, low-cost interaction with simulation tools, making them accessible to educators, students, and small enterprises. Experiments show strong performance in simple scenarios, though limitations persist in handling complex, multi-step configurations and dynamic conditions. Future work will focus on improving code generation, error handling, and adaptability to scale toward production-level use. The research marks a key step toward more intelligent and inclusive network simulation, highlighting agentic AI's potential to democratize access and streamline network design and testing.

REFERENCES

[1] C. Smera and J. Sandeep, "Networks simulation: Research based implementation using tools and approaches," in *2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT)*. IEEE, 2022, pp. 1–7.

[2] J. Pan and R. Jain, "A survey of network simulation tools: Current status and future developments," *Email: jp10@ cse. wustl. edu*, vol. 2, no. 4, p. 45, 2008.

[3] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.

[4] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, "Agentic large language models, a survey," *arXiv preprint arXiv:2503.23037*, 2025.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[6] J. Lee and K. Toutanova, "Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, vol. 3, no. 8, 2018.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] Y. Tian, T. Huang, M. Liu, D. Jiang, A. Spangher, M. Chen, J. May, and N. Peng, "Are large language models capable of generating human-level narratives?" *arXiv preprint arXiv:2407.13248*, 2024.

[9] O. G. Lira, O. M. Caicedo, and N. L. da Fonseca, "Large language models for zero touch network configuration management," *IEEE Communications Magazine*, 2024.

[10] M. El Rajab, L. Yang, and A. Shami, "Zero-touch networks: Towards next-generation network automation," *Computer Networks*, vol. 243, p. 110294, 2024.

[11] D. B. Acharya, K. Kuppan, and B. Divya, "Agentic ai: Autonomous intelligence for complex goals–a comprehensive survey," *IEEE Access*, 2025.

[12] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. pearson, 2016.

[13] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[14] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[15] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-wifi: Emulating software-defined wireless networks," in *2015 11th International conference on network and service management (CNSM)*. IEEE, 2015, pp. 384–389.

[16] J. Gomez, E. F. Kfoury, J. Crichigno, and G. Srivastava, "A survey on network simulators, emulators, and testbeds used for research and education," *Computer Networks*, vol. 237, p. 110054, 2023.

[17] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *Ieee Network*, vol. 32, no. 2, pp. 92–99, 2017.

[18] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, pp. 1–99, 2018.

[19] M. Wooldridge, *An introduction to multiagent systems*. John wiley & sons, 2009.

[20] M. A. Ridwan, N. A. M. Radzi, F. Abdullah, and Y. Jalil, "Applications of machine learning in networking: A survey of current issues and future challenges," *IEEE access*, vol. 9, pp. 52 523–52 556, 2021.