Enhancing Network Management Using Code Generated by Large Language Models

Sathiya Kumaran Mani[§] Yajie Zhou[§†] Kevin Hsieh[§]
Santiago Segarra[§⋆] Ranveer Chandra[§] Srikanth Kandula[§]

[§]*Microsoft Research* [†]*Boston University* [⋆]*Rice University*

arXiv:2308.06261v1 [cs.NI] 11 Aug 2023

ABSTRACT

Analyzing network topologies and communication graphs plays a crucial role in contemporary network management. However, the absence of a cohesive approach leads to a challenging learning curve, heightened errors, and inefficiencies. In this paper, we introduce a novel approach to facilitate a natural-language-based network management experience, utilizing large language models (LLMs) to generate task-specific code from natural language queries. This method tackles the challenges of explainability, scalability, and privacy by allowing network operators to inspect the generated code, eliminating the need to share network data with LLMs, and concentrating on application-specific requests combined with general program synthesis techniques. We design and evaluate a prototype system using benchmark applications, showcasing high accuracy, cost-effectiveness, and the potential for further enhancements using complementary program synthesis techniques.

1 Introduction

A critical aspect of contemporary network management involves analyzing and performing actions on network topologies and communication graphs for tasks such as capacity planning [39], configuration analysis [5, 17], and traffic analysis [24, 25, 60]. For instance, network operators may pose capacity planning questions, such as "What is the most cost-efficient way to double the network bandwidth between these two data centers?" using network topology data. Similarly, they may ask diagnostic questions like, "What is the required number of hops for data transmission between these two nodes?" using communication graphs. Network operators today rely on an expanding array of tools and domain-specific languages (DSLs) for these operations [17, 39]. A unified approach holds significant potential to reduce the learning curve and minimize errors and inefficiencies in manual operations.

The recent advancements in large language models (LLMs) [1,6,12,46,53] provide a valuable opportunity to carry out network management tasks using natural language. LLMs have demonstrated exceptional proficiency in interpreting human language and providing high-quality answers across various domains [16, 33, 50, 54]. The capabilities of LLMs can potentially bridge the gap between diverse tools and DSLs, leading to a more cohesive and user-friendly approach to handling network-related questions and tasks.

Unfortunately, while numerous network management operations can be represented as graph analysis or manipulation tasks, no existing systems facilitate graph manipulation using natural language. Asking LLMs to directly manipulate network topologies introduces three fundamental challenges related to explainability, scalability, and privacy. First, explaining the output of LLMs and enabling them to reason about complex problems remain unsolved issues [59]. Even state-of-the-art LLMs suffer from well-established problems such as hallucinations [35] and making basic arithmetic mistakes [7, 13]. This complicates the process of determining the methods employed by LLMs in deriving answers and evaluating their correctness. Second, LLMs are constrained by limited token window sizes [57], which restrict their capacity to process extensive network topologies and communication graphs. For example, state-of-the-art LLMs such as Bard [20], ChatGPT [44], and GPT-4 [46] permit only 2k to 32k tokens in their prompts, which can only accommodate small network topologies comprising tens of nodes and hundreds of edges. Third, network data may consist of personally identifiable information (PII), such as IP addresses [55], raising privacy concerns when transferring this information to LLMs for processing. Addressing these challenges is crucial to develop a more effective approach to integrating LLMs in network management tasks.

Vision and Techniques. In this paper, we present a novel approach to enhance network management by leveraging the power of LLMs to create *task-specific code for graph analysis and manipulation*, which facilitates a natural-language-based network administration experience. Figure 1 depicts an example of how this system generates and executes LLM-produced code in response to a network operator's natural language query. This approach tackles the explainability challenge by allowing network operators to examine the LLM-generated code, enabling them to comprehend the underlying logic and procedures to fulfill the natural language query. Additionally, it delegates computation to program execution engines, thereby minimizing arithmetic inaccuracies and LLM-induced hallucinations. Furthermore, this approach overcomes the scalability and privacy concerns by removing the necessity to transfer network data to LLMs, as the input for LLMs is

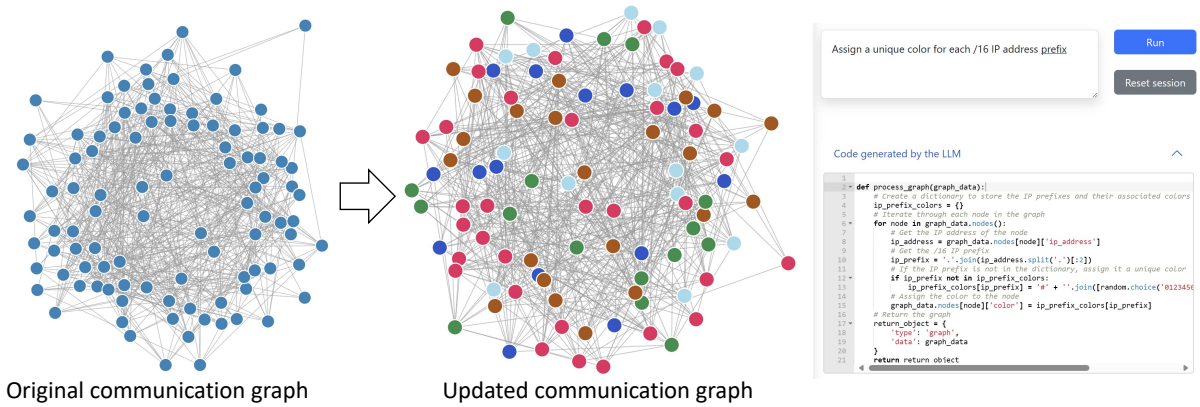


Figure 1: An example of how a natural-language-based network management system generates and executes a program in response to a network operator's query: "Assign a unique color for each /16 IP address prefix". The system displays the LLM-generated code and the updated communication graph.

the natural language query and the output solely comprises LLM-generated code.

The primary technical challenge in this approach lies in generating high-quality code that can reliably accomplish network management tasks. Although LLMs have demonstrated remarkable capabilities in general code generation [2, 7, 33], they lack an understanding of domain-specific and application-specific requirements. To tackle this challenge, we propose a novel framework that combines application-specific requests with general program synthesis techniques to create customized code for graph manipulation tasks in network management. Our architecture divides the process of generating high-quality code into two key components: (1) an application-specific element that provides context, instructions, or plugins, which enhances the LLMs' comprehension of network structures, attributes, and terminology, and (2) a code generation element that leverages suitable libraries and cutting-edge program synthesis techniques [2, 9–11, 48, 49] to produce code. This architecture fosters independent innovation of distinct components, and our preliminary study indicates substantial improvements in code quality.

Implementation and Evaluation. We design a prototype system that allows network operators to submit natural-language queries and obtain code to handle network topologies and communication graphs (Figure 1). To systematically assess effectiveness, we establish a benchmark, NeMoEval, consisting of two applications that can be modeled as graph manipulation: (1) network traffic analysis using communication graphs [24, 25, 60], and (2) network lifecycle management based on Multi-Abstraction-Layer Topology representation (MALT) [39]. To assess generalizability, we evaluate these applications using three code generation approaches (SQL [14], pandas [41], and NetworkX [15]) and four distinct LLMs [10, 20, 44, 46]. Our preliminary investigation shows that our system is capable of producing high-quality code for graph manipulation tasks. Utilizing the NetworkX-based approach, we attain average code correctness of 68% and 56% across all tasks for the four LLMs (up to 88% and 78% with GPT-4) for network traffic analysis and network lifecycle management, respectively. In comparison, the strawman baseline, which inputs the limited-sized graph data directly into LLMs, only reaches an average correctness of 23% for the traffic analysis application. Our method significantly improves the average correctness by 45%, making it a more viable option. Additionally, we demonstrate that integrating our system with complementary program synthesis methods could further enhance code quality for complex tasks. Finally, we demonstrate that our approach is cost-effective, with an average expense of $0.1 per task, and the LLM cost stays constant regardless of network sizes. Our study indicates that this is a potentially promising research direction. We release NeMoEval[1], our benchmark and datasets, to foster further research.

Contributions. We make the following contributions:

- Towards enabling natural-language-based network administration experience, we introduce a novel approach that uses LLMs to generate code for graph manipulation tasks. This work is, to the best of our knowledge, the first to investigate the utilization of LLMs for graph manipulation and network management.
- We develop and release a benchmark that encompasses two network administration applications: network traffic analysis and network lifecycle management.
- We evaluate these applications with three code generation techniques and four distinct LLMs to validate the capability of our approach in generating high-quality code for graph manipulation tasks.

2 Preliminaries

We examine graph analysis and manipulation's role in network management, followed by discussing recent LLM advances and their potential application to network management.

[1] https://github.com/microsoft/NeMoEval

2.1 Graph Analysis and Manipulation in Network Management

Network management involves an array of tasks such as network planning, monitoring, configuration, and troubleshooting. As networks expand in size and complexity, these tasks become progressively more challenging. For instance, network operators are required to configure numerous network devices to enforce intricate policies and monitor these devices to guarantee their proper functionality. Numerous operations can be modeled as graph analysis and manipulation for network topologies or communication graphs. Two examples of these tasks are described below.

Network Traffic Analysis. Network operators analyze network traffic for various reasons, including identifying bottlenecks, congestion points, and underutilized resources, as well as performing traffic classification. A valuable representation in traffic analysis is traffic dispersion graphs (TDGs) [25] or communication graphs [19], in which nodes represent network components like routers, switches, or devices, and edges symbolize the connections or paths between these components (e.g., Figure 1). These graphs offer a visual representation of data packet paths, facilitating a comprehensive understanding of traffic patterns. Network operators typically utilize these graphs in two ways: (1) examining these graphs to understand the network's current state for network performance optimization [25], traffic classification [52], and anomaly detection [29], and (2) manipulating the nodes and edges to simulate the impact of their actions on the network's performance and reliability [30].

Network Lifecycle Management. Managing the entire lifecycle of a network entails various phases, including capacity planning, network topology design, deployment planning, and diagnostic operations. The majority of these operations necessitate an accurate representation of network topology at different abstraction levels and the manipulation of topology to achieve the desired network state [39]. For example, network operators might employ a high-level topology to plan the network's capacity and explore different alternatives to increase bandwidth between two data centers. Similarly, network engineers may use a low-level topology to determine the location of a specific network device and its connections to other devices.

Hence, graph analysis and manipulation are crucial parts of network management. A unified interface capable of comprehending and executing these tasks has the potential to significantly simplify the process, saving network operators considerable time and effort.

2.2 LLMs and Program Synthesis

Automated program generation based on natural language descriptions, also known as program synthesis, has been a longstanding research challenge [3,23,34]. Until recently, program synthesis had primarily been limited to specific domains, such as string processing [22], program generation based on input-output examples [4], and natural language for database queries (e.g., [26, 28, 31]). In contrast, general program synthesis was considered to be out of reach [2]. The breakthrough emerged with the advancement of LLMs [6, 10, 18, 20, 32, 46], which are trained on extensive corpora of natural language text from the internet and massive code repositories such as GitHub. LLMs have demonstrated remarkable proficiency in learning the relationship between natural language and code, achieving state-of-the-art performance in domain-specific tasks such as natural language to database query [40, 51], as well as human-level performance in tasks like programming competitions [33] and mock technical interviews [7]. Just recently, these advancements have led to experimental plugins designed to solve mathematical problems and perform data analysis through code generation [43].

The recent breakthrough in program synthesis using LLMs has ignited a surge of research aimed at advancing the state of the art in this field. These techniques can generally be classified into three approaches: (1) code selection, which involves generating multiple samples with LLMs and choosing the best one based on the consistency of execution results [48] or auto-generated test cases [9]; (2) few-shot examples, which supply LLMs with several examples of the target program's input-output behavior [2]; and (3) feedback and self-reflection, which incorporates a feedback or reinforcement learning outer loop to help LLMs learn from their errors [8, 11, 49]. These advanced techniques continue to expand the horizons of program synthesis, empowering LLMs to generate more complex and accurate programs.

As Section 1 discusses, LLM-generated code can tackle explainability, scalability, and privacy challenges in LLM-based network management. However, our initial study shows that merely applying existing approaches is inadequate for network management tasks, as existing techniques do not comprehend the domain-specific and application-specific requirements. The key technical challenge lies in harnessing recent advancements in LLMs and general program synthesis to develop a unified interface capable of accomplishing network management tasks, which forms the design requirements for our proposed solution.

3 System Framework

We present a novel system framework designed to enhance network management by utilizing LLMs to generate task-specific code. Our framework is founded on two insights. First, we can transform many network management operations into graph analysis and manipulation tasks (Section 2.1), which allows for a unified design and a more focused task for code generation. Second, we can divide prompt generation into two aspects: domain-specific requirements and general program synthesis. By combining the strengths of domain specialization with recent advances in program synthesis techniques (Section 2.2), we can generate high-quality code for network management tasks. Figure 2 illustrates our system framework.

The framework we propose consists of an application wrapper (① in Figure 2) that uses domain-specific knowledge,

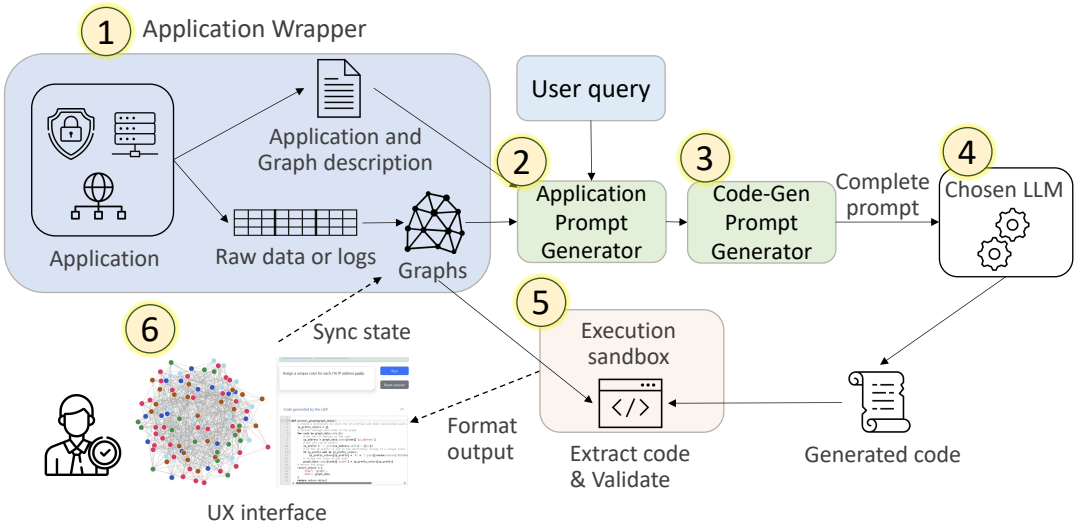


Figure 2: A general framework for network management systems using natural language and LLM-generated code

such as the definitions of nodes and edges, to transform the application data into a graph representation. This information, together with user queries in natural language, is processed by an application prompt generator (②) to create a task-specific prompt for the LLM. Subsequently, the task-specific prompt is combined with a general code-gen prompt generator (③) to instruct the LLM (④) to produce code. The generated code utilizes plugins and libraries to respond to the user's natural language queries in the constructed graph. An execution sandbox (⑤) executes the code on the graph representation of the network. The code and its results are displayed on a UX interface (⑥). If the user approves the results, the UX sends the updated graph back to the application wrapper (①) to modify the network state and record the input/output for future prompt enhancements [2, 11, 49]. We describe the key components below.

Application Wrapper (①). The application wrapper offers context-specific information related to the network management application and the network itself. For instance, the Multi-Abstraction-Layer Topology representation (MALT) wrapper [39] can extract the graph of entities and relationships from the underlying data, describing entities (e.g., packet switches, control points, etc.) and relationships (e.g., contains, controls, etc.) in natural language. This information assists LLMs in comprehending the network management application and the graph data structure. Additionally, the application wrapper can provide application-specific plugins [42] or code libraries to make LLM tasks more straightforward.

Application Prompt Generator (②). The purpose of the application prompt generator is to accept both the user query and the information from the application wrapper as input, and then generate a prompt specifically tailored to the query and task for the LLM. To achieve this, the prompt generator can utilize a range of static and dynamic techniques [37, 56, 58]. For instance, when working with MALT, the prompt generator can dynamically select relevant entities and relationships based on the user query, and then populate a prompt template with the contextual information. Our framework is designed to offer flexibility regarding the code-gen prompt generator (③) and LLMs (④), enabling the use of various techniques for different applications.

Execution Sandbox (⑤). As highlighted in previous research [10], it is crucial to have a secure environment to run the code generated by LLMs. The execution sandbox can be established using virtualization or containerization techniques, ensuring limited access to program libraries and system calls. Additionally, this module provides a chance to enhance the security of both code and system by validating network invariants or examining output formats.

4 Implementation and Evaluation

4.1 Benchmark

We design a general benchmark, NeMoEval, to evaluate the effectiveness of LLM-based network management systems. Figure 3 illustrates the architecture of our benchmark, which consists of three primary components:

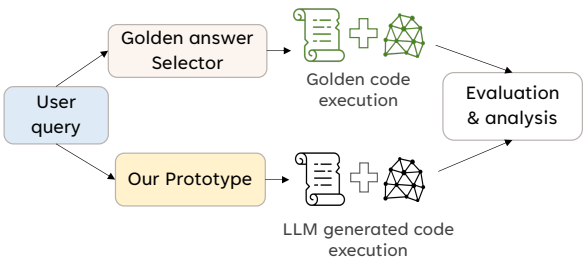


Figure 3: Benchmark design

Golden Answer Selector. For each input user query, we create a "golden answer" with the help of human experts.

Table 1: User query examples. See all queries in NeMoEval.

Complexity level	Traffic Analysis	MALT
Easy	Add a label app:prodution to nodes with address prefix 15.76	List all ports that are contained by packet switch ju1.a1.m1.s2c1.
Medium	Assign a unique color for each /16 IP address prefix.	Find the first and the second largest Chassis by capacity.
Hard	Calculate total byte weight on each node, cluster them into 5 groups.	Remove packet switch P1 from Chassis 4, balance the capacity afterward.

These verified answers, stored in a selector's dictionary file, act as the ground truth to evaluate LLM-generated code.

Results Evaluator. The system executes the LLM-generated code on network data, comparing outcomes with the golden answer's results. If they match, the LLM passes; otherwise, it fails, and we document the findings for further analysis.

Results Logger. To facilitate the analysis of the LLM's performance and the identification of potential improvement, we log the results of each query, including the LLM-generated code, the golden answer, and the comparison results. The results logger also records any code execution errors that may have occurred during the evaluation process.

4.2 Experimental Setup

Applications and Queries. We implement and evaluate two applications, network traffic analysis and network lifecycle management (Section 2.1):

- *Network Traffic Analysis.* We generate synthetic communication graphs with varying numbers of nodes and edges. Each edge represents communication activities between two nodes with random weights in bytes, connections, and packets. We develop 24 queries by curating trial users' queries, encompassing common tasks such as topology analysis, information computation, and graph manipulation.
- *Network Lifecycle Management.* We use the example MALT dataset [21] and convert it into a directed graph with 5493 nodes and 6424 edges. Each node represents one or more types in a network, such as packet switches, chassis, and ports, with different node types containing various attributes. Directed edges encapsulate relationships between devices, like control or containment associations. We develop 9 network management queries focusing on operational management, WAN capacity planning, and topology design.

The queries are categorized into three complexity levels ("Easy", "Medium", and "Hard") based on the complexity of their respective golden answers. Table 1 displays an example query from each category due to page limits. We release the complete list of queries, their respective golden answers, and the benchmark to facilitate future research[2].

LLMs. We conduct our study on four state-of-the-art LLMs, including GPT-4 [46], GPT-3 [6], Text-davinci-003 (a variant of GPT 3.5) [45], and Google Bard [20]. We further explore two open LLMs, StarCoder [32] and InCoder [18]. However, we do not show their results here because of inconsistent answers. We intend to report their results once they achieve consistent performance in future investigation. With all OpenAI LLMs, we set their temperature to 0 to ensure consistent output across multiple trials. Since we cannot change the temperature of Google Bard, we send each query five times and calculate the average passing probability [10].

[2] https://github.com/microsoft/NeMoEval

Table 2: Accuracy Summary for Both Applications

	Traffic Analysis				MALT		
	Strawman	SQL	Pandas	NetworkX	SQL	Pandas	NetworkX
GPT-4	0.29	0.50	0.38	**0.88**	0.11	0.56	**0.78**
GPT-3	0.17	0.13	0.25	**0.63**	0.11	**0.44**	**0.44**
text-davinci-003	0.21	0.29	0.29	**0.63**	0.11	0.22	**0.56**
Google Bard	0.25	0.21	0.25	**0.59**	0.11	0.33	**0.44**

Table 3: Breakdown for Trafic Analysis

	Strawman	SQL	Pandas	NetworkX
	E(8)/M(8)/H(8)	E(8)/M(8)/H(8)	E(8)/M(8)/H(8)	E(8)/M(8)/H(8)
GPT-4	0.50/0.38/0.0	0.75/0.50/0.25	0.50/0.50/0.13	1.0/1.0/0.63
GPT-3	0.38/0.13/0.0	0.25/0.13/0.0	0.50/0.25/0.0	1.0/0.63/0.25
text-davinci-003	0.38/0.25/0.0	0.63/0.25/0.0	0.63/0.25/0.0	1.0/0.75/0.13
Google Bard	0.50/0.25/0.0	0.38/0.25/0.0	0.50/0.13/0.13	0.88/0.50/0.38

Approaches. We implement three code generation methods using well-established data/graph manipulation libraries, which offer abundant examples in public code repositories for LLMs to learn from:

- *NetworkX.* We represent the network data as a NetworkX [15] graph, which offers flexible APIs for efficient manipulation and analysis of network graphs.
- *pandas.* We represent the network data using two pandas [41] dataframes: a node dataframe, which stores node indices and attributes, and an edge dataframe, which encapsulates the link information among nodes through an edge list. Pandas provides many built-in data manipulation techniques, such as filtering, sorting, and grouping.
- *SQL.* We represent the network data as a relational database queried through SQL [14], consisting of a table for nodes and another for edges. The table schemas are similar to those in pandas. Recent work has demonstrated that LLMs are capable of generating SQL queries with state-of-the-art accuracy [40, 51].

We also evaluate an alternative baseline (*strawman*) that directly feeds the original network graph data in JSON format to the LLM and requests it to address the query. However, owing to the token constraints on LLMs, we limit our evaluation of this approach to synthetic graphs for network traffic analysis, where data size can be controlled.

Table 4: Breakdown for MALT

	SQL	Pandas	NetworkX
	E(3)/M(3)/H(3)	E(3)/M(3)/H(3)	E(3)/M(3)/H(3)
GPT-4	0.33/0.0/0.0	0.67/0.67/0.33	1.0/1.0/0.33
GPT-3	0.33/0.0/0.0	0.67/0.67/0.0	0.67/0.67/0.0
text-davinci-003	0.33/0.0/0.0	0.33/0.33/0.0	0.67/0.67/0.33
Google Bard	0.33/0.0/0.0	0.67/0.33/0.0	0.67/0.33/0.33

Table 5: Error Type Summary of LLM Generated Code

LLM's error type (NetworkX)	Traffic Analysis (35)	MALT (17)
Syntax error	9	0
Imaginary graph attributes	9	1
Imaginary files/function arguments	3	2
Arguments error	7	8
Operation error	4	2
Wrong calculation logic	2	3
Graphs are not identical	1	1

4.3 Code Quality

Table 2 summarizes the benchmark results for network traffic analysis and network lifecycle management, respectively. We observe three key points. First, utilizing LLMs for generating code in network management significantly surpasses the strawman baseline in both applications, as the generated code reduces arithmetic errors and LLM hallucinations. Second, employing a graph library (NetworkX) greatly enhances code accuracy compared to pandas and SQL, as LLMs can leverage NetworkX's graph manipulation APIs to simplify the generated code. This trend is consistent across all four LLMs. Finally, pairing NetworkX with the state-of-the-art GPT-4 model produces the highest results (88% and 78%, respectively), making it a promising strategy for network management code generation.

To understand the impact of task difficulty, we break down the accuracy results in Tables 3 and 4. We observe that the accuracy of LLM-generated code decreases as task complexity increases. This trend is consistent across all LLMs and approaches, with the performance disparities becoming more pronounced for network lifecycle management (Table 4).

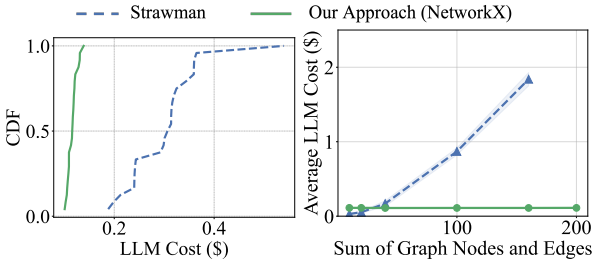
Our analysis of the LLM-generated code reveals that the complex relationships in the MALT dataset make LLMs more prone to errors in challenging tasks, and future research should focus on improving LLMs' ability to handle complex network management tasks.

4.4 Case Study on Potential Improvement

For the NetworkX approach across all four LLMs, there are 35 failures out of 96 tests (24×4) for network traffic analysis and 17 failures out of 36 tests (9×4) for network lifecycle management, respectively. Table 5 summarizes the error types. More than half of the errors are associated with syntax errors or imaginary (non-existent) attributes. We conduct a case study to see whether using complementary program synthesis techniques (Section 2.2) could correct these errors.

Table 6: Improvement Cases with Bard on MALT

	Bard + Pass@1	Bard + Pass@5	Bard + Self-debug
NetworkX	0.44	1.0	0.67



(a) CDF of LLM cost per query (80 nodes and edges) **(b) Cost analysis on graph size**

Figure 4: Cost and scalability Analysis

We assess two techniques: (1) pass@k [10], where the LLM is queried k times with the same question, and it is deemed successful if at least one of the answers is correct. This method reduces errors arising from the LLM's inherent randomness and can be combined with code selection techniques [9, 10, 48] for improved results; (2) self-debug [11], which involves providing the error message back to the LLM and encouraging it to correct the previous response.

We carry out a case study using the Bard model and three unsuccessful network lifecycle queries with the NetworkX approach. Table 6 shows that both pass@k ($k = 5$) and self-debug significantly enhance code quality, resulting in improvements of 100% and 67%, respectively. These results indicate that applying complementary techniques has considerable potential for further improving the accuracy of LLM-generated code in network management applications.

4.5 Cost and Scalability Analysis

We examine the LLM cost utilizing GPT-4 pricing on Azure [36] for the network traffic analysis application. Figure 4a reveals that the strawman approach is three times costlier than our method for a small graph with 80 nodes and edges. As the graph size expands (Figure 4b), the gap between the two approaches grows, with the strawman approach surpassing the LLM's token limit for a moderate graph containing 150 nodes and edges. Conversely, our method has a small cost (<$0.2 per query) that remains unaffected by graph size increases.

5 Discussion and Conclusion

Recent advancement in LLMs has paved the way for new opportunities in network management. We introduce a system framework that leverages LLMs to create task-specific code for graph manipulation, tackling issues of explainability, scalability, and privacy. While our prototype and preliminary study indicate the potential of this method, many open questions remain in this nascent area of research.

Code Quality for Complex Tasks. As our evaluation demonstrates, the LLM-generated code is highly accurate for easy

and medium tasks; however, the accuracy decreases for more complex tasks. This is partially due to the LLMs being trained on a general code corpus without specific network management knowledge. An open question is how to develop domain-specific program synthesis techniques capable of generating high-quality code for complex network management tasks, such as decomposing the task into simpler sub-tasks [56], incorporating application-specific plugins [42], or fine-tuning the model with application-specific code examples.

Code Comprehension and Validation. Ensuring correctness and understanding LLM-generated code can be challenging for network operators. While general approaches like LLM-generated test cases [9] and code explanation [38] exist, they are insufficient for complex tasks. Developing robust, application-specific methods to aid comprehension and validation is a crucial challenge.

Expanding Benchmarks and Applications. Extending our current benchmark to cover more network management tasks raises questions about broader effectiveness and applicability to other applications, such as network failure diagnosis [27,47] and configuration verification [5, 17]. Addressing these challenges requires exploring new network state representation, code generation strategies, and application-specific libraries and plugins.

In summary, we take a pioneering step in introducing a general framework to use LLMs in network management, presenting a new frontier for simplifying network operators' tasks. We hope that our work, along with our benchmarks and datasets, will stimulate continued exploration in this field.

6 References

[1] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, E. Chu, J. H. Clark, L. E. Shafey, Y. Huang, K. Meier-Hellstern, G. Mishra, E. Moreira, M. Omernick, K. Robinson, S. Ruder, Y. Tay, K. Xiao, Y. Xu, Y. Zhang, G. H. Ábrego, J. Ahn, J. Austin, P. Barham, J. A. Botha, J. Bradbury, S. Brahma, K. Brooks, M. Catasta, Y. Cheng, C. Cherry, C. A. Choquette-Choo, A. Chowdhery, C. Crepy, S. Dave, M. Dehghani, S. Dev, J. Devlin, M. Díaz, N. Du, E. Dyer, V. Feinberg, F. Feng, V. Fienber, M. Freitag, X. Garcia, S. Gehrmann, L. Gonzalez, and et al. PaLM 2 technical report. *CoRR*, abs/2305.10403, 2023.

[2] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[3] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *The 1957 western joint computer conference: Techniques for reliability (IRE-AIEE-ACM)*, 1957.

[4] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. DeepCoder: Learning to write programs. In *Proceedings of 5th International Conference on Learning Representations (ICLR)*, 2017.

[5] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[7] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

[8] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez. Improving code generation by training with natural language feedback. *CoRR*, abs/2303.16749, 2023.

[9] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen. CodeT: Code generation with generated tests. *CoRR*, abs/2207.10397, 2022.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[11] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. *CoRR*, abs/2304.05128, 2023.

[12] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. PaLM: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[13] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

[14] C. J. Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[15] N. Developers. NetworkX: Network analysis in Python. `https://networkx.org/`, Retrieved on 2023-02.

[16] T. Eloundou, S. Manning, P. Mishkin, and D. Rock. GPTs are GPTs: An early look at the labor market impact potential of large language models. *CoRR*, abs/2303.10130, 2023.

[17] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. D. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[18] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. Yih, L. Zettlemoyer, and M. Lewis. InCoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022.

[19] E. Glatz, S. Mavromatidis, B. Ager, and X. A. Dimitropoulos. Visualizing big network traffic data using frequent pattern mining and hypergraphs. *Computing*, 96(1):27–38, 2014.

[20] Google. Google Bard. `https://bard.google.com/`, Retrieved on 2023-06.

[21] Google. MALT example models. `https://github.com/google/malt-example-models`, Retrieved on 2023-06.

[22] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[23] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2), 2017.

[24] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and

W. Willinger. Sonata: query-driven streaming network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[25] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (TDGs). In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference (IMC)*, 2007.

[26] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2017.

[27] S. Kandula, D. Katabi, and J. Vasseur. Shrink: a tool for failure diagnosis in IP networks. In *Proceedings of the 1st Annual ACM Workshop on Mining Network Data (MineNet)*, 2005.

[28] H. Kim, B. So, W. Han, and H. Lee. Natural language to SQL: where are we today? *Proc. VLDB Endow.*, 13(10), 2020.

[29] D. Q. Le, T. Jeong, H. E. Roman, and J. W. Hong. Traffic dispersion graph based anomaly detection. In *Proceedings of the Symposium on Information and Communication Technology (SoICT)*, 2011.

[30] S. Lee, K. Levanti, and H. S. Kim. Network monitoring: Present and future. *Comput. Networks*, 65, 2014.

[31] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1), 2014.

[32] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Moustafa-Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. StarCoder: may the source be with you! *CoRR*, abs/2305.06161, 2023.

[33] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with AlphaCode. *CoRR*, abs/2203.07814, 2022.

[34] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3), 1971.

[35] J. Maynez, S. Narayan, B. Bohnet, and R. T. McDonald. On faithfulness and factuality in abstractive summarization. In D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[36] Microsoft. Azure OpenAI service pricing. `https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/`, Retrieved on 2023-06.

[37] Microsoft. A guidance language for controlling large language models. `https://github.com/microsoft/guidance`, Retrieved on 2023-06.

[38] Microsoft. Introducing GitHub Copilot X. `https://github.com/features/preview/copilot-x`, Retrieved on 2023-06.

[39] J. C. Mogul, D. Goricanec, M. Pool, A. Shaikh, D. Turk, B. Koley, and X. Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[40] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W. Yih, S. I. Wang, and X. V. Lin. LEVER: learning to verify language-to-code generation with execution. *CoRR*, abs/2302.08468, 2023.

[41] NumFOCUS. pandas. `https://pandas.pydata.org/`, Retrieved on 2023-06.

[42] OpenAI. ChatGPT plugins. `https://openai.com/blog/chatgpt-plugins`, Retrieved on 2023-05.

[43] OpenAI. Code interpreter. `https://openai.com/blog/chatgpt-plugins#code-interpreter`, Retrieved on 2023-08.

[44] OpenAI. Introducing ChatGPT. `https://openai.com/blog/chatgpt`, Retrieved on 2023-02.

[45] OpenAI. OpenAI models. `https://platform.openai.com/docs/models/overview`, Retrieved on 2023-06.

[46] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[47] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren. Passive realtime datacenter fault detection and localization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[48] F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang. Natural language to code translation with execution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[49] N. Shinn, F. Cassano, B. Labash, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *CoRR*, abs/2303.11366, 2023.

[50] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. K. Tanwani, H. Cole-Lewis, S. Pfohl, P. Payne, M. Seneviratne, P. Gamble, C. Kelly, N. Schärli, A. Chowdhery, P. A. Mansfield, B. A. y Arcas, D. R. Webster, G. S. Corrado, Y. Matias, K. Chou, J. Gottweis, N. Tomasev, Y. Liu, A. Rajkomar, J. K. Barral, C. Semturs, A. Karthikesalingam, and V. Natarajan. Large language models encode clinical knowledge. *CoRR*, abs/2212.13138, 2022.

[51] R. Sun, S. Ö. Arik, H. Nakhost, H. Dai, R. Sinha, P. Yin, and T. Pfister. SQL-PaLM: Improved large language model adaptation for text-to-SQL. *CoRR*, abs/2306.00739, 2023.

[52] H. Tahaei, F. Afifi, A. Asemi, F. Zaki, and N. B. Anuar. The rise of traffic classification in iot networks: A survey. *J. Netw. Comput. Appl.*, 154:102538, 2020.

[53] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. LLaMA: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[54] I. Trummer. CodexDB: Synthesizing code for query processing from natural language instructions using GPT-3 codex. *Proc. VLDB Endow.*, 15(11), 2022.

[55] E. Union. General data protection regulation (GDPR). `https://commission.europa.eu/law/law-topic/data-protection_en`, Retrieved on 2023-04.

[56] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[57] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *CoRR*, abs/2303.07839, 2023.

[58] Z. Zhang, A. Zhang, M. Li, and A. Smola. Automatic chain of thought prompting in large language models. *CoRR*, abs/2210.03493, 2022.

[59] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J. Nie, and J. Wen. A survey of large language models. *CoRR*, abs/2303.18223, 2023.

[60] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.