Enhancing Network Management Using Code Generated by Large Language Models

Sathiya Kumaran Mani[§] Yajie Zhou[§†] Kevin Hsieh[§] Santiago Segarra[§⋆]
Trevor Eberl[§] Eliran Azulai[§] Ido Frizler[§] Ranveer Chandra[§] Srikanth Kandula[§]
[§]*Microsoft* [†]*University of Maryland* [⋆]*Rice University*

Abstract

Analyzing network topologies and communication graphs is essential in modern network management. However, the lack of a cohesive approach results in a steep learning curve, increased errors, and inefficiencies. In this paper, we present a novel approach that enables natural-language-based network management experiences, leveraging large language models (LLMs) to generate task-specific code from natural language queries. This method addresses the challenges of explainability, scalability, and privacy by allowing network operators to inspect the generated code, removing the need to share network data with LLMs, and focusing on application-specific requests combined with program synthesis techniques. We develop and evaluate a prototype system using benchmark applications, demonstrating high accuracy, cost-effectiveness, and potential for further improvements using complementary program synthesis techniques.

CCS Concepts

• **Networks** → **Network management**;

Keywords

Network management; Large language model; Program synthesis; Natural language processing; Graph manipulation; Communication graphs; Network lifecycle management

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 number of hops for data transmission between these two nodes?" using communication graphs. Network operators today rely on an 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.

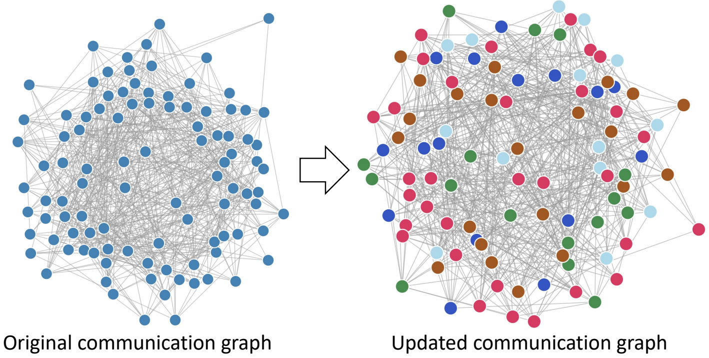
The recent advancements in large language models (LLMs) [1, 6, 12, 46, 53] provide a valuable opportunity to carry out network management tasks using natural language. LLMs have demonstrated exceptional proficiency in interpreting human language and providing high-quality answers across various domains [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.

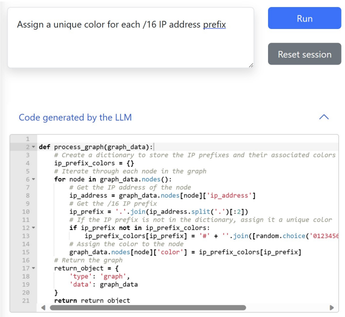
Unfortunately, while numerous network management operations can be modeled as graph analysis or manipulation tasks, no existing systems facilitate graph manipulation using natural language. Asking LLMs to directly manipulate network topologies introduces three fundamental challenges: explainability, 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 problems such as hallucinations [35] and arithmetic errors [7, 13]. This makes it hard to assess LLMs' methods and answers. Second, LLMs are constrained by limited token window sizes [57], which restrict their capacity to process extensive network topologies and communication graphs. For example, modern 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 with tens of nodes and hundreds of edges. Third, network data may contain 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 integrate LLMs in network management tasks.

Vision and Techniques. We present a novel approach to enhance network management by leveraging LLMs to create *task-specific code for graph analysis and manipulation*, which facilitates a natural-language-based network administration

HotNets '23, November 28–29, 2023, Cambridge, MA, USA

ACM ISBN 979-8-4007-0415-4/23/11…$15.00
https://doi.org/10.1145/3626111.3628183





Assign a unique color for each /16 IP address prefix

```

def process_graph(graph_data):
    # Create a dictionary to store the IP prefixes and their associated colors
    ip_prefix_colors = {}
    # Iterate through each node in the graph
    for node in graph_data.nodes():
        # Get the IP address of the node
        ip_address = graph_data.nodes[node]['ip_address']
        # Get the /16 IP prefix
        ip_prefix = '.'.join(ip_address.split('.')[:2])
        # If the IP prefix is not in the dictionary, assign it a unique color
        if ip_prefix not in ip_prefix_colors:
            ip_prefix_colors[ip_prefix] = '#' + ''.join([random.choice('0123456
        # Assign the color to the node
        graph_data.nodes[node]['color'] = ip_prefix_colors[ip_prefix]
    # Return the graph
    return_object = {
        'type': 'graph',
        'data': graph_data
    }
    return return_object
```

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.

experience. Figure 1 depicts how an example 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 that fulfills the query. Additionally, it delegates computation to program execution engines, thereby minimizing arithmetic inaccuracies and LLM-induced hallucinations. Furthermore, this approach overcomes scalability and privacy issues by removing the need to share network data with LLMs.

The primary technical challenge lies in generating high-quality code to accomplish network management tasks. Although LLMs have shown remarkable capabilities in general code generation [2, 7, 33], they lack an understanding of domain 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 code generation into two components: (1) an application-specific element that provides context, instructions, or plugins, which enhance the LLMs' comprehension of network structures and terminology, and (2) a code generation element that leverages suitable libraries and program synthesis techniques [2, 9–11, 48, 49]. This architecture fosters independent innovation of the distinct components, and our preliminary study indicates substantial code quality improvements.

Implementation and Evaluation. We design a prototype system that allows network operators to submit natural-language queries to network topologies and communication graphs (Figure 1). To 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 three code generation approaches (SQL [14], pandas [41], and NetworkX [15]) and four distinct LLMs [10, 20, 44, 46]. Our preliminary results show that our system produces high-quality code. Utilizing the NetworkX-based approach, we attain average code correctness (i.e., code with correct functionality for the query) of 63% 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 graph data 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 adding complementary program synthesis methods could further enhance code quality. 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. 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 usage 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 our approach for generating high-quality code for graph manipulation.

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

2 Preliminaries

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

2.1 Graph Analysis and Manipulation in Network Management

Network management involves 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 and monitor numerous devices to enforce intricate policies and ensure proper functionality. Numerous operations can be modeled as graph analysis and manipulation for network topologies or communication graphs. Two examples are described below.

Network Traffic Analysis. Network operators analyze traffic to identify bottlenecks, congestion, and underused resources, as well as for 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 a network's lifecycle involves phases like capacity planning, topology design, deployment, and diagnostics. Most operations require precise topology representations at various abstraction levels and the manipulation of topology to achieve the desired network state [39]. For example, network operators may employ a high-level topology to plan the network's capacity and explore 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 for 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, also known as program synthesis, has been a long-standing research challenge [3, 23, 34]. Until recently, program synthesis had primarily been limited to specific domains, such as string processing [22], programs based on input-output examples [4], and 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 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]. 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 and application specific requirements. The key technical challenge lies in harnessing advancements in LLMs and general program synthesis to develop a unified interface to accomplish network management tasks, which forms the design requirements for our proposed solution.

3 System Framework

We present a novel general framework to enhance network management by utilizing LLMs to generate task-specific code. Our framework is based 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 separate prompt generation into two aspects: domain-specific requirements and general program synthesis. By combining the strengths of domain specialization with advances in program synthesis techniques (Section 2.2), we can generate high-quality code for network management tasks. Figure 2 illustrates our system framework.

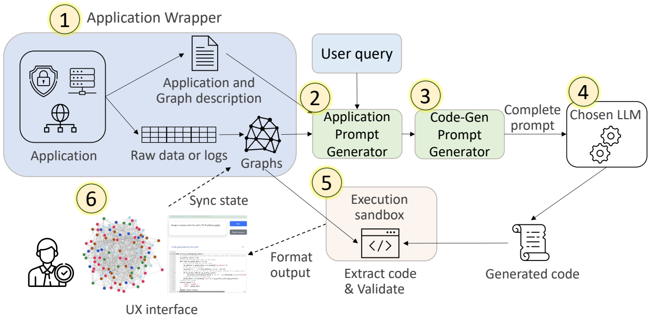


Figure 2: A general framework for network management systems using LLM-generated code

Our framework consists of an application wrapper (① in Figure 2) that uses domain-specific knowledge, 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, which can be generated with templates and query contexts. 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 user's queries in the constructed graph. An execution sandbox (⑤) executes 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 UX sends the updated graph 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.

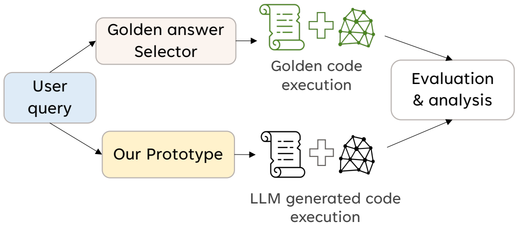


Figure 3: Benchmark design

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 benchmark, NeMoEval, to evaluate LLM-based network management systems. As Figure 3 illustrates, NeMoEval consists of three main components:

Golden Answer Selector. For each input user query, we create a "golden answer" which contains the expected correct code functionality with the help of human experts. 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 in the sandbox, comparing outcomes (e.g., an updated graph or the output information) with the golden answer's executed results.

Results Logger. To analyze the LLM's performance and improvement potential, we log each query's results, including the LLM-generated code, the golden answer, and the comparison. We also record any code execution errors.

4.2 Experimental Setup

Applications and Queries. We implement and evaluate two applications as described in Section 2.1:

- *Network Traffic Analysis.* We generate synthetic communication graphs with varying numbers of nodes and edges. Each edge represents communication between two nodes with weights in bytes, connections, and packets. By curating trial queries from product users, we develop 24 queries encompassing tasks such as topology analysis, information computation, and graph manipulation.

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.

- *Network Lifecycle Management.* We convert the example MALT dataset [21] to 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. Based on the examples in the MALT paper [39], we develop 9 network management queries that consist of operational management, WAN capacity planning, and topology design.

Queries are grouped into three levels (“Easy”, “Medium”, and “Hard”) based on the complexity of their golden answers. Table 1 displays an example query from each category due to page limits. The release of NeMoEval[1] contains the complete list of queries and their respective golden answers.

LLMs. We study four leading LLMs: GPT-4 [46], GPT-3 [6], Text-davinci-003 [45], and Google Bard [20]. We also test two open LLMs, StarCoder [32] and InCoder [18], but omit their results due to their inconsistency. We will report on them once they are stabilized. We set the temperature of OpenAI LLMs to 0 for uniform output across trials. Since we cannot change the temperature of Google Bard, we send each query five times and calculate the average passing probability [10].

Approaches. We implement three code generation methods for LLMs using well-established data/graph libraries with numerous public code examples 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 databases queried through SQL [14], consisting of a table for nodes and another for edges. The schemas are similar to those in pandas. Recent work has demonstrated that LLMs are capable of generating SQL 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 LLMs and requests them to address the query. However, owing to the token limitations on LLMs, we limit our evaluation of this approach to synthetic graphs for network traffic analysis, where data size can be controlled.

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	0.25	0.13	0.25	**0.46**	0.11	0.33	**0.44**
text-davinci-003	0.21	0.29	0.29	**0.58**	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	1.0/0.88/0.75
GPT-3	0.38/0.38/0.0	0.25/0.13/0.0	0.50/0.25/0.0	0.63/0.50/0.25
text-davinci-003	0.38/0.25/0.0	0.63/0.25/0.0	0.63/0.25/0.0	1.0/0.63/0.13
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

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	0.67/0.33/0.0	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

4.3 Code Quality

Table 2 summarizes the code correctness results. We observe three key points. First, utilizing LLMs for generating code in network management significantly outperforms 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 directly map natural-language graph operations to NetworkX’s graph manipulation APIs, which simplifies 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 best 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

Table 5: Error Type Summary of LLM Generated Code

LLM's error type (NetworkX)	Traffic Analysis (36)	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	2	1

Table 6: Improvement Cases with Bard on MALT

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

should focus on improving LLMs' ability to handle complex network management tasks.

4.4 Case Study on Potential Improvement

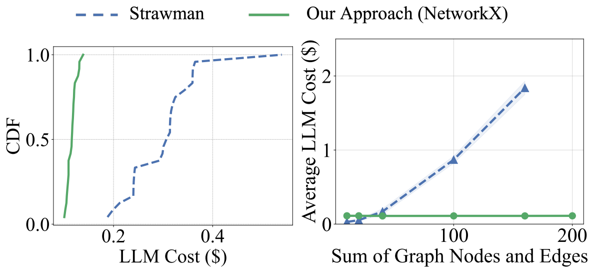
For the NetworkX approach across all four LLMs, there are 36 failures out of 96 tests (24 × 4) for network traffic analysis and 17 failures out of 36 tests (9 × 4) for network lifecycle management, respectively. Table 5 summarizes the error types. More than half of the errors are associated with syntax 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.

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 strawman 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 (~$0.1 per query) that remains unaffected by graph size increases.



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

Figure 4: Cost and scalability Analysis

5 Discussion and Conclusion

Recent advancements in LLMs have 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 shows, the LLM-generated code is highly accurate for easy and medium tasks; however, the accuracy decreases for 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 conclusion, we introduce 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.

References

[1] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernández Ábrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan A. Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vladimir Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, and et al. 2023. PaLM 2 Technical Report. *CoRR* abs/2305.10403 (2023).

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021).

[3] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, Irving Ziller, Robert A. Hughes, and R. Nutt. 1957. The FORTRAN automatic coding system. In *The 1957 western joint computer conference: Techniques for reliability (IRE-AIEE-ACM).*

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of 5th International Conference on Learning Representations (ICLR).*

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM).*

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS).*

[7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR* abs/2303.12712 (2023).

[8] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving Code Generation by Training with Natural Language Feedback. *CoRR* abs/2303.16749 (2023).

[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. *CoRR* abs/2207.10397 (2022).

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

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023).

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

[13] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021).

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

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

[16] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models. *CoRR* abs/2303.10130 (2023).

[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).

[19] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager, and Xenofontas A. Dimitropoulos. 2014. Visualizing big network traffic data using frequent pattern mining and hypergraphs. *Computing* 96, 1 (2014), 27–38.

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

[23] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017).

[24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: query-driven streaming network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM).*

[25] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. 2007. Network monitoring using traffic dispersion graphs (TDGs). In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference (IMC)*.

[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

[28] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proc. VLDB Endow.* 13, 10 (2020).

[29] Do Quoc Le, Taeyoel Jeong, H. Eduardo Roman, and James Won-Ki Hong. 2011. Traffic dispersion graph based anomaly detection. In *Proceedings of the Symposium on Information and Communication Technology (SoICT)*.

[30] Sihyung Lee, Kyriaki Levanti, and Hyong S. Kim. 2014. Network monitoring: Present and future. *Comput. Networks* 65 (2014).

[31] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (2014).

[32] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023).

[33] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022).

[34] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (1971).

[35] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan T. McDonald. 2020. On Faithfulness and Factuality in Abstractive Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[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] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. 2020. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[40] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. 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. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[47] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. 2017. Passive Realtime Datacenter Fault Detection and Localization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[48] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural Language to Code Translation with Execution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[49] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *CoRR* abs/2303.11366 (2023).

[50] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Kumar Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Schärli, Aakanksha Chowdhery, Philip Andrew Mansfield, Blaise Agüera y Arcas, Dale R. Webster, Gregory S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle K. Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. 2022. Large Language Models Encode Clinical Knowledge. *CoRR* abs/2212.13138 (2022).

[51] Ruoxi Sun, Sercan Ö. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *CoRR* abs/2306.00739 (2023).

[52] Hamid Tahaei, Firdaus Afifi, Adeleh Asemi, Faiz Zaki, and Nor Badrul Anuar. 2020. The rise of traffic classification in IoT networks: A survey. *J. Netw. Comput. Appl.* 154 (2020), 102538.

[53] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[54] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *Proc. VLDB Endow.* 15, 11 (2022).

[55] European Union. General Data Protection Regulation (GDPR). https://commission.europa.eu/law/law-topic/data-protection_en, Retrieved on 2023-04.

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[57] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. *CoRR* abs/2303.07839 (2023).

[58] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic Chain of Thought Prompting in Large Language Models. *CoRR* abs/2210.03493 (2022).

[59] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023).

[60] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. Flow Event Telemetry on Programmable Data Plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.