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
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 NeMoEval1, 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.

1https://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.
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.

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

**Table 2: Accuracy Summary for Both Applications**

<table>
<thead>
<tr>
<th></th>
<th>Traffic Analysis</th>
<th>MALT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strawman</td>
<td>SQL</td>
</tr>
<tr>
<td>GPT-4</td>
<td>0.29/0.50/0.38</td>
<td>0.88/0.11/0.56</td>
</tr>
<tr>
<td>GPT-3</td>
<td>0.25/0.13/0.25</td>
<td>0.46/0.11/0.33</td>
</tr>
<tr>
<td>text-davinci-003</td>
<td>0.21/0.29/0.29</td>
<td>0.58/0.11/0.22</td>
</tr>
<tr>
<td>Google Bard</td>
<td>0.25/0.21/0.25</td>
<td>0.59/0.11/0.33</td>
</tr>
</tbody>
</table>

**Table 3: Breakdown for Traffic Analysis**

<table>
<thead>
<tr>
<th></th>
<th>Traffic Analysis</th>
<th>MALT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strawman</td>
<td>SQL</td>
</tr>
<tr>
<td>GPT-4</td>
<td>0.50/0.38/0.00</td>
<td>0.75/0.25/0.00</td>
</tr>
<tr>
<td>GPT-3</td>
<td>0.38/0.13/0.00</td>
<td>0.63/0.25/0.00</td>
</tr>
<tr>
<td>text-davinci-003</td>
<td>0.38/0.00/0.00</td>
<td>0.67/0.33/0.00</td>
</tr>
<tr>
<td>Google Bard</td>
<td>0.50/0.25/0.00</td>
<td>0.50/0.13/0.13</td>
</tr>
</tbody>
</table>

**Table 4: Breakdown for MALT**

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Pandas</th>
<th>NetworkX</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-4</td>
<td>0.35/0.00/0.00</td>
<td>0.67/0.73/0.33</td>
<td>1.0/1.0/0.33</td>
</tr>
<tr>
<td>GPT-3</td>
<td>0.33/0.00/0.00</td>
<td>0.67/0.33/0.00</td>
<td>0.67/0.67/0.00</td>
</tr>
<tr>
<td>text-davinci-003</td>
<td>0.33/0.00/0.00</td>
<td>0.33/0.33/0.00</td>
<td>0.67/0.67/0.33</td>
</tr>
<tr>
<td>Google Bard</td>
<td>0.33/0.00/0.00</td>
<td>0.67/0.33/0.00</td>
<td>0.67/0.33/0.33</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>LLM’s error type (NetworkX)</th>
<th>Traffic Analysis (36)</th>
<th>MALT (17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax error</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Imaginary graph attributes</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Imaginary files/function arguments</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Arguments error</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Operation error</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Wrong calculation logic</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Graphs are not identical</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6: Improvement Cases with Bard on MALT

<table>
<thead>
<tr>
<th></th>
<th>Bard + Pass@1</th>
<th>Bard + Pass@5</th>
<th>Bard + Self-debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetworkX</td>
<td>0.44</td>
<td>1.0</td>
<td>0.67</td>
</tr>
</tbody>
</table>

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 \( \times 4 \)) for network traffic analysis and 17 failures out of 36 tests (9 \( \times 4 \)) 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.

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


