MeshAgent: Enabling Reliable Network Management with Large Language Models

YAJIE ZHOU, University of Maryland, College Park, USA
KEVIN HSIEH, Microsoft Research, USA
SATHIYA KUMARAN MANI, Microsoft Research, USA
SRIKANTH KANDULA, Amazon Web Services, USA
ZAOXING LIU, University of Maryland, College Park, USA

The emergence of large language models (LLMs) offers great promise for building domain-specific agents, but adapting them for network management remains challenging. To understand why, we conduct a case study on network management tasks and find that state-of-the-art specialization techniques rely heavily on extensive, high-quality task-specific data to produce precise solutions. However, real-world network queries are often diverse and unpredictable, making such techniques difficult to scale. Motivated by this gap, we propose MeshAgent[1], a new workflow that improves precision by extracting domain-specific invariants from sample queries and encoding them as constraints. These constraints guide LLM's generation and validation process, narrowing the search space and enabling low-effort adaptation. We evaluate our method across three network management applications and a user study involving industrial network professionals, showing that it complements existing techniques and consistently improves accuracy. We also introduce reliability metrics and demonstrate that our system is more dependable, with the ability to abstain when confidence is low. Overall, our results show that MeshAgent achieves over 95% accuracy, reaching 100% when paired with fine-tuned agents, and improves accuracy by up to 26% compared to baseline methods. The extraction of reusable invariants provides a practical and scalable alternative to traditional LLM specialization, enabling the development of more reliable agents for real-world network management.

CCS Concepts: • **Networks** → **Network management**; *Cloud computing*; **Network manageability**; **Topology analysis and generation**; • **Software and its engineering** → *Software verification and validation*; • **Computing methodologies** → *Reasoning about belief and knowledge.*

Additional Key Words and Phrases: Large Language Models (LLMs) for Networking; Agentic System Reliability.

ACM Reference Format:
Yajie Zhou, Kevin Hsieh, Sathiya Kumaran Mani, Srikanth Kandula, and Zaoxing Liu. 2025. MeshAgent: Enabling Reliable Network Management with Large Language Models. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 3, Article 52 (December 2025), 36 pages. https://doi.org/10.1145/3771567

1 Introduction

Large language models (LLMs) have recently sparked interest in automating network management tasks using natural language interfaces [28, 43, 45, 72]. Analogous to intent-based networking (IBN) [30, 35, 57, 59, 62], LLM agents could improve productivity by allowing operators to specify

[1]We open source our dataset and code at https://github.com/Froot-NetSys/MeshAgent

Authors' Contact Information: Yajie Zhou, University of Maryland, College Park, USA, leszhou@umd.edu; Kevin Hsieh, Microsoft Research, USA, Kevin.Hsieh@microsoft.com; Sathiya Kumaran Mani, Microsoft Research, USA, sathiya.mani@microsoft.com; Srikanth Kandula, Amazon Web Services, USA, kandula@gmail.com; Zaoxing Liu, University of Maryland, College Park, USA, zaoxing@umd.edu.

ACM 2476-1249/2025/12-ART52
https://doi.org/10.1145/3771567

high-level goals in natural language. Figure 1 shows an example of using natural language to manage datacenter networks. Despite the potential, their real-world adoption remains limited [26], especially in networking and systems [84]. The core reason is not just accuracy, but the lack of *reliable and adaptable* LLM agents that can handle complex, safety-critical networking tasks like configuration generation [72], fault diagnosis [12, 60], and resource planning [45].

A key barrier to building such agents is the need to specialize LLMs with task-specific knowledge, including network designs, topologies, hardware configurations, and telemetry systems. Most existing specialization techniques, such as retrieval-augmented generation (RAG) [19, 56], prompt engineering [55], and fine-tuning with LoRA [27, 76], require large volumes of curated input-output examples. Unfortunately, in networking, such data is often proprietary, difficult to collect, and rarely generalizes to the diversity of real-world queries. This raises a critical open question: ***How much task-specific data is truly needed to build deployable LLM agents for networking?***

To investigate this, we conduct a case study using current LLM specialization techniques, along with a user study involving network professionals focused on traffic analysis. Two consistent insights emerge from our study results: (1) the effectiveness of an LLM agent depends greatly on how closely its prompts or fine-tuning examples match real-world queries, and (2) user queries are highly diverse and difficult to enumerate in advance. Although creating input-output examples by experts improves performance, our study finds that this process is both error-prone and time-consuming. Even experienced network professionals often require hours to craft accurate solutions for their own queries. These findings reveal a fundamental bottleneck: expert-driven data curation does not scale to the diversity of real-world queries, limiting the practicality of current LLM specialization methods for network management.

Another significant challenge in adopting LLM agents for networking tasks is the reliability required for operations such as planning, monitoring, configuration, and troubleshooting. Unlike general question-answering, errors in these domains can lead to serious consequences, including service outages [65] or critical security vulnerabilities. For example, a recent incident involving a simple misconfiguration caused a major network outage across Facebook services, impacting 2.9 billion users for over six hours [18]. For these high-stakes environments, it is essential for LLM agents to recognize their limitations and abstain from unsafe actions when confidence is low.

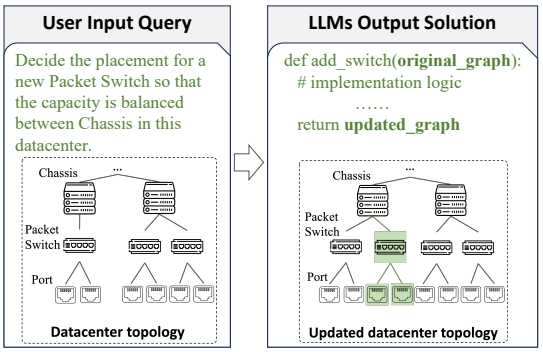


Fig. 1. Example input and output for a datacenter capacity planning query.

In this paper, we introduce MeshAgent, a new framework for building specialized and reliable LLM agents for network management tasks. We focus on graph-structure applications such as capacity planning [50], traffic analysis [23, 29, 85], and cloud resource configuration [48]. In these applications, the system translates a user's goals into specific commands (using a domain-specific language, or DSL) to analyze and manipulate a network graph. While user queries are diverse, we observe that they often share consistent structural invariants. MeshAgent uses these invariants to guide DSL generation, improve correctness, and indicate the reliability of the solution. Our contributions are as follows:

- **Constraint Guided Optimization for Reliable Code Generation.** We introduce a method to encode application-specific invariants (e.g., safety conditions, dependency rules) as *constraints*.

These constraints are applied both during generation (to guide the LLM via the prompt) and after generation (to validate the LLM's output), ensuring the results meet application-specific requirements. If a constraint is violated, the system enters an error reduction loop that iteratively refines the output. MeshAgent also introduces an abstention mechanism, enabling the agent to withhold output when its confidence is low. Confidence scores are computed dynamically using task-specific signals such as output consistency and error frequency, and low-confidence queries trigger a semi-automated review process to update the constraints.

- **Semi-automated Constraint Creation via Failure Driven Learning.** To reduce adaptation cost, MeshAgent provides a constraint synthesis pipeline. It extracts structural constraints (e.g., node and edge types, graph schemas) from the network topology and derives higher-level behavioral constraints by analyzing failures in a small set of sample and abstained queries. Constraints are stored in a database with auto-validation support for continuous refinement. New constraints are compared against existing ones using similarity between embeddings. If similarity falls below a threshold, the new entry is added directly; otherwise, it is reconciled with similar entries to avoid duplication. This enables generalization across diverse networking tasks with minimal expert review.
- **Evaluation on Diverse Benchmark Queries and Real-world Use Case.** We evaluate MeshAgent on three network management applications using a benchmark of 240 curated queries across five agent types and three state-of-the-art LLMs (GPT-4o, Gemini-2, and DeepSeek-v3). MeshAgent consistently improves both reliability and generalization: it achieves 98% accuracy (excluding abstention) and 91% overall using only 14 constraints, compared to 85% and 74% without constraints, respectively. In a real-world user study with 200 open ended queries collected from industry professionals, MeshAgent generalizes to unseen tasks with 96% accuracy using constraints built from just 15 seed input/output examples. We release code, data, and benchmarks to support future research on building and evaluating LLM agents in the networking domain.

2 Motivation

Large language models (LLMs) [1, 7, 13, 52, 68, 70] offer a compelling opportunity to bring natural language interfaces to network management. These models excel at understanding human intent and have achieved remarkable success across diverse domains [37, 66, 71]. Their application in networking is gaining momentum, with early efforts exploring tasks such as configuration generation [3, 72], root cause analysis [12, 25, 60], and broader operator workflows [31, 61, 77]. Despite this promise, LLM-based agents often struggle in real world deployments [12, 25, 60, 72]. Two challenges stand out: accuracy and adaptability. First, even the most advanced models are prone to hallucinations [46] and reasoning errors [8, 14], which can lead to critical misdiagnoses or unsafe network configurations. Second, adapting these agents is a major obstacle because the immense diversity of user objectives, network topologies, and organizational requirements makes it unclear how much task-specific data is sufficient.

To better understand the challenges, we conduct a case study of state-of-the-art LLM agents across three network management tasks. This serves as a concrete lens to evaluate their reliability and adaptability in practical settings.

2.1 Graph Analysis and Manipulation in Network Management

Many network management and control tasks can be framed as *graph analysis and manipulation* over network topologies or communication graphs. In this work, we focus on three representative applications. Table 1 shows example queries from each task.

Apps	Query details
TA	Calculate the byte weight of edges incident per node, cluster into 5 groups using k-means, and color the nodes by cluster.
	How many unique nodes have edges to nodes with label `app:prod` and don't contain the label `app:prod`?
	What is the average byte weight and connection weight of edges incident on nodes with labels `app:user`?
MALT	Optimize topology by identifying removable `PACKET_SWITCH` nodes that won't affect `CONTROL_DOMAIN` connectivity.
	Determine optimal placement of new `PACKET_SWITCH ju1.a1.m1.s2c9` with 5 `PORT`s to balance `AGG_BLOCK` capacity.
	Remove packet switch `ju1.a1.m1.s2c4` from `Chassis` c4. How to balance the capacity between `Chassis`?
CRG	Cut the graph into two parts such that the number of virtual networks nodes between the cuts is the same.
	For all network security groups nodes with name `AllowVnetInBound`, list all ports and rank them based on priority.
	Identify all paths from the network interfaces nodes to virtual networks node with name `Subnet-1`.

Table 1. Representative query examples per application. See Appendix A, Table 7 for more.

- *Traffic Analysis (TA).* 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) [29] or communication graphs [20], in which nodes represent network components such as routers, switches, or devices, and edges symbolize connections or paths between components. 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 current state of the network for performance optimization [29], traffic classification [67], and anomaly detection [33], and (2) manipulating the nodes and edges to simulate the impact of their actions on network performance and reliability [34].
- *Network Lifecycle Management (MALT).* 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 [50]. 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.
- *Cloud Resource Configuration Analysis (CRG).* Cloud resource management involves providing efficient resource exploration and allocation across the cloud. A common approach is to use graph representations and DSLs [48] to facilitate querying of resources and cloud subscription information. Network operators rely on complex query operations (e.g., filtering, grouping, and sorting by resource properties) to manage cloud environments and to assess the effect of applying policies. For instance, users can identify the ports that allow inbound traffic with a specific network security group and rank the network policy by priority.

2.2 Case Study: Specializing LLMs for Network Management

To evaluate state-of-the-art LLM specialization techniques for network management tasks, we collect 30 diverse queries per application from public documents and reports [44, 47, 50], using the applications described in Section 2.1. We use LLM-based agents to generate Python code for natural language queries, verifying its output against the ground truth to ensure it aligns with the query's intent. The accuracy of the techniques is reported based on all 90 queries as the test set. Each query is run five times to determine the average accuracy against the ground truth for the predominant specialization techniques discussed below.

- **Context Injection and Prompt Engineering.** A common way to specialize LLMs is to provide domain-specific context via Retrieval-Augmented Generation (RAG) [19, 36, 56] and few-shot

		Prompt: CoT+RAG + ReAct	Fine-tuned:CoT+RAG (tune w.10 <q,a>)	Fine-tuned:CoT+RAG (tune w.20 <q,a>)
TA	GPT-o1-mini/4o	0.76	0.55	0.71
	Gemini-2/1.5	0.68	0.60	0.65
MALT	GPT-o1-mini/4o	0.71	0.50	0.69
	Gemini-2/1.5	0.67	0.52	0.63
CRG	GPT-o1-mini/4o	0.69	0.64	0.75
	Gemini-2/1.5	0.68	0.57	0.72

Table 2. Existing LLM specialization approaches show limited effectiveness, with low accuracy and generalization on a set of 90 representative networking tasks.

prompting [55]. In RAG, the system retrieves relevant information (e.g., documents, specifications) and prepends it to the prompt. Few-shot prompting provides a small set of query-response examples, sometimes with comments or reasoning steps, to guide the LLM's output. In networking, few-shot examples typically pair user queries with configurations or operational procedures. Furthermore, recent LLM-based code generation introduce more sophisticated prompting methods to improve reasoning, such as ReAct [82] and feedback-driven refinement [9, 11, 63]. These approaches aim to improve output quality by encouraging step-by-step reasoning, incorporating environment feedback, or selecting among multiple candidate solutions.

Evaluation. We test Gemini-2 [69] and GPT-o1-mini [51] using prompting techniques including chain-of-thought (CoT) [75], ReAct [82], and RAG via LlamaIndex [42]. As shown in Table 2, these agents achieve only 67% to 76% accuracy across applications. We find that complex, multi-step requirements in networking queries pose major difficulties. For example, even with ReAct reasoning, GPT-4o places a new packet switch in parallel with a chassis, instead of following the required hierarchy where it should be added as a child node of the chassis. Additionally, it ignores the required capacity attribute on the Port of the new packet switch. These results indicate that supplying more context alone does not ensure reliable execution, especially for network tasks with structural dependencies and latent constraints.

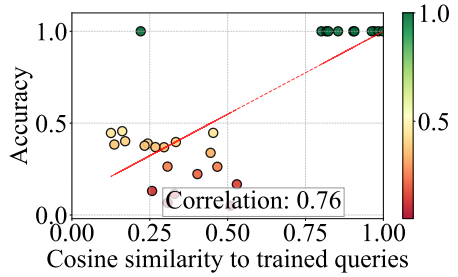
> ***Insight 1:*** *More context is NOT always better. Injecting network domain knowledge via prompting can degrade LLM performance when structural and latent dependencies are not explicitly modeled. Even advanced prompting fails on networking queries that require precise multi-step reasoning.*

- **Model Fine-Tuning.** Another strategy is to fine-tune LLMs on domain-specific data by adapting model weights using example input-output (query-answer) pairs [24, 38, 74]. However, this approach faces generalization challenges similar to traditional machine learning: models often overfit to training patterns and perform poorly on unseen queries. This places a heavy burden on network experts to curate broad and diverse training data, which is time-consuming and difficult to scale. Some recent methods apply reinforcement learning (RL) after fine-tuning to improve robustness [9, 32, 54], but these approaches rely on designing reward signals to provide consistent feedback, which are difficult to obtain in complex network systems.

  Evaluation. To assess this, we fine-tune two LLMs (GPT-4o-mini and Gemini-1.5), combining them with prompting techniques for complementarity. Each model is fine-tuned on 10 query-answer pairs for each application. While both models correctly answer all 10 training queries, they frequently fail on unseen ones, often generating code that mimics the structure of training solutions even when logically incorrect. For example, after fine-tuning Gemini-1.5 on a query

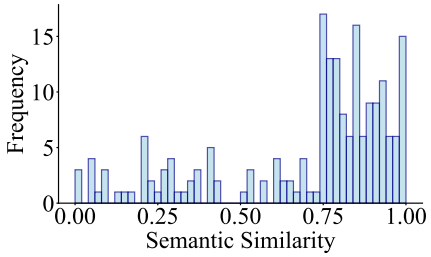
Sample query used for fine-tuning	• `Add a new packet switch with 5 ports, balance the current capacity on all chassis.`
Similar testing query example	• `Add a new packet switch with 10 ports, balance the current capacity on all aggregation blocks.`
Dissimilar testing query example	• `Optimize the network by identifying packet switches that can be removed without affecting the connectivity between all control domain.`

(a) Green highlights denote queries with high semantic similarity.

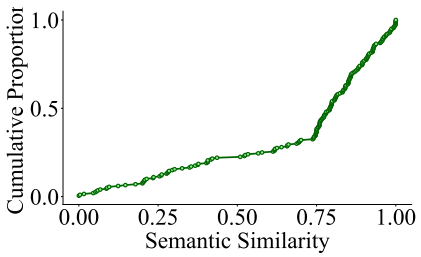


(b) Higher semantic similarity to training data leads to better testing accuracy.

Fig. 2. Fine-tuned models perform poorly on dissimilar queries in the MALT application, revealing limited generalization beyond training examples.



(a) Histogram of semantic similarity scores



(b) CDF of semantic similarity scores

Fig. 3. Semantic similarity between 200 user-submitted and 30 pre-sampled traffic analysis queries shows a long tail of dissimilar cases, highlighting the unpredictability of real-world tasks.

such as "Add a new packet switch so that the capacity is balanced among chassis," along with its corresponding solution, the model learns to apply capacity balancing as a default behavior. As a result, when queried with any request to add a new node to the datacenter, it attempts to balance capacity among chassis at the final step, even in cases where this action is unnecessary or incorrect. (Figure 2a compares a similar testing query with a dissimilar query). As shown in Table 2, this setup performs even worse than prompting alone. We further expand the training set to 20 queries and observe the same overfitting behavior. We also compute the cosine similarity between query embeddings and find a strong correlation between similarity to training examples and test accuracy (Figure 2b).

> ***Insight 2:*** *Fine-tuned LLM agents tend to overfit to training queries and fail to generalize. Without diverse, task-specific data, fine-tuning degrades reliability and mimics solution patterns without understanding the logic.*

2.3 User Study with Network Professionals

Our case study above highlights that the effectiveness of LLM agents strongly depends on the similarity between task-specific examples and real-world queries. However, real users often pose diverse and unpredictable questions that are difficult for domain experts to anticipate. To further investigate this, we conduct a user study at a major cloud provider focused on traffic analysis (§6).

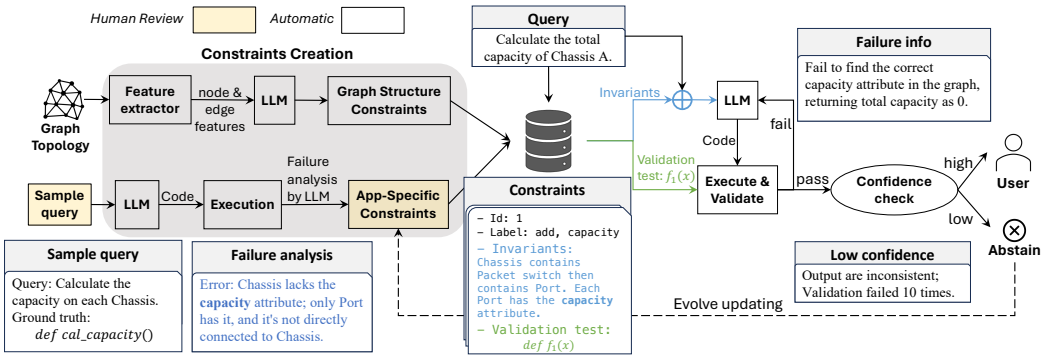


Fig. 4. Instead of relying on prompting or fine-tuning with numerous examples, our framework specializes LLM agents using reusable constraints. This approach offers greater generalization, improved reliability, and reduced human effort for real-world network management tasks.

Network engineers submit 200 open-ended queries. While some resemble our benchmark queries (e.g., calculating total bytes, clustering nodes), many exhibit substantial variation.

Figure 3 shows the semantic similarity distribution across these queries. Over 70% shows high sematic similarity, but a long tail of highly diverse queries accounts for nearly 25% of the dataset. Notably, participants were asked to write ground-truth solutions for their own queries and log the time required. The average answer creation time per query exceeds 1.5 hours. This reinforces a key insight: manually curating representative examples is not only error-prone and labor-intensive, but fundamentally unscalable. The core challenge is thus to go beyond surface-level examples and enable principled generalization. Our user study further underscores the potential need for *structured, reusable knowledge representations* that can help LLMs generalize across the long tail of real-world tasks.

> ***Insight 3:*** *Queries collected from real-world network professionals are diverse. LLM agents relying on pre-defined examples cannot scale to the unpredictable, long-tail nature of production queries.*

3 Design

Overview. We introduce MeshAgent, an end-to-end workflow for specializing LLM agents to network management tasks by extracting *reusable knowledge representations*, which we call *constraints*. Unlike data input-output examples that are unbounded and hard to enumerate, constraints offer a more compact and generalizable way to guide LLM behavior. Intuitively, there may be many different queries, but only a limited number of underlying rules and invariants. However, it remains unclear how many constraints are needed, how reliable they are, and what is the best way to extract them when adapting LLMs to a new application. MeshAgent addresses these questions with a practical framework designed to (1) reduce adaptation overhead, and (2) ensure high reliability by abstaining when uncertain. It is also designed to be dynamic and continually improve through constraint refinement (Figure 4).

(1) Adaptation-time. Traditional LLM specialization techniques, such as RAG and fine-tuning, rely heavily on manual data collection and expert curation. These processes are difficult to scale and lack standardization. MeshAgent addresses these challenges by introducing *constraints* as a unifying abstraction to reduce human intervention. Each constraint consists of an *invariant*, which is a natural language rule that define fundamental properties that must always hold, and a

```
"id": 1,
"label": node, hierarchy,
"invariant": Nodes must follow the hierarchy: CHASSIS contains PACKET_SWITCH, JUPITER contains
SUPERBLOCK, SUPERBLOCK contains AGG_BLOCK, AGG_BLOCK contains PACKET_SWITCH, PACKET_SWITCH contains PORT,
"validation_test":
    def verify_node_hierarchy (self):
            hierarchy = {
                "JUPITER": ["SUPERBLOCK"],
                "SUPERBLOCK": ["AGG_BLOCK"],
                "AGG_BLOCK": ["PACKET_SWITCH"],
                "CHASSIS": ["PACKET_SWITCH"],
                "PACKET_SWITCH": ["PORT"],
            }

            for edge in self.graph.edges(data=True):
                if 'CONTAINS' in edge[2]['type']:
                    source_node_types = self.graph.nodes[edge[0]]['type']
                    target_node_types = self.graph.nodes[edge[1]]['type']
                    for source_type in source_node_types:
                        if source_type in hierarchy and any(target_type in hierarchy[source_type] for
                        target_type in target_node_types):
                            return True, ""

            raise Exception("verify_node_hierarchy failed at edge: " + str(edge))
```

Fig. 5. Each `Constraint` includes a label describing the keyword, a natural language invariant defining the application-specific condition, and a validation function to check if the generated result satisfies it.

corresponding *validation test* that verifies whether LLM outputs satisfy the rule. MeshAgent prepares these constraints using a hybrid process. First, it automatically extracts structural features from the network graph (such as nodes, edges, and metadata) and uses LLMs to generate corresponding invariants and validation code. Second, MeshAgent analyzes LLM failures on abstained queries, identifies recurring error patterns, and infers updated constraints. Engineers only need to review, edit and approve the constraint suggestions, which significantly reduces manual effort.

② **Run-time.** After constraints are built, MeshAgent integrates with the LLM agent by retrieving relevant entries from the constraint database. The invariants are included in the prompt, and the LLM-generated output is executed in a sandbox environment before being checked against validation tests. If it fails, MeshAgent enters an error-reduction loop: the original query, constraints, and error context are provided to the LLM to generate a corrected response. Only validated outputs are returned to the user. To further improve reliability, MeshAgent includes a *heuristic abstention mechanism* that suppresses low-confidence answers. Confidence scores are computed using task-specific signals, such as output consistency and error rates. If confidence remains low after refinement, the query is routed back to the adaptation-time workflow for constraint update. This design reduces the need for manual debugging and helps prevent the deployment of incorrect responses.

3.1 Building Constraints for Specific Network Applications

To improve LLM performance on domain-specific queries, prior work [19] uses Retrieval-Augmented Generation (RAG), where an external database provides contextual information for prompt construction. This typically includes past user queries with answers (few-shot learning) or domain handbooks (e.g., tool instructions or library guides). However, applying this approach directly to networking tasks introduces two key challenges: (1) Exhaustively enumerating user queries is impractical and risks introducing low-quality examples; (2) Injecting excessive or irrelevant context can mislead the LLM. For instance, in datacenter capacity planning, if attributes like ‘switch_loc‘ (location) are unnecessarily included in the prompt, a simple query like “Calculate capacity?” may result in incorrect logic that focuses on location-level aggregation rather than port-level summation.

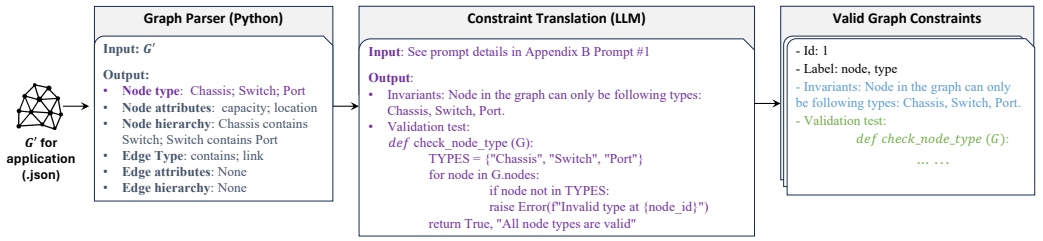


Fig. 6. Graph structure constraints are directly extracted from the graph data and translated into code by an LLM. Their explicit structure makes them easy for LLMs to process accurately.

Constraint Definition. To address these issues, we propose focusing on a small set of critical, reusable invariants instead of injecting verbose domain context. The key insight is that *while user queries are diverse and long-tailed, the correctness conditions governing their answers (i.e., invariants) are limited and stable across tasks.* By guiding the LLM with these invariants, we avoid the overhead of exhaustive enumeration and reduce prompt noise, leading to more accurate and reliable outputs.

We define a structured 'Constraint' format that supports both runtime prompting and post-generation validation (example in Figure 5):

- *Label*: A keyword-based summary describing the constraint's abstraction. It supports fuzzy matching between user queries and relevant constraints.
- *Invariant*: A natural language rule specifying the application-specific condition that must hold (e.g., valid node types, required field formats).
- *Validation test*: A test function that checks whether the generated output satisfies the constraint. This is used in the error-reduction module (§3.2).

Constraint Generalization. At first glance, building *Constraints* for each new application may appear labor-intensive. A natural approach is to either (1) rely on domain experts to manually define them, or (2) use LLMs to generate them automatically. However, both have limitations: Manual creation is time-consuming, error-prone, and bounded by the expert's prior knowledge, which may miss critical cases. Automated generation with LLMs is faster but often unreliable, since LLMs can produce plausible but incorrect or inconsistent constraints. Furthermore, generating accurate constraints often requires giving LLM the full access to the network graph data, which may be restricted in industrial settings due to privacy concerns.

To address these issues, we introduce a semi-automated constraint creation workflow that balances efficiency with reliability. Instead of relying entirely on manual effort or LLMs, we leverage LLMs to only translate structured inputs into constraint formats, while engineers review and refine the results rather than creating them from scratch. We categorize constraints in graph-based network management into two types:

(1) *Graph-Structure Constraints*: These constraints define the attributes and relationships of nodes and edges within network graphs. (i) We first implement a feature extractor in Python. Given a graph as input, it extracts structural features such as all node and edge types, attribute fields, and hierarchical relationships. (ii) Each extracted feature is then sent to an LLM-based constraint translator, which generates a constraint invariant and a corresponding validation test using structured prompting (See full prompt in Appendix B). (iii) The validation test is executed directly on the graph data to verify the correctness and functionality of the generated constraint. Because the input is structured and unambiguous, this automated process produces high-quality constraints without requiring manual review (Figure 6).

(2) *Application-Specific Constraints*: These constraints encode logical rules specific to an application, such as valid actions, dependencies, or invariants that are inexplicit from the graph data. (i) Engineers begin by providing a small set of sample queries and their ground truth answers (we use 10 in our evaluation). (ii) For each sample query, the LLM agent attempts to generate a solution using the existing graph constraints. (iii) The output is executed and compared to the ground truth. If the execution results of the generated solution, under current constraints, is numerically equivalent to the ground truth known by operators, no new constraint is needed. If they differ, the failure information and ground truth are sent to the LLM-based constraint translator to produce a new constraint based on the observed failure (See full prompt in Appendix B). (iv) Newly generated constraints are manually reviewed for correctness before being incorporated into the constraint database. The LLM agent is then re-run on the same query using the updated constraint set. If it passes, the constraint is validated and retained. Although this process involves human-provided sample queries, our evaluation shows it produces more generalizable constraints with significantly less effort compared to traditional fine-tuning or prompt engineering approaches (Figure 4).

Query-specific Constraint Extraction. Once constraints for an application are created, the challenge becomes selecting the most relevant subset for each query. Simply appending all constraints degrades accuracy ("All constraints" in Table 3), as excessive context make LLMs lost in the middle [41]. Instead, we dynamically extract query-specific constraints to form focused prompts.

To retrieve the most relevant constraints, MeshAgent uses a *hybrid search* method that combines keyword and vector-based similarities. This approach is especially effective for network queries, which often involve technical terms and domain-specific synonyms. For instance, "port" and "switch" are semantically related but may not always co-occur. Hybrid search captures such relations by combining two perspectives: exact matches (via keyword search) and semantic proximity (via vector embeddings). For each user query, we compute the hybrid match score using Reciprocal Rank Fusion (RRF) [15], which integrates ranked results from both search methods, where k is a smoothing constant, and $r(c)$ is the rank of constraint c in each individual search:

$$RRFscore(c \in C) = \sum \left[\frac{1}{k + r(c)} \right] \tag{1}$$

MeshAgent applies this fusion process between the query and constraint representations (including their labels) to select those with a similarity score above 0.7 (tunable threshold) for inclusion in the prompt. When queries contain specific terms (e.g., "packet switch"), keyword rankings tend to dominate; whereas in cases with differing surface terms (e.g., "capacity" vs. "bandwidth"), vector similarities play a greater role. MeshAgent empirically adopts RRF as a common and effective fusion method for combining keyword and vector search results. While our focus is not on optimizing the fusion strategy itself, more advanced algorithms could further improve constraint retrieval performance.

Dynamic Constraint Evolution. Network management queries often follow predictable patterns but also include a long-tail of rare or complex cases (Figure 3b). To handle these edge cases, MeshAgent dynamically evolves its constraint set based on system interactions. When the LLM abstains from answering a query due to low confidence, the associated failure log is used to identify gaps in the constraint

	All Constraints	Keyword Search	Vector Search	Hybrid Search
Precision	0.56	0.81	0.90	0.91
Recall	1.00	0.89	0.88	0.94
F1	0.72	0.85	0.89	0.93

Table 3. Hybrid search consistently outperforms keyword and vector methods, achieving higher constraint matching accuracy across diverse network queries.

database. If a network engineer requests, "Cluster all nodes in the communication graph into five groups," and the LLM abstains, the logs may reveal that the agent attempted various clustering algorithms, but some were incompatible with the structure or semantics of communication graphs. In such cases, engineers can refine the constraint set by adding new invariants to exclude inappropriate algorithm choices, enabling the agent to avoid similar failures in future requests.

To maintain consistency and avoid redundancy, each new invariant is compared against existing entries using cosine similarity between their embeddings. If the similarity falls below a set threshold, the invariant is added directly; otherwise, it is reconciled with overlapping entries. This keeps the constraint database concise and relevant. Validation tests are also revised as network structures or safety policies evolve. While human oversight may still be required, this semi-automated pipeline substantially reduces manual workload and improves adaptability. We evaluate this mechanism using abstained queries collected from the real-world user study in Section 6.

3.2 Contextual Error Reduction

Even with constraint guidance to narrow the LLM's search space, models still struggle with complex network management queries. For example, a query like "Cluster the graph nodes based on bytes transmitted" requires multi-step reasoning and execution, which LLMs often fail to handle in a single pass. To address this, we incorporate the existing Chain-of-Thought (CoT) principle [75] to explicitly decompose such queries into sub-steps, reducing complexity per step.

Once decomposed, the next challenge is to minimize errors before executing these steps in a real environment. Prior approaches use reinforcement learning [11] or generate unit tests to select valid outputs [10, 64, 83], but as the case study (Table 2) shows, these methods are ineffective for network management. LLMs often produce syntactically correct code that violates domain-specific constraints, or fail to generate valid test cases due to inconsistent formats across applications.

To address this, we introduce a bi-level error detection and reduction mechanism with contextual support at each step of execution (Algorithm 1):

- *Execution error check*: Detects failures in the sandbox environment caused by hallucinated attributes, invalid API calls, or syntax errors.
- *Constraint error check*: Uses validation functions from the constraint database to verify that the generated output satisfies application-specific rules.

If an error is detected, the system feeds the query, execution step, relevant constraint, and error trace back to the LLM to regenerate a corrected version. If the model continues to fail after N iterations, the system generates a structured summary of the failure, including error types, affected steps, and violated constraints. This summary enables a network engineer to intervene with minimal effort.

This error-reducer is especially effective for abstract or optimization queries where a one-shot response is unlikely to be correct. For instance, the prompt "Optimize the current network topology by removing redundant nodes" typically requires multiple iterations. As shown in Fig. 14, both execution and constraint errors become more frequent with increasing query complexity, requiring more correction rounds.

Pipelined Execution to Reduce Latency. The sequential nature of multi-step debugging introduces latency. To mitigate this, we pipeline error checks across steps. As soon as code for one step is generated, the next step begins while the current one is being validated. Final output is returned only when all steps pass. If an error is detected, downstream steps are paused until the issue is resolved. In practice, most errors occur in later steps, allowing early pipelining to reduce latency. This technique yields a 76% latency improvement on CoT method (Fig. 16b).

3.3 Enhancing Agent Reliability

Even with the use of constraints and contextual error reduction, we cannot fully guarantee that an LLM will always produce correct outputs. This limitation poses a critical challenge in network management tasks, where reliability is essential. LLMs generate text by predicting the next token rather than through true semantic understanding [46, 58, 79]. As a result, they may produce confident-sounding but incorrect responses—an issue that becomes especially risky in networking for two key reasons. First, incorrect outputs can directly compromise system integrity. For example, in response to the query "Split the graph into two parts such that the number of virtual networks is the same," an LLM may mishandle the configuration of security groups, inadvertently assigning overly permissive access rules and exposing critical cloud resources. Second, relying on human engineers to manually verify and debug LLM responses is impractical. Network queries often involve platform-specific syntax and large outputs, making error detection labor-intensive and inconsistent—ultimately hindering the usability of LLMs for high-stakes operational tasks.

To address this, we define reliability as the agent's ability to either produce a correct answer or abstain when uncertainty is high. Many prior methods estimate confidence using model-generated scores. One approach asks the LLM to report its own confidence [78], but these scores are often biased. Another uses log probability or perplexity as a proxy, but this is known to correlate poorly with correctness [6].

Heuristic-Based Confidence Scoring. We propose a heuristic confidence scoring mechanism tailored to our constraint-guided workflow. If an output fails error-checking after debugging, the agent abstains with zero confidence. Otherwise, the confidence score $S_{\text{confidence}}$ is computed based on two key factors: (1) the semantic consistency of the LLM's output across iterations (denoted as C_{semantic}), and (2) the number of iterations needed in the debugging loop to reach a valid response (denoted as I_{debug}). This formulation prioritizes outputs that require fewer corrections and exhibit higher consistency:

$$S_{\text{confidence}} = \begin{cases} 0, & \text{if } E_{\text{check}} = 0 \\ w \cdot C_{\text{semantic}} + (1-w) \cdot \left(1 - \frac{I_{\text{debug}}}{N}\right), & \text{if } E_{\text{check}} = 1 \end{cases} \tag{2}$$

Here, E_{check} indicates whether the output passes the error-checking stage. N is the maximum number of allowed debugging iterations. The weight $w \in [0, 1]$ controls the relative importance of semantic consistency versus the number of correction iterations. In our evaluation, we use $N = 5$ and $w = 0.5$, which can be tuned based on application-specific reliability requirements.

Mitigating Uncertainty with Abstention. Unlike prior methods that rely on internal estimates from LLM, our approach bases confidence on observable external signals tailored to the specific network application. This design ensures the system avoids high-confidence errors and abstains when needed. We evaluate this in graph-based network management scenarios where correctness is critical. As shown in Figure 5, our method significantly improves the agent's reliability by correctly abstaining from uncertain answers and reducing the likelihood of incorrect outputs.

4 Implementation and Benchmark

We implement MeshAgent in Python with approximately 1,200 lines of code. For each application, we build a graph manipulation simulator to execute and validate LLM-generated solutions. To systematically assess LLM agent performance in network management, we introduce a comprehensive benchmark including three components:

- **Ground Truth Selector.** This component defines a "ground truth" for each user query, representing the expected functionality or correct output. Creating and validating these ground truths

can be labor-intensive, requiring human expert review (e.g., with an average of 1.5 hours per input-output pair based on our experience). The validated results reference dictionary used to assess the accuracy of LLM-generated outputs.

- **Results Evaluator.** This component executes LLM agent generated code in an isolated environment using real network data. It compares the outputs, such as modified graph structures or extracted information, against the predefined ground truth. Outputs matching the ground truth are classified as correct, while mismatches are flagged for further analysis of errors.
- **Results Logger.** This component records all evaluation details, including LLM-generated code, ground truth answers, and comparison results. Additionally, it logs execution errors and discrepancies, enabling iterative refinement of the benchmark by expanding its constraints database to support more robust testing scenarios.

5 Evaluation

To evaluate the effectiveness of MeshAgent in building domain-specific LLM agents, we conduct a comprehensive study across six existing specialization techniques as baselines. Our evaluation is guided by the following research questions:

- **Q1**: To what extent can LLM agents deliver accurate and reliable results across a wide range of network management tasks?
- **Q2**: Can a small set of structured constraints effectively adapt LLM agents to new applications? How much effort does their creation require?
- **Q3**: What are the dominant failure modes of LLM agents in network management?
- **Q4**: How do LLM agents perform in real-world usage by network engineers, both in terms of accuracy and user experience?

5.1 Experimental Setup

LLMs. We study two leading proprietary LLMs: GPT-4o [53] and Gemini-2 [69], as well as one latest open-source model: DeepSeek-V3 [39]. At the time of this writing, reasoning-focused models such as GPT-o1 and DeepSeek-R1 exhibit prohibitively high latency at inference time. For example, evaluating a single query on GPT-4o typically takes under 30 seconds, while DeepSeek-R1 requires over 30 minutes per query, more than 60 times slower. Such latency becomes a major bottleneck for high volume query evaluation (although we test GPT-o1-mini on smaller set of queries in §2.2). For each query with one LLM agent, it is run five times to reduce variance.

Applications and Queries. We implement a network graph manipulation simulator in Python and evaluate three applications as described in Section 2.1. A total of 240 distinct queries with corresponding ground truth are constructed based on prior research and publicly available examples [21, 44, 47, 50]. Additionally, evaluation using 200 open-ended queries is included in the user study results presented in Section 6.

Baseline Agents. Based on their adoption in prior work, we select representative optimization techniques for domain-specific LLM agents. These methods, which are complementary in nature, serve as strong baselines for accuracy evaluation. Specifically, we define two primary baselines for accuracy comparison: *CoT+Few-shot* and *CoT+Fine-tuned*. In addition, we evaluate several other techniques to provide a more comprehensive analysis:

- *Chain-of-Thought prompting (CoT) [75]* prompts LLMs to decompose questions into reasoning steps. For each application, we use standard CoT prompts that encourage step-by-step reasoning, enhancing output accuracy.

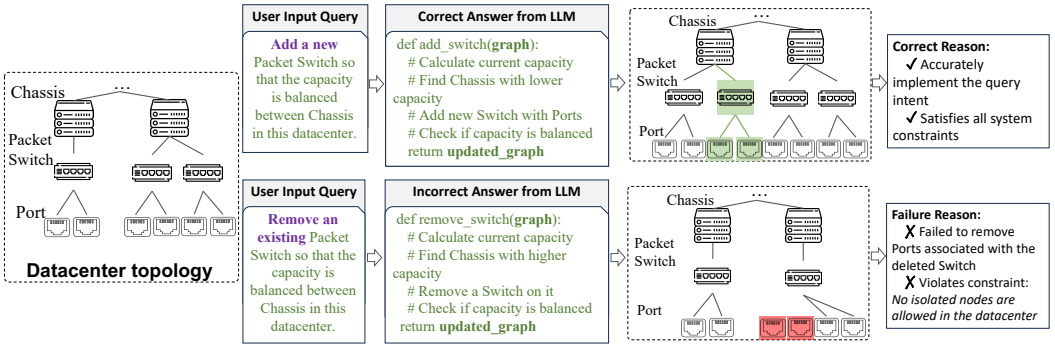


Fig. 7. LLM-generated examples from MALT: one correct output and one failure case.

- *Few-shot Retrieval-Augmented Generation (Few-shot) [42]* integrates agents with domain-specific knowledge. We upload input-output pairs identical to those used in MeshAgent 's adaptation. Few-shot examples guide the model's responses through retrieval-based techniques.
- *Reinforcement learning with ReAct (RL) [82]* combines reasoning with real-time decision-making by adjusting responses based on immediate feedback. We evaluated this method using official prompts tailored to adapt reasoning steps dynamically based on action outcomes.
- *Fine-tuned model (Fine-tuned) [74].* We fine-tune GPT-4o-mini[53] via Azure OpenAI [49] and Gemini-1.5 [69] using the Gemini API [22] on the same input/output pairs as MeshAgent's adaptation. Platform-specific auto-optimization refine training parameters for improved performance.
- *Language Agent Tree Search (LATS) [83]* integrates Monte Carlo Tree Search with model-based reinforcement learning to improve decision-making in code generation. By leveraging external feedback, it surpasses prompting methods like Tree-of-Thought[81] in reasoning accuracy.

5.2 Code Accuracy and Reliability

Result Examples. To clarify how accuracy is measured in our evaluation, we present representative examples of both successful and failed LLM agent outputs. Figure 7 shows a correct response and an incorrect one, along with their respective execution results. In the successful case, the LLM correctly interprets the intent to add a new switch: it inserts the switch node, attaches new ports to it, and assigns the appropriate capacity attribute to maintain balanced total capacity across all Chassis nodes. In contrast, the failed example involves a query requesting the removal of a packet switch to rebalance capacity. The LLM-generated code removes only the switch node but neglects to remove its connected ports, resulting in orphaned nodes. This violates a key structural constraint: "No isolated nodes are allowed in the datacenter."

From our user study observations, we find that users (e.g., operators under time constraints) do not always specify all implicit requirements in their queries (e.g., "remove the switch" should implicitly include removing its connected ports). This highlights the importance of leveraging system constraints both to enrich the prompt fed to the LLM and to validate the output post-generation. Further analysis of failure types and statistics is provided in §5.4.

Reliable Accuracy. This metric measures correctness for queries that receive a response from the LLM agent, excluding those where the agent abstains. It reflects performance conditional on agent confidence. To ensure fairness, we apply the same confidence scoring mechanism across all methods, including the two primary baselines.

		CoT+Few-shot	with MeshAgent	CoT+Fine-tuned	with MeshAgent
TA	GPT-4o	0.821	0.987 (↑ 0.17)	0.829	0.991 (↑ 0.16)
	Gemini-2/1.5	0.805	0.956 (↑ 0.15)	0.809	0.967 (↑ 0.16)
	DeepSeek-v3	0.835	0.962 (↑ 0.13)	-	-
MALT	GPT-4o	0.842	0.986 (↑ 0.14)	0.910	0.986 (↑ 0.08)
	Gemini-2/1.5	0.898	0.958 (↑ 0.06)	0.923	0.979 (↑ 0.06)
	DeepSeek-v3	0.810	0.964 (↑ 0.15)	-	-
CRG	GPT-4o	0.742	1.000 (↑ 0.26)	0.835	1.000 (↑ 0.17)
	Gemini-2/1.5	0.793	0.957 (↑ 0.16)	0.832	0.981 (↑ 0.15)
	DeepSeek-v3	0.803	0.989 (↑ 0.19)	-	-

Table 4. With an average of just 12 sample queries per application, MeshAgent boosts correctness on answered queries by enabling abstention, and consistently enhances all agent types across models and applications.

Results in Table 4 highlight two key takeaways. First, MeshAgent consistently improves accuracy across all models, agents, and applications. This broad improvement demonstrates the effectiveness and generality of MeshAgent 's workflow in building specific LLM agents for networking tasks. Second, when applied to CoT+Fine-tuned, MeshAgent achieves higher reliable accuracy than with CoT+Few-shot. This stems from how confidence scores reward output consistency, and fine-tuned models tend to generate more stable code due to their exposure to structured training data. These findings emphasize that, when powered by MeshAgent, fine-tuning holds strong potential for improving LLM reliability in complex, domain-specific scenarios.

Total Accuracy. Unlike reliable accuracy, which measures correctness only when the agent chooses to respond, total accuracy evaluates correctness across all queries regardless of abstentions. This provides a direct assessment of raw end-to-end performance. We compute average accuracy and variance across three applications and LLMs to evaluate different baseline combinations.

Figure 8 shows that MeshAgent consistently improves accuracy across all agents, aligning with trends in Table 4. However, not all agent types are equally suited for network applications. First, incorporating RL does not always enhance performance, likely due to the complexity of network tasks. ReAct, the RL-based approach, relies on the model to autonomously execute reasoning steps. While effective in structured environments like text-based games [82], its effectiveness declines in network management due to the lack of reliable execution feedback and the inability to guarantee correct reward signals for complex queries. Second, LATS, the code generation framework, does not outperform simpler methods. This indicates that in network management, mechanism sophistication alone does not guarantee better accuracy. Instead, effectiveness depends on selecting approaches that align with network tasks in terms of execution reliability, and adaptability to complex queries.

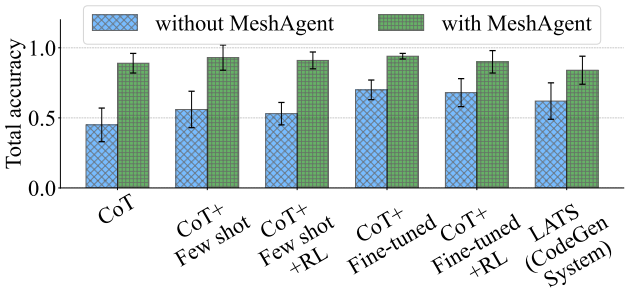


Fig. 8. Average total accuracy across three apps.

		Abstain-Accu	Abstain-Precision	Abstain-Recall	Abstain-Rate
TA	MeshAgent	**0.99**	**0.83**	**1.00**	**0.08**
	LLM-gen	0.66	0.23	0.78	0.40
	Perplexity	0.78	0.30	0.67	0.25
MALT	MeshAgent	**0.97**	**0.75**	**1.00**	**0.10**
	LLM-gen	0.65	0.21	0.77	0.41
	Perplexity	0.77	0.26	0.56	0.24
CRG	MeshAgent	**0.99**	**0.87**	**1.00**	**0.10**
	LLM-gen	0.69	0.23	0.78	0.37
	Perplexity	0.71	0.25	0.78	0.35

Table 5. MeshAgent's confidence score achieves higher abstention accuracy than existing confidence score generation methods.

Abstention Accuracy. Inspired by [17], we evaluate the agent's abstention performance using four metrics as follow. See Figure 9 for the meaning of a, b, c, d.

- $Abstention\ Accuracy = \frac{a+d}{a+b+c+d}$
- $Abstention\ Precision = \frac{d}{c+d}$
- $Abstention\ Recall = \frac{d}{b+d}$
- $Abstention\ Rate = \frac{c+d}{a+b+c+d}$

	Correct Results	Wrong Results
Output	a	b
Abstain	c	d

Fig. 9. Abstention metric.

Abstain-Accu measures the overall correctness of both answering and abstention decisions. Abstain-Precision quantifies how often abstention successfully avoided incorrect answers. Abstain-Recall represents the proportion of incorrect responses that were correctly abstained from. Abstain-Rate indicates the overall frequency of abstentions.

We compare MeshAgent's confidence score design with two prior methods: LLM-generated confidence scores [78] and perplexity-based confidence estimation [6]. Table 5 summarizes the abstention performance of the MeshAgent (CoT+Few-shot) agent with GPT-4o across all applications, using a confidence score threshold of 0.7. Among the three abstention methods, MeshAgent demonstrates the most stable performance, achieving higher abstain precision and recall while maintaining a lower abstain rate. An abstain recall of 1.0 ensures the agent abstains only when necessary. However, lower precision indicates occasional over-cautiousness, abstaining despite correct answers due to output inconsistencies, reflecting the trade-off between reliability and unnecessary abstentions. These findings provide valuable insights for network operators selecting the most effective model.

In practice, operators can evaluate different agents on a set of queries and choose the one with the lowest abstain rate while maintaining high recall and precision. While the confidence score threshold can be adjusted for specific applications, exploring optimal threshold tuning is left for future work.

Takeaway for Q1: *MeshAgent consistently boosts solution accuracy and reliability across all agent types, regardless of specialization method. Surprisingly, when paired with MeshAgent, simple baselines like few-shot prompting outperform more complex approaches, highlighting that using explicit constraints matters more than model complexity in network tasks.*

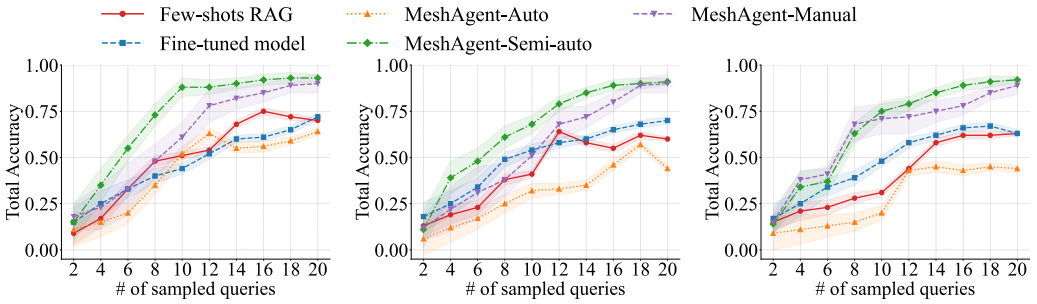


Fig. 10. MeshAgent adapts to applications with fewer input samples compared to baseline agents.

5.3 Constraints Adaptation Efficiency & Effort Analysis

Constraints Adaptation Efficiency. To evaluate the adaptation data efficiency of MeshAgent approach compared to traditional methods, we measure total accuracy on testing query using the same number of sample queries. We also compare three constraint-generation methods in MeshAgent. Figure 10 illustrates how adaptation accuracy evolves with the number of sample queries. While all methods improve with more samples, their convergence rates differ. MeshAgent converges significantly faster, suggesting that underlying invariants within each application remain consistent and can effectively guide LLMs. In contrast, the Few-shot and Fine-tuned approaches show gradual improvement. However, the accuracy of Few-shot declines as more examples are added, likely due to increased noise in the prompt. This highlights a key limitation of traditional external knowledge augmentation, where maintaining concise prompts is critical for accuracy in complex tasks.

Among the constraint-generation methods, MeshAgent (semi-auto, constraints generated by LLMs and then human-reviewed) and MeshAgent (manual, fully handcrafted constraints) yield the most stable and consistent improvements in adaptation accuracy. In contrast, MeshAgent (auto, fully LLM-generated constraints) exhibits lower stability, as the absence of human oversight can lead to erroneous constraints. A single incorrect constraint can degrade performance significantly because it affects multiple test queries. This finding underscores the importance of human oversight in constraint creation, balancing automation with reliability to maximize accuracy and efficiency.

Constraint Number and Quality. The quality of generated constraints is closely tied to the diversity and relevance of sampled queries used during adaptation time. Since there is no universally principled method for generating such queries, we collect real-world examples from publicly available documentation and query sets for each application domain [47, 50]. The most critical factor determining constraint usefulness is their specificity to application-level details. To produce such domain-specific constraints, we employ a few-shot prompting strategy where the LLM is asked to translate observed failure cases into precise, actionable constraints (see Prompt B). Furthermore, application-specific constraints are manually reviewed by human to ensure their correctness.

To evaluate whether LLMs are sensitive to prompt length and constraint ordering, we shuffle the full constraint set three times and plot the average accuracy under different numbers of constraints in App-TA. As shown in Figure 11, contrary to the common assumption that more information improves performance, we find that after a

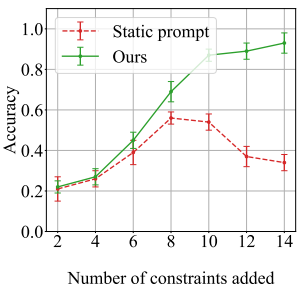
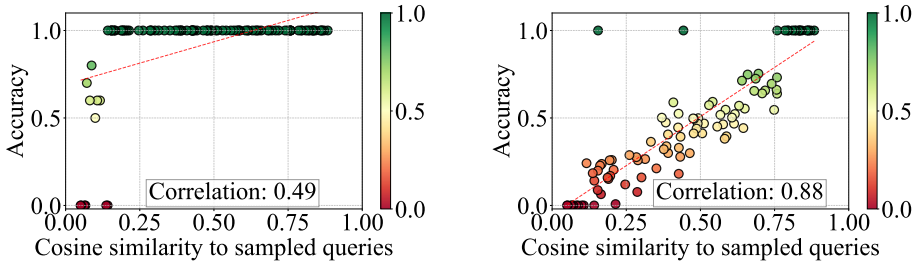


Fig. 11. Constraint # Effect



(a) MeshAgent generalization **(b) Few-shot generalization**

Fig. 12. Constraints enable stronger generalization, even for testing queries with low similarity.

	Avg.# of iters	Avg.time (mins)
Human creation (Manual)	1	20.5
LLM creation (Auto)	1	0.2
MeshAgent (Semi-auto)	2	1.3

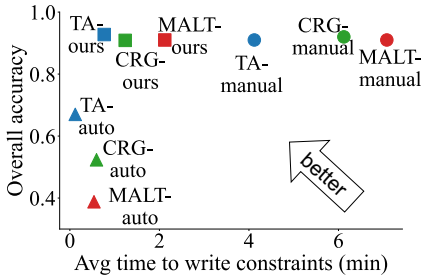


Fig. 13. MeshAgent's constraints creation offers an effective balance between accuracy and effort.

certain threshold, adding more constraints in a static prompt setting *reduces* accuracy, as excessive and irrelevant context distracts the LLM's attention. On the other hand, since MeshAgent uses hybrid-search to dynamically extracts the most related constraints per query, adding more constraints can still improve accuracy. We provide the constraint sets for each application in the Appendix D.

Generalization to Testing Queries. To evaluate MeshAgent's effectiveness across diverse queries, we analyze its accuracy relative to the cosine semantic similarity of testing queries in the MALT application using 14 sampled queries. As shown in Figure 12a, MeshAgent maintains consistently high accuracy across a broad similarity range, leveraging common failure modes to improve generalization. Accuracy only declines at very low similarity levels (around 0.1), which can be mitigated by incorporating new constraints to address observed failures, enhancing adaptability. In contrast, Figure 12b highlights Few-shot's strong dependence on query similarity. While effective for queries closely matching its samples, its accuracy declines sharply below a similarity threshold of 0.78, exhibiting a near-linear drop. This occurs because Few-shot relies on memorizing examples rather than identifying underlying patterns, limiting its scalability and adaptability, particularly with limited data samples. These findings highlight MeshAgent's scalability and generalization capabilities. By capturing failure modes rather than memorizing individual examples, MeshAgent reduces the need for extensive human intervention, ensuring long-term efficiency and adaptability, even for dissimilar testing queries.

Constraint Creation Effort Estimation. We assess the effort required to create constraints in MeshAgent by comparing three methods and measuring the average time taken. While the time measurements are inherently subjective, they offer meaningful insights into efficiency trends. The left side of Figure 13 presents the average constraint creation time and variance across three applications. The right side of Figure 13 examines the trade-off between time spent and resulting query accuracy. MeshAgent (semi-auto) consistently achieves the best balance, significantly reducing

effort while maintaining high accuracy. Both semi-auto and manual approaches show varying time requirements depending on application complexity. For example, the MALT application requires more time due to its intricate graph structures and hierarchical relationships. In such cases, verifying constraints is more involved, often requiring engineers to write and execute validation scripts, as correctness cannot always be determined from raw data alone.

This study highlights the strengths of MeshAgent as an end-to-end framework that streamlines constraint creation while ensuring accuracy across diverse network applications. Our findings also reveal opportunities for further optimization, particularly in handling complex applications with intricate data structures. One direction is developing a specialized LLM agent for constraint verification, potentially integrating formal verification techniques to enhance validation and further reduce human intervention while maintaining constraint correctness. Exploring this optimization presents a valuable opportunity for future research.

Takeaway for Q2: *MeshAgent achieves over 80% accuracy with just 12 sample queries, while Few-shot requires more than 20 queries to reach 60% accuracy. Its constraint creation method also offers the best tradeoff between accuracy and effort. Notably, MeshAgent generalizes well to testing queries, even when they differ significantly from the sample queries, making it well-suited for real-world deployment.*

5.4 Detailed Error Analysis

Failure Patterns and Examples. To understand agent failures, we analyze error types in MeshAgent (CoT+Few-shot) using GPT-4o. Table 6 shows that errors vary by application. In traffic analysis, most failures stem from incorrect aggregation logic in multi-condition queries. For example, in the query "What is the average byte weight and connection weight of edges incident on nodes with labels app:prod and app:test?", the agent may incorrectly include outbound edges when only inbound edges should be considered. It sometimes calculates average across all edges instead of grouping by label, leading to inaccurate results. In MALT, failures arise when the generated code violates structural constraints. For instance, in the query "Remove packet switch 'ju1.a1.m1.s2c4' from Chassis c4. How should capacity be balanced?", the agent fails to redistribute links properly, resulting in disconnected spine-leaf segments or isolated nodes that break the datacenter's routing fabric. Similar issues appear in CRG, where structural constraints are not always maintained. While confidence scoring helps abstain from uncertain responses, analyzing failure patterns can help operators improve the constraint set for long-term reliability.

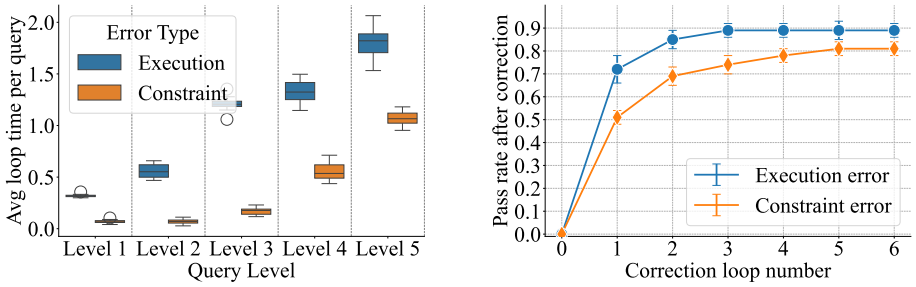
Error Type	TA(5)	MALT(7)	CRG(6)
Operational error	1	2	0
Incorrect logic	4	1	2
Constraints violation	0	4	4

Table 6. Error types of CoT+Few-shot agent with GPT-4o.

Error Reduction Efficiency. MeshAgent incorporates a built-in contextual error reducer to minimize potential errors. We evaluate its effectiveness by categorizing MALT test queries into five complexity levels based on ground truth code length[2]. Figure 14a shows that execution errors occur more frequently than constraint errors, especially in lower-level queries, where issues like syntax errors or invalid attribute calls are resolved before constraint validation.

Interestingly, self-improving loops do not have a strong correlation with accuracy gains. As shown in Figure 14b, the most significant accuracy improvements occur in the first iteration of both execution and constraint error checks, with diminishing returns in subsequent loops. Execution

[2]Code length serves as one possible objective proxy for complexity, reducing subjective bias.



(a) Error appearance count **(b) Error reduction efficiency**

Fig. 14. Constraint-guided error reduction analysis

errors, such as unsupported package versions or ambiguous function calls, converge quickly since repeated attempts rarely resolve them. Constraint errors, while less frequent, take longer to stabilize as they involve more complex logical validation. To improve efficiency, MeshAgent limits self-improving loops to $N = 5$. If errors persist, agent abstain and escalate the error to human engineers for updating the constraints with failure analysis.

> ***Takeaway for Q3:*** *LLM for simple queries often fail due to execution issues, while complex ones fail when constraints are not met. Most failures are fixed in the first round of error reduction checks.*

6 User Study: Quality of User Experience

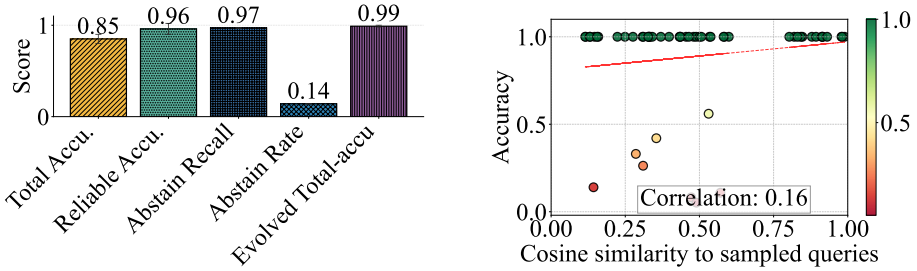
To evaluate MeshAgent with real-world network management users, we conducted a user study comparing it against alternative approaches chosen by participants.

Methodology. We recruited 20 participants with backgrounds in computer science and network management to simulate real-world network traffic analysis engineers. The participants included 17 individuals from a major network service provider company: 15 interns in the networking research group and 2 full-time professionals, also 3 PhD students from a university. MeshAgent was deployed via a web-based system that allowed users to visualize network graphs, input queries, execute MeshAgent, edit code, and review results (Appendix C Figure 17).

Users tested two types of queries: *canned* and *open-ended*. The canned queries consisted of three increasingly complex tasks with predefined ground truth checks. Each user used both MeshAgent and a freely chosen alternative (e.g., ChatGPT, Gemini) and recorded their preferred method along with execution time. For open-ended queries, users submitted any network-related question, reviewed MeshAgent's generated code, and modified it as needed. We logged the time spent on each step, both in-system and through surveys.

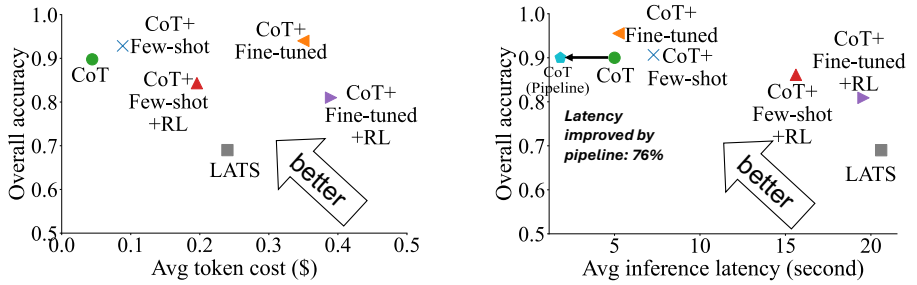
Results. Users rated overall satisfaction of using MeshAgent at 8.3/10. For canned queries, 95% preferred MeshAgent over alternatives, with full preference for the most complex queries. MeshAgent also achieved a 100% success rate, maintaining average latency under 40 seconds, while alternatives were $13 - 21\times$ slower.

For open-ended queries, we collected 200 unique questions, with each participant submitting 10 queries. While 90% of the questions were similar (e.g., identifying the highest-degree nodes), a long-tail distribution of diverse topics also emerged (Figure 3). After collection, we asked domain experts to generate ground truth for each query, and compared these with MeshAgent's responses. The results indicate that MeshAgent achieves 85% total accuracy and 96% reliable accuracy, with a



(a) Accuracy summary **(b) Generalization test**

Fig. 15. MeshAgent's workflow and pre-built constraints shows strong generalization on open-ended user-submitted queries.



(a) Token cost **(b) Query latency**

Fig. 16. Except for LATS, all are variants of MeshAgent. CoT+Few-shot and CoT+Fine-tuned offer efficient token usage and latency, with pipeline optimization providing additional speedup.

high abstain recall (Figure 15a). Notably, MeshAgent's pre-built constraints generalized effectively even to low-similarity queries (Figure 15b), demonstrating strong adaptability to diverse queries.

To assess dynamic improvement, we fed the abstained queries into our semi-automated constraint creation pipeline for failure analysis to generate new constraint entries. After integrating these additional constraints, overall accuracy increased to 99% on open-ended queries (Figure 15a). These findings underscore that MeshAgent not only efficiently extracts underlying invariants from queries, eliminating the need for exhaustive example enumeration, but can also evolve over time by learning from its failures.

Takeaway for Q4: *In a user study with 20 network engineers, MeshAgent consistently outperforms public LLMs in both accuracy and user preference. On user's open-ended queries, MeshAgent's built-in constraints generalize well, achieving 85% accuracy. After incorporating new constraints learned from abstained cases, accuracy increases to 99%. This shows that updating constraints based on abstentions significantly improves system accuracy over time in real-world settings.*

7 LLM Agents Inference Cost and Latency

To evaluate trade-offs among agents using MeshAgent, we compare token cost, inference latency, and accuracy. Overall, as agents become more sophisticated, diminishing returns emerge. The basic CoT approach achieves 90% accuracy at a low cost ($0.04). Few-shot improves accuracy by +2.5% but increases costs by 28%. Fine-tuning adds additional costs beyond inference, as it requires input tokens for model training. RL-enhanced variants, despite being 2.5–5.5× more expensive, reduce accuracy by 5–13%, showing that added complexity does not always improve performance.

Latency analysis reveals that CoT and fine-tuned models maintain practical response times (5.4–6.2s), making them viable for real-time use. Importantly, MeshAgent's pipeline optimization can reduce CoT agents latency by 76%. However, RL-based approaches see a sharp increase (16–19s), limiting their practicality. LATS further underscores these challenges, with 69% accuracy, $0.24 cost, and 20.6s latency. These findings offer insights for network engineers to select the most suitable LLM agents, balancing cost, latency, and accuracy for new applications as needed.

8 Related Work

General program synthesis with LLMs. Recent LLM-based program synthesis includes (1) code selections where multiple samples are generated for choosing a best one based on the consistency of execution results [63] or auto-generated test cases [10]; (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 feedback or reinforcement learning outer loop to help LLMs learn from their errors [9, 11, 64]. We have shown that applying general program synthesis methods is insufficient to answer network management queries correctly.

LLMs for networking. Recent efforts in LLMs for network systems have covered various tasks such as root cause analysis and troubleshooting [12, 60], reproducing research outcomes [77], simulating agents for investigating internet incidents [86], extracting protocol specifications [61], performing data retrieval and analytics on operator logs [31], generating task-specific responses to handle multimodal networking data [76], generating network configurations [72], and using chatbots for datacenter network diagnostics [73]. These proposals mainly focus on the feasibility of specific applications and do not address the challenges faced by network experts in building and adapting LLM agents for each application.

Constraints used in other LLM applications. Modern generative systems increasingly adopt constraint-driven mechanisms to steer outputs both ethically and technically. For instance, Dong et al.[16] introduces systematic pre- and post-generation filters to block unsafe or biased responses. For policy alignment, Bai et al.[4] formulates explicit rule lists (the "AI constitution") to constrain model behavior toward ethical principles. Liu et al. [40] identify low-level (ensures the output adhere to a structured format) and high-level constraints (requires the output to follow semantic and stylistic guidelines without hallucination) from a user-centered perspective. Furthermore, constraint languages and frameworks such as LMQL [5] exemplify broader efforts to codify constraint specification and enforcement in generation pipelines. These studies highlight how explicit constraints form a unifying mechanism for controlling generative models across domains. MeshAgent is the first framework to formalize and implement constraint generation and enforcement specifically for network management tasks.

9 Discussions

Limitation of MeshAgent. Network management involves a broad range of applications. MeshAgent specifically targets those that can be framed as graph manipulation tasks, where the underlying system state or topology can be modeled as a graph, and the goal is to use LLMs to generate code that operates on this graph structure (e.g., capacity planning over datacenter topologies and traffic engineering based on network graphs). However, it does not support applications that cannot be efficiently transformed into graph-based representations. One example is *flow-level network performance monitoring*, which typically requires statistical analysis over time-series data and continuous tracking of metrics like latency, jitter, and packet loss tasks that are better suited to signal processing or time-series models rather than graph manipulation. Another limitation of MeshAgent is the inherent nondeterminism of LLMs output, which leads to the absence of formal

correctness guarantees. Moreover, the completeness of the generated constraint sets can only be evaluated empirically, as it depends on the sample query set.

Evaluating MeshAgent on emerging benchmarks Since MeshAgent was developed without access to domain-specific network benchmarks, we also conducted a real-world user study to ensure representativeness. As future work, MeshAgent can be evaluated using newly available public benchmark datasets. For example, NetPress [84] introduces a dynamic benchmark generation framework tailored to network and system applications. Evaluating MeshAgent's agent capabilities, specifically in terms of correctness, safety (i.e., constraint violations), and latency, on such benchmarks would be a valuable next step. This would also enable direct comparison with vanilla LLMs and other agents on the NetPress leaderboard: netpress.ai.

Augmentation to future, more powerful LLMs. MeshAgent is designed to be model-agnostic, and it has demonstrated ability to complement existing LLM agents. As more advanced LLMs with enhanced reasoning and contextual understanding emerge, they present an opportunity to further improve performance in network management tasks. MeshAgent is designed for seamless integration with next-generation LLMs and advanced techniques such as continual learning, online fine-tuning, and‘ reinforcement learning to dynamically adapt constraints, ensuring the framework evolves alongside rapid AI advancements.

Security vulnerabilities of LLM agents. Deploying network LLM agents exposes them to jailbreaking attacks, where adversaries craft prompts to bypass safeguards and extract sensitive information or execute unauthorized commands. Although ML security remains a broad challenge beyond this paper's scope, potential mitigations include incorporating multiple layers of protection within MeshAgent. Techniques like regex-based blacklisting and semantic similarity checks can filter adversarial queries [80], while reinforcement learning with human feedback (RLHF) [54] can further refine model behavior. Additionally, input and output validation that cross-checks responses against known safe outputs can prevent unintended or malicious actions.

10 Conclusions

We introduced MeshAgent, a framework aimed at improving the workflow of building task-specific LLM agents for network management. By incorporating domain-specific invariants as constraints and developing a semi-automated workflow to simplify their creation, MeshAgent reduces the reliance on extensive domain-specific data while improving both accuracy and reliability. Our evaluation shows that MeshAgent effectively complements existing methods, enhances generated code accuracy, and provides robust mechanisms for managing uncertainty, facilitating the practical adoption of LLMs in critical network management tasks.

Ethics: This work does not raise ethical issues. The user study was reviewed and approved by an institutional review board.

Acknowledgments

We thank all the anonymous reviewers for their constructive feedback and valuable suggestions, which greatly improved the quality of this paper. This work was supported in part by the U.S. NSF grants CNS-2431093, CNS-2415758, and SaTC-2415754. We extend our special thanks to our shepherd for the thoughtful guidance throughout the revision process. We are grateful to Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, and Ranveer Chandra for their early feedback, which helped shape the first version of this work. We thank Sadjad Fouladi for his insightful suggestions, which led to a much better end-to-end system overview figure. Finally, we sincerely appreciate all user study participants for their time and insights.

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). https://doi.org/10.48550/arXiv.2305.10403

[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). https://arxiv.org/abs/2108.07732

[3] Victor Bahl. 2024. Empowering operators through generative AI technologies with Azure for Operators. https://azure.microsoft.com/en-us/blog/empowering-operators-through-generative-ai-technologies\protect\discretionary{\char\hyphenchar\font}{}{}with-azure-for-operators/.

[4] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073* (2022).

[5] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.

[6] Nik Bear Brown. 2024. Enhancing Trust in LLMs: Algorithms for Comparing and Interpreting LLMs. *arXiv preprint arXiv:2406.01943* (2024).

[7] 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)*.

[8] 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). https://doi.org/10.48550/arXiv.2303.12712

[9] 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). https://doi.org/10.48550/arXiv.2303.16749

[10] 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).

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023). https://doi.org/10.48550/arXiv.2304.05128

[12] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.

[13] 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). https://doi.org/10.48550/arXiv.2204.02311

[14] 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). https://arxiv.org/abs/2110.14168

[15] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. 758–759.
[16] Yi Dong, Ronghui Mu, Gaojie Jin, Yi Qi, Jinwei Hu, Xingyu Zhao, Jie Meng, Wenjie Ruan, and Xiaowei Huang. 2024. Building guardrails for large language models. *arXiv preprint arXiv:2402.01822* (2024).
[17] Shangbin Feng, Weijia Shi, Yike Wang, Wenxuan Ding, Vidhisha Balachandran, and Yulia Tsvetkov. 2024. Don't Hallucinate, Abstain: Identifying LLM Knowledge Gaps via Multi-LLM Collaboration. *arXiv preprint arXiv:2402.00367* (2024).
[18] FireMon. [n. d.]. One Simple Misconfiguration: 2.9 Billion Users Down. https://www.firemon.com/blog/one-simple-misconfiguration-2-9-billion-users-down/?t. Accessed: 2025-01-30.
[19] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).
[20] 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. https://doi.org/10.1007/s00607-013-0282-8
[21] Google. [n. d.]. MALT example models. https://github.com/google/malt-example-models, Retrieved on 2023-06.
[22] Google. 2024. Gemini AI for developers. https://ai.google.dev/.
[23] 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)*. https://doi.org/10.1145/3230543.3230555
[24] Aman Gupta, Anup Shirgaonkar, Angels de Luis Balaguer, Bruno Silva, Daniel Holstein, Dawei Li, Jennifer Marsman, Leonardo O Nunes, Mahsa Rouzbahman, Morris Sharp, et al. 2024. RAG vs Fine-tuning: Pipelines, Tradeoffs, and a Case Study on Agriculture. *arXiv preprint arXiv:2401.08406* (2024).
[25] Pouya Hamadanian, Behnaz Arzani, Sadjad Fouladi, Siva Kesava Reddy Kakarla, Rodrigo Fonseca, Denizcan Billor, Ahmad Cheema, Edet Nkposong, and Ranveer Chandra. 2023. A Holistic View of AI-driven Network Incident Management. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets)*. 180–188.
[26] Xinyi Hou, Jiahao Han, Yanjie Zhao, and Haoyu Wang. 2025. Unveiling the Landscape of LLM Deployment in the Wild: An Empirical Study. *arXiv preprint arXiv:2505.02502* (2025).
[27] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *The Tenth International Conference on Learning Representations (ICLR)*.
[28] Yudong Huang, Hongyang Du, Xinyuan Zhang, Dusit Niyato, Jiawen Kang, Zehui Xiong, Shuo Wang, and Tao Huang. 2024. Large language models for networking: Applications, enabling techniques, and challenges. *IEEE Network* (2024).
[29] 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)*. https://doi.org/10.1145/1298306.1298349
[30] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. 2021. Hey, lumi! using natural language for {intent-based} network management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 625–639.
[31] Manikanta Kotaru. 2023. Adapting Foundation Models for Operator Data Analytics. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 172–179.
[32] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. 2025. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321* (2025).
[33] 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)*. https://doi.org/10.1145/2069216.2069227
[34] Sihyung Lee, Kyriaki Levanti, and Hyong S. Kim. 2014. Network monitoring: Present and future. *Comput. Networks* 65 (2014). https://doi.org/10.1016/j.comnet.2014.03.007
[35] Aris Leivadeas and Matthias Falkner. 2023. A Survey on Intent-Based Networking. *IEEE Commun. Surv. Tutorials* 25, 1 (2023), 625–655.
[36] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[37] 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). https://doi.org/10.48550/arXiv.2203.07814

[38] Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient Fine-tuning for LLM-based Recommendation. *arXiv preprint arXiv:2401.17197* (2024).

[39] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[40] Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. 2024. " we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–9.

[41] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. *CoRR* abs/2307.03172 (2023).

[42] LlamaIndex. 2024. Building Performant RAG Applications for Production. https://docs.llamaindex.ai/en/stable/optimizing/production_rag/.

[43] Sifan Long, Jingjing Tan, Bomin Mao, Fengxiao Tang, Yangfan Li, Ming Zhao, and Nei Kato. 2025. A Survey on Intelligent Network Operations and Performance Optimization Based on Large Language Models. *IEEE Communications Surveys & Tutorials* (2025). doi:10.1109/COMST.2025.3526606

[44] Sathiya Kumaran Mani, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Ranveer Chandra, Eliran Azulai, Narayan Annamalai, Deepak Bansal, and Srikanth Kandula. 2023. Securing Public Clouds using Dynamic Communication Graphs. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 272–279.

[45] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 196–204.

[46] 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)*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). https://doi.org/10.18653/v1/2020.acl-main.173

[47] Microsoft. [n. d.]. Starter Resource Graph query samples. https://learn.microsoft.com/en-us/azure/governance/resource-graph/samples/starter, Retrieved on 2023-12.

[48] Microsoft. 2023. Azure Resource Graph documentation. https://learn.microsoft.com/en-us/azure/governance/resource-graph/ Retrieved December 2023.

[49] Microsoft. 2024. Azure OpenAI Service. https://azure.microsoft.com/en-us/products/ai-services/openai-service.

[50] 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)*. https://www.usenix.org/conference/nsdi20/presentation/mogul

[51] OpenAI. [n. d.]. Introducing OpenAI o1-preview. https://openai.com/index/introducing-openai-o1-preview/.

[52] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774

[53] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

[54] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[55] Archit Parnami and Minwoo Lee. 2022. Learning from few examples: A summary of approaches to few-shot learning. *arXiv preprint arXiv:2203.04291* (2022).

[56] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601* (2021).

[57] Sasha Ratkovic. 2017. What is Intent-Based Networking? https://blogs.juniper.net/en-us/enterprise-cloud-and-transformation/what-is-intent-based-networking.

[58] Vipula Rawte, Amit P. Sheth, and Amitava Das. 2023. A Survey of Hallucination in Large Foundation Models. *CoRR* abs/2309.05922 (2023).

[59] Gartner Research. 2017. Innovation Insight: Intent-Based Networking Systems. https://www.gartner.com/en/documents/3599617.

[60] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-based Agents for Root Cause Analysis. *arXiv preprint arXiv:2403.04123* (2024).

[61] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 41–47.

[62] Prashanth Shenoy. 2017. Journey to an Intent-based Network. https://blogs.cisco.com/networking/journey-to-an-intent-based-network.

[63] 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)*. https://aclanthology.org/2022.emnlp-main.231

[64] 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).

[65] SIFF. [n. d.]. Configuration Outages That We Should Be Learning From. https://www.siff.io/configuration-outages-that-we-should-be-learning-from/?t. Accessed: 2025-01-30.

[66] 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). https://doi.org/10.48550/arXiv.2212.13138

[67] 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. https://doi.org/10.1016/j.jnca.2020.102538

[68] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[69] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[70] 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). https://doi.org/10.48550/arXiv.2302.13971

[71] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *Proc. VLDB Endow.* 15, 11 (2022). https://www.vldb.org/pvldb/vol15/p2921-trummer.pdf

[72] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. NetConfEval: Can LLMs Facilitate Network Configuration? *Proceedings of the ACM on Networking* 2, CoNEXT2 (2024), 1–25.

[73] Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, et al. 2024. {NetAssistant}: Dialogue Based Network Diagnosis in Data Center Networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2011–2024.

[74] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *The Tenth International Conference on Learning Representations (ICLR)*.

[75] 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)*. http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[76] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. 2024. NetLLM: Adapting Large Language Models for Networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 661–678.

[77] Qiao Xiang, Yuling Lin, Mingjun Fang, Bang Huang, Siyong Huang, Ridi Wen, Franck Le, Linghe Kong, and Jiwu Shu. 2023. Toward Reproducing Network Research Results Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 56–62.

[78] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms. *arXiv preprint arXiv:2306.13063* (2023).

[79] Ziwei Xu, Sanjay Jain, and Mohan S. Kankanhalli. 2024. Hallucination is Inevitable: An Innate Limitation of Large Language Models. *CoRR* abs/2401.11817 (2024).

[80] Zihao Xu, Yi Liu, Gelei Deng, Yuekang Li, and Stjepan Picek. 2024. LLM Jailbreak Attack versus Defense Techniques–A Comprehensive Study. *arXiv preprint arXiv:2402.13457* (2024).

[81] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[82] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[83] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

[84] Yajie Zhou, Jiajun Ruan, Eric S Wang, Sadjad Fouladi, Francis Y Yan, Kevin Hsieh, and Zaoxing Liu. 2025. NetPress: Dynamically Generated LLM Benchmarks for Network Applications. *arXiv preprint arXiv:2506.03231* (2025).

[85] 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)*. https://doi.org/10.1145/3387514.3406214

[86] Yajie Zhou, Nengneng Yu, and Zaoxing Liu. 2023. Towards Interactive Research Agents for Internet Incident Investigation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 33–40.

Algorithm 1 Stepwise Code Execution with Error Reduction

Require: Q: input network query; C: set of extracted constraints; N: maximum number of regeneration attempts per step.

procedure ExecuteQuery(Q, C, N)
  $\mathcal{S} \leftarrow$ Decompose(Q) ▷ $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$: sequence of reasoning steps
  for $i = 1$ to $|\mathcal{S}|$ **do**
    $s \leftarrow \mathcal{S}[i]$ ▷ s: current step
    $c \leftarrow$ GetConstraint(s, C) ▷ c: constraint relevant to step s
    for $t = 1$ to N **do**
      $code \leftarrow$ LLM(Q, s, c) ▷ Generate code for step s
      if ¬PassExec($code$) **then**
        $e_{\text{exec}} \leftarrow$ ExecErr($code$) ▷ Capture execution-error (e.g., syntax)
        $code \leftarrow$ LLMFix(Q, s, c, e_{exec}) ▷ Regenerate code using feedback
      else if ¬PassConstraint($code, c$) **then**
        $e_{\text{cons}} \leftarrow$ ConstraintErr($code, c$) ▷ Capture constraint-error
        $code \leftarrow$ LLMFix(Q, s, c, e_{cons}) ▷ Regenerate code using feedback
      else
        Save($i, code$) ▷ Store valid code for step i
        break
      end if
    end for
    if $t > N$ **then**
      Report(Q, s, c, i) ▷ Escalate to human with detailed failure report
      return FAIL
    end if
  end for
  return Assemble() ▷ Combine all step-wise code into a final solution
end procedure

A Query examples

We list more queries for each application in Table 7. The full query list and their ground truth will be released afterward.

Apps	Query examples
TA	What are max degree and min degree in the graph?
	Assign a unique color for each /16 IP address prefix and color the nodes accordingly.
	Color the size of the node with max degree green and double its size.
	Find nodes with top 10 number of degrees, list nodes, labels, and number of degrees.
	Color the nodes that can be connected to nodes with labels `app:prod` with green.
	Cut the graph into two parts such that the number of edges between the cuts is the same. Color two parts with red and blue.
	Identify the unique labels in the graph and create a new graph with a node for each unique label.
	Calculate the total byte weight of edges incident on each node, cluster into 5 groups using k-means, and color nodes by cluster.
	How many maximal cliques are in the graph?
	Remove the label `type:VM` from all the nodes. Create a new graph with the same nodes and edges.
	Bisect the network such that the number of nodes on either side of the cut is equal.
	How many unique nodes have edges to nodes with label `app:prod` and don't contain the label `app:prod`?
	Show me the unique IP address prefix and the number of nodes per prefix.
	Delete all edges whose byte weight is less than the median byte weight in the graph.
	What is the average byte weight and connection weight of edges incident on nodes with labels `app:prod`?
MALT	Add a new `packet_switch` `ju1.a1.m1.s4c7` on jupiter 1, aggregation block 1, domain 1, with 5 ports.
	Update the `physical_capacity_bps` from 1000 Mbps to 4000 Mbps on node `ju1.a1.m1.s2c2.p14`.
	Identify all `CONTROL_POINT` nodes that are also `PACKET_SWITCH` type within the `AGG_BLOCK` type node `ju1.a4.m4`.
	Display all `CONTROL_DOMAIN` that contains at least 3 `CONTROL_POINT`.
	What is the bandwidth on packet switch `ju1.a2.m1.s2c2` in Mbps?
	Find the first and the second largest `Chassis` by capacity on `ju1.a1.m1`.
	Show the average `physical_capacity_bps` for all `PORT` in all `PACKET_SWITCH`.
	For each `AGG_BLOCK`, list the number of `PACKET_SWITCH` and `PORT` it contains.
	Identify all `PACKET_SWITCH` nodes in `AGG_BLOCK` `ju1.a1.m1` and calculate their average `physical_capacity_bps`.
	Find all `PACKET_SWITCH` nodes that have capacity more than the average.
	Remove packet switch `ju1.a1.m1.s2c4` out from `Chassis` c4, how to balance the capacity between `Chassis`?
	Remove five `PORT` nodes from each `PACKET_SWITCH` while maintaining balanced capacity.
	Identify all paths from `CONTROL_DOMAIN` `ju1.a1.dom` to `PORT` `ju1.a1.m1.s2c1.p1`, ranked by hop count.
	Analyze the redundancy level of each `SUPERBLOCK` by calculating alternative paths between `CHASSIS` pairs.
	Optimize topology by identifying removable `PACKET_SWITCH` nodes that won't affect `CONTROL_DOMAIN` connectivity.
	Determine optimal placement of new `PACKET_SWITCH` `ju1.a1.m1.s2c9` with 5 `PORT`s to balance `AGG_BLOCK` capacity.
CRG	Show the most connected five `VM` nodes by their name and their `osType`.
	Find all nodes that can be connected to virtual networks nodes with `addressPrefixes` as `10.0.0.1` and `port` as 26.
	Extract all `VM` nodes with OS type as `linux` and their connected nodes with links.
	Find all network security groups nodes that allow inbound traffic.
	With all `VM` nodes that have name `Subnet-2`, list the top five node degrees.
	How many virtual networks nodes allow `port` 26?
	How many network interfaces nodes have properties `virtualnetworks` as `Subnet-1`?
	Find the node ID with max degree and min degree.
	How many nodes have links to network security groups nodes?
	How many isolated nodes without any links exist in the graph?
	Show me the unique `addressPrefixes` for virtual networks nodes and the number of nodes per prefix.
	Cut the graph into two parts such that the number of virtual networks nodes between the cuts is the same.
	How many nodes, except from type `networksecuritygroups`, have more than ten links in the graph?
	Count network security groups nodes that are related to inbound traffic.
	Identify all paths from the network interfaces nodes to virtual networks node with name `Subnet-1`.
	For all network security groups nodes with name `AllowVnetInBound`, list all ports and rank them based on priority.

Table 7. Sample query lists for each application

B Prompts.

Prompt #1: Generating Graph Structure Constraints

You are a domain expert tasked with specifying precise and actionable constraints for a graph-based topology system. Based on provided topology features, generate a structured JSON file that captures the expected structural and operational rules of the system.
Output Format: Each constraint must be in the following JSON format:

```
[
  {
    "id": "<sequential_number>",
    "label": "<category>, <subcategory>",
    "invariant": "<constraint_in_natural_language>",
    "validation_test":
        def verify_<constraint_name>(self.graph):
            """
            Validation check: <constraint_description>
            """
            # Return True, "" if valid
            # Raise Exception("<error_message>") if violated
  }
]
```

Constraint Writing Guidelines:

- Include examples when helpful (e.g., "For example, a PORT node name is ju1.a1.m1.s2c1.p3")
- Specify data types and formats (e.g., "type must be a list")
- Include all unique node and edge type values
- Define hierarchical relationships in graph clearly
- Specify validation rules and checks

Input Feature Information You'll Receive:

- Node types and their hierarchical relationships
- Edge types and their meanings
- Attribute specifications for each node/edge type
- Naming conventions
- Operational requirements (how to add, update, calculate)

Example Constraint Patterns:

- "Hierarchy: PARENT contains CHILD, CHILD contains GRANDCHILD"
- "When adding new nodes, you should also add edges based on their relationship with existing nodes."
- "To find node based on type, check the name and type list. For example, [validation_example]."
- "When calculating capacity of a node, you need to sum the physical_capacity_bps on the PORT of each hierarchy contains in this node."

Prompt #2: Generating App-Specific Constraints via Failure Analysis

You are a domain expert tasked with analyzing failure cases in graph-based topology systems and deriving precise constraints to prevent such failures in the future. Given an **original user query**, the corresponding **failure message**, and the **correct expected answer**, your task is to diagnose the root cause of the failure and convert this insight into a formal, enforceable constraint.

Output Format: Your output should be a structured JSON object following the format below:

```
[
  {
    "id": "<sequential_number>",
    "label": "<category>, <subcategory>",
    "invariant": "<constraint_in_natural_language>",
    "validation_test":
        def verify_<constraint_name>(self.graph):
            """
            Validation check: <invariant_description>
            """
            # Return True, "" if valid
            # Raise Exception("<error_message>") if violated
  }
]
```

Failure Analysis Guidelines:

- Identify the key gap or incorrect assumption that led to the failure.
- Map the failure to a missing or violated constraint.
- Generalize the insight into a reusable constraint that applies across similar queries or applications.
- Constraints must be specific, testable, and grounded in the topology semantics.
- Use examples from the query or answer when helpful.

Input Provided:

- **Query:** The original user query
- **Failure Message:** The system's error or misbehavior description
- **Correct Answer:** A verified correct solution or expected outcome

Example Failure Patterns:

- **Invalid Topology:** "A PORT must always be linked to a Packet Switch"
- **Computation Mismatch:** "Capacity attribute only exists at Port level nodes."
- **Naming Conflict:** "Node names must be globally unique within the same topology context."

C User Study Interface

Figure 17 depicts the screenshot of how MeshAgent generates and executes LLM-produced code in response to a network operator's natural language query. Each graph node included attributes such as color, size, and label, while edges contained network traffic attributes such as byte weight, connection weight, and packet weight. This method addresses the explainability challenge by

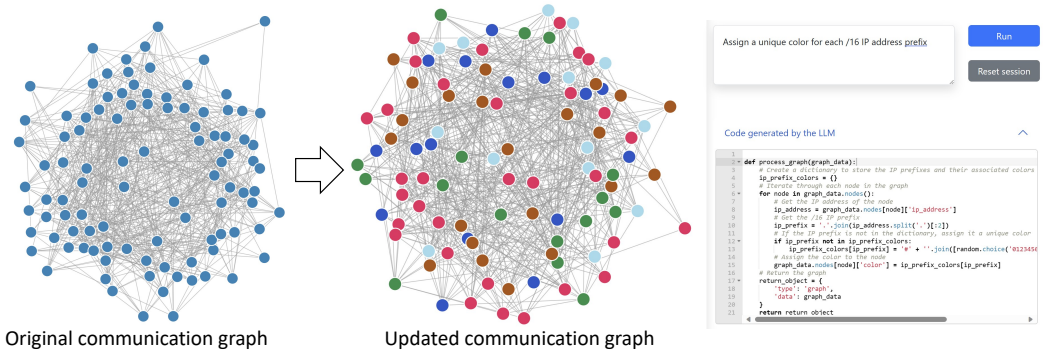


Fig. 17. An example of how a natural-language-based network management system generates and executes a program in response to a network operator's query: "Assign a unique color for each /16 IP address prefix". The system displays the LLM-generated code and the updated communication graph.

enabling network operators to examine the code and understand the techniques used by LLMs to derive answers while assessing their accuracy. Moreover, it overcomes both scalability and privacy concerns by removing the necessity to transfer network data to LLMs, as the input for LLMs is the natural language query and the output solely comprises LLM-generated code. Users are also able to verify and modify the code if they want, once a code is verified, it is added to the *library* RAG for future query reference.

D Invariants of Applications

We list the extracted invariants of each application in Table 8, Table 9, Table 10.

Received July 2025; revised September 2025; accepted October 2025

ID	Label	Invariant
1	graph, type	The data is represented as a `networkx` graph made up of nodes representing virtual machines in a network.
2	node, attribute	Each node has attributes such as a unique `ip_address` and one or more labels of the form `key=value`.
3	node, label	Each node has an attribute `labels`.
4	node, label	Each node has an attribute `labels`; it is a list, for example: [`app:test`, `app:prod`].
5	edge	Each edge connects two nodes if there is a data connection between the virtual machines represented by the nodes.
6	edge, attribute	Each edge has attributes `byte_weight`, `connection_weight`, and `packet_weight`, represented as ratios of the total number of bytes, connections, and packets of the entire network.
7	node, attribute	Each node has an attribute `ip_address`; this should be used when checking by IP address.
8	node, add	Adding new nodes needs to consider corresponding edges.
9	node, remove	There should not be any nodes in the graph that are not connected to any other nodes.
10	edge, remove	There should not be any edges in the graph that are not connected to any other nodes.

Table 8. Invariants list for App-TA.

ID	Label	Invariant
1	graph, type	The graph is directed and each node has a `name` attribute to represent itself.
2	node, type	Each node has a `type` attribute in the format of `EK_{TYPE}`. This is important: `type` must be a list, can include [`EK_SUPERBLOCK`, `EK_CHASSIS`, `EK_RACK`, `EK_AGG_BLOCK`, `EK_JUPITER`, `EK_PORT`, `EK_SPINEBLOCK`, `EK_PACKET_SWITCH`, `EK_CONTROL_POINT`, `EK_CONTROL_DOMAIN`].
3	node, attribute	Each node can have other attributes depending on its type.
4	edge, type	Each directed edge also has a `type` attribute, including `RK_CONTAINS`, `RK_CONTROL`.
5	relationship	You should check relationship based on edge, check name based on node attribute.
6	node, add	Adding new nodes needs to consider all hierarchy. For example, adding a new switch requires adding it to the corresponding jupiter, aggregation block, and domain.
7	edge, add	When adding new nodes, you should also add edges based on their relationship with existing nodes.
8	PORT, attribute	Each `PORT` node has an attribute `physical_capacity_bps`.
9	PORT, name	For example, a `PORT` node name is `ju1.a1.m1.s2c1.p3`.
10	capacity	When calculating capacity of a node, sum the `physical_capacity_bps` on the `PORT`s contained in this node.
11	node, hierarchy	Hierarchy: `CHASSIS` contains `PACKET_SWITCH`, `JUPITER` contains `SUPERBLOCK`, etc.
12	graph, add	When creating a new graph, filter nodes and edges with attributes from the original graph.
13	graph, update	When updating a graph, always create a graph copy; do not modify the input graph.
14	node, attribute	Packet-switch nodes also have a switch-location attribute `switch_loc` in node attribute `packet_switch_attr`.
15	node, type	To find node based on type, check the name and type list. Example: `[node[0] == 'ju1.a1.m1.s2c1' and 'EK_PACKET_SWITCH' in node[1]['type']]`.

Table 9. Invariants list for App-MALT.

ID	Label	Invariant
1	graph, type	The data is represented as a `networkx` graph made up of cloud resource graph in Azure network.
2	node, type	Node `type` can be `virtualmachines`, `Networkinterfaces`, `virtualnetworks`, or `networksecuritygroups`.
3	node, name	Node `name` is a string depending on its `type`.
4	node, properties, virtualmachines	When `type=virtualmachines`, `properties` has `osType`, `Networkinterfaces`. When `type=Networkinterfaces`, `properties` has `virtualnetworks`, `Networkinterfaces`. When `type=virtualnetworks`, `properties` has `provisioningState`, `addressPrefixes`, `port`. When `type=networksecuritygroups`, `properties` has `protocol`, `addressPrefixes`, `port`, `priority`.
5	node, properties, Networkinterfaces	When `type=Networkinterfaces`, `properties` has `virtualnetworks`, `Networkinterfaces`. When `type=virtualnetworks`, `properties` has `provisioningState`, `addressPrefixes`, `port`. When `type=networksecuritygroups`, `properties` has `protocol`, `addressPrefixes`, `port`, `priority`.
6	node, properties, virtualnetworks	When `type=virtualnetworks`, `properties` has `provisioningState`, `addressPrefixes`, `port`. When `type=networksecuritygroups`, `properties` has `protocol`, `addressPrefixes`, `port`, `priority`.
7	node, properties, networksecuritygroups	When `type=networksecuritygroups`, `properties` has `protocol`, `addressPrefixes`, `port`, `priority`.
8	edge, virtualmachines, Networkinterfaces	A `virtualmachines` node can be connected to a `Networkinterfaces` node if they have the same value for `Networkinterfaces` in `properties`.
9	edge, Networkinterfaces, virtualnetworks	A `Networkinterfaces` node can be connected to a `virtualnetworks` node if they share the same value for `addressPrefixes` in `properties`.
10	edge, virtualnetworks, networksecuritygroups	A `virtualnetworks` node can be connected to a `networksecuritygroups` node if they have the same `addressPrefixes` and `port` in `properties`.
11	node, VM	VM node refers to a `virtualmachines` type node in the graph.
12	node, network interfaces	Network interfaces node refers to a `Networkinterfaces` type node in the graph.
13	node, virtual networks	Virtual networks node refers to a `virtualnetworks` type node in the graph.
14	node, network security groups	Network security groups node refers to a `networksecuritygroups` type node in the graph.
15	inbound, traffic	When checking inbound traffic, verify whether `InBound` is contained in the node's `name`.

Table 10. Invariants list for App-CRG.