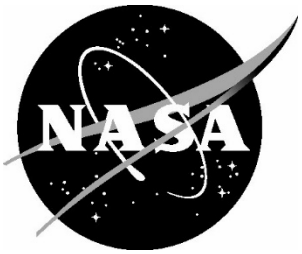
NASA/TM–20260002739



# Progressing Advanced Air Mobility Research Through QLoRA Based Fine-Tuning of Local LLMs for Graph Database Interactions

*Braxton R. VanGundy*
*NASA Langley Research Center, Hampton, Virginia*

*Joshua Bonner*
*CACI International, Inc., NASA Langley Research Center, Hampton, Virginia*

*Mikhail K. Schneide, Isabella M. Do*
*Guardians of Honor, LLC, NASA Langley Research Center, Hampton, Virginia*

April 2026

# NASA STI Program Report Series

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

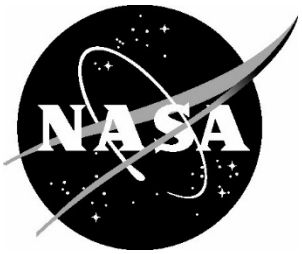
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

NASA/TM–20260002739



# Progressing Advanced Air Mobility Research Through QLoRA Based Fine-Tuning of Local LLMs for Graph Database Interactions

*Braxton R. VanGundy*
*NASA Langley Research Center, Hampton, Virginia*

*Joshua A. Bonner*
*CACI International, Inc., NASA Langley Reseach Center, Hampton, Virginia*

*Mikhail K. Schneide, Isabella M. Do*
*Guardians of Honor, LLC, NASA Langley Reseach Center, Hampton, Virginia*

National Aeronautics and
Space Administration

NASA Langley Research Center
Hampton, VA

April 2026

**Acknowledgements**

Our team would like to acknowledge the efforts and assistance of Barclay Brown, of Collins Aerospace, who worked with us to perform much of the foundational research that made this paper possible. We would also like to thank Nipa Phojanamongkolkij and Ian Levitt, of NASA Langley Research Center, for providing excellent project and research guidance.



Available from:

NASA STI Program / Mail Stop 050
NASA Langley Research Center
Hampton, VA 23681-2199

## Abstract

Most recent locally deployable Large Language Models (LLMs) demonstrate adequate performance regarding Cypher query language generation out of the box, however, these models tend to fall short when performing complex operations such as advance graph analysis tasks and full text searches. To address these shortfalls, our approach employs Quantized Low-Rank Adaptation (QLoRA) fine-tuning combined with specialized prompt engineering to enhance a local LLM's ability to translate plain English language into equivalent Cypher query language. The fine-tuned models generated by our team showed substantial improvement in generating accurate Cypher queries for complex graph operations, successfully enabling non-technical users to perform sophisticated analysis of Advanced Air Mobility system architectures using natural English language. Additionally, through the use of an extensible system prompt during the fine-tuning process, tuned models demonstrated easy adaptability to new graph database schemas through only minor prompt modifications, providing a practical approach for developing a single model that can serve as a translational layer for graph databases across a wide variety of domains without the need for additional fine-tuning.


## 1. Introduction

A research effort led by the Air Mobility Pathfinders (AMP) project at the National Aeronautics and Space Administration (NASA) is developing a system architecture for safe, secure, and scalable Advanced Air Mobility (AAM) operations [1]. The AMP project's Operational Concepts, Architecture, and Requirements Integration (OCARI) team is using a Model Based Systems Engineering (MBSE) approach for integration, interoperability, and traceability of AAM ecosystems centered around urban air taxi services. The team's goal is to define the performers, capabilities, and behaviors needed for system feasibility, readiness, and interoperability through the development of reference, solution, and test architectures [2]. This system architecture is complex to navigate due to the hundreds of interconnected behaviors, performers, and capabilities that are found across several different AAM use cases represented within the architecture. Such a complex web of interconnected nodes and links, seen in Figure 1, makes the architecture a prime candidate for representation using a knowledge graph. Knowledge graphs use a graph topology to represent complex data structures. Graphs utilize nodes and edges that describe the relationships, or lack thereof, between elements. Nodes that represent various stages of a sequence can have assigned metadata and directional edges. Taken together, this provides a contextualized matrix of information that can be difficult to parse in non-graph formats.

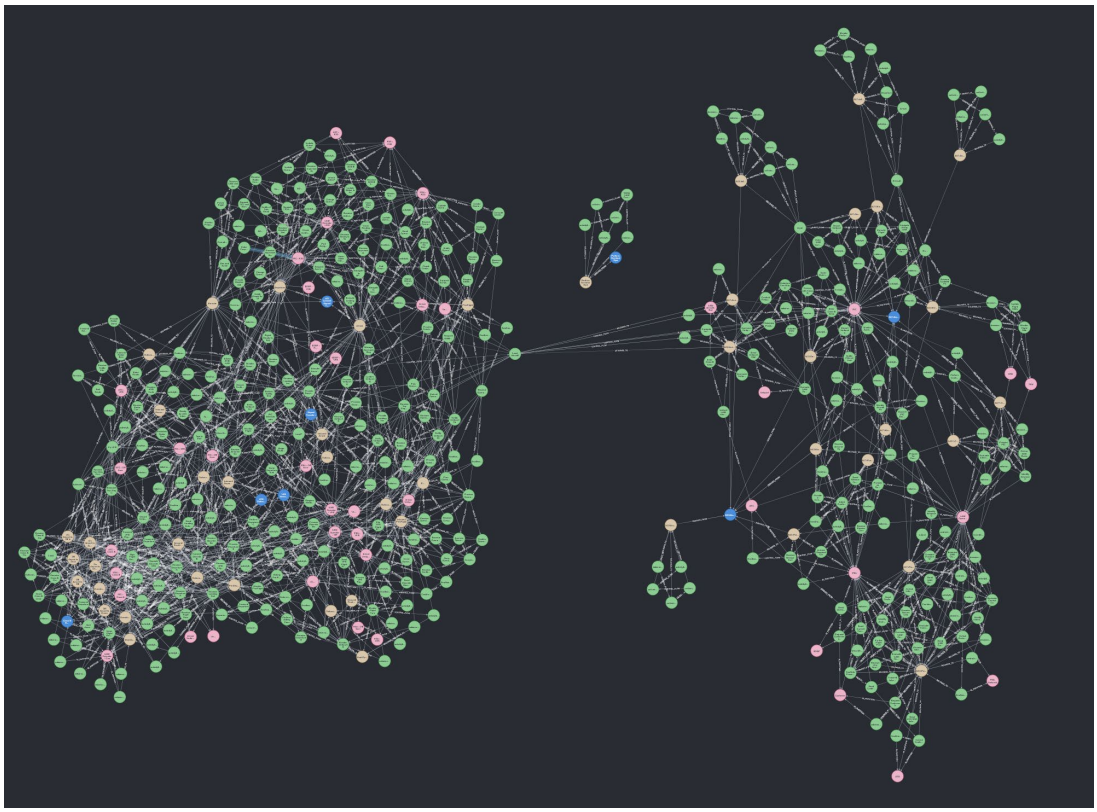


*Figure 1. Visualization of a subset of the AAM Architecture graph structure within the Neo4j graph database.*

Many graph database management systems (GDMS) also provide easy access to advanced graph data science (GDS) algorithms such as community detection, link prediction, and various centrality measures which can be employed to further analyze the efficacy of the relationships in the network. Though these capabilities greatly simplify the process of navigating and performing analysis on the AAM system architecture, the language used to query the graph database, known as "Cypher", is non-trivial to learn and queries can be several lines long. An example of a multi-line Cypher query used to perform link prediction can be seen in Figure 2 below.

```
MATCH (n:Actor{name:'Vertiport Operator'}), (b:`Roadmap Requirement`) WHERE NOT
  (n)-[]-(b) WITH n.name AS input_name, n.text AS input_description, b.name AS
 predicted_roadmap_requirement_name, b.text AS roadmap_requirement_description,
   gds.alpha.linkprediction.adamicAdar(n, b) AS score WHERE score > 0 RETURN
      input_name, input_description, predicted_roadmap_requirement_name,
      roadmap_requirement_description, score ORDER BY score DESC LIMIT 2
```

*Figure 2. An example of a complex Cypher query used to call link prediction capabilities.*

To make these capabilities accessible for the everyday user, our team employed an LLM to act as a translational layer between the user and the graph database. In earlier works published on this topic, a commercial service, Open AI's Chat GPT-4, was utilized as the translation component. This approach proved to be cost prohibitive and, at the time, the Chat GPT-4 service was not approved for use with sensitive data within our organization which significantly limited the usefulness of such a concept [3]. In an effort to reduce operational costs and expand the applicability of the developed prototype to non-public data, our team turned to deploying local LLMs to a Linux server using off the shelf graphics processing units (GPUs). Testing using earlier models such as Mistral AI's Mistral Instruct V0.1 and V0.2 showed poor performance out of the box with complex Cypher queries such as the one in Figure 2 (see the "Results" section for performance metrics). As local models continued to advance during our research, their out of the box Cypher generation performance improved significantly. However, later models such as Mistral Nemo 12B still showed poor performance when it came to correctly utilizing full text search

capabilities within Neo4j. Since performing a full fine-tune of the model was out of reach due to resource constraints, our team instead employed a Low-Rank Adaptation (LoRA) based fine-tuning approach with a focus on making the tuned version of the model easily extensible to other graph database schemas outside the field of AAM without the need for any additional fine-tuning. Using this fine-tuned model, a digital assistant was deployed, allowing AAM researchers to perform analysis on the node-link structure of the AAM architecture using plain English language. This interface provides researchers with access to powerful algorithms for analysis of the architecture's structure while completely abstracting the complicated Cypher query language from the user's view.

# 2. Knowledge Graphs

Because Advanced Air Mobility entails a sequence of checks, activities, and processes, it is a perfect use case for deployment as a knowledge graph. For example, an Electric Vertical Take-off and Landing (eVTOL) vehicle will go through several stages from departure to arrival, including preflight checks and control communication, each of which is translated into a node within the graph. Once the data is in a knowledge graph, several GDS algorithms can be employed to further analyze the efficacy of the relationships between different nodes.

## 2.1 AAM Graph Database Ontology

The AAM Neo4j Graph Database employs a hierarchical structure designed to capture the relationships between Use Cases and their constituent Activities, which are accomplished by a sequence of Behaviors undertaken by Performers. The ontology focuses on the Behaviors as the core operations of a sequence diagram. These nodes and corresponding relationships are represented in our graph database schema as seen in Figure 3. A description of each of the node types in the database is provided below:

- **Use Case Nodes** serve as the top-level organizational entities within the graph. These represent the scenarios or profiles by which we can examine behavior relationships and traversal patterns. Each use case may contain metadata that contextualizes its latent behaviors.
- **Activity Nodes** represent the operational phase that comprise use cases. These nodes maintain sequence information and relationships.
- **Behavior Nodes** define the specific actions, procedures, responses, or protocols required to accomplish each activity. These nodes capture both automated and manual functions that govern operation execution.
- **Performer Nodes** identify the entities responsible for executing behaviors. Performers encompass a range of operators including but not limited to automated systems, human actors, aircraft platforms, and ground infrastructure.

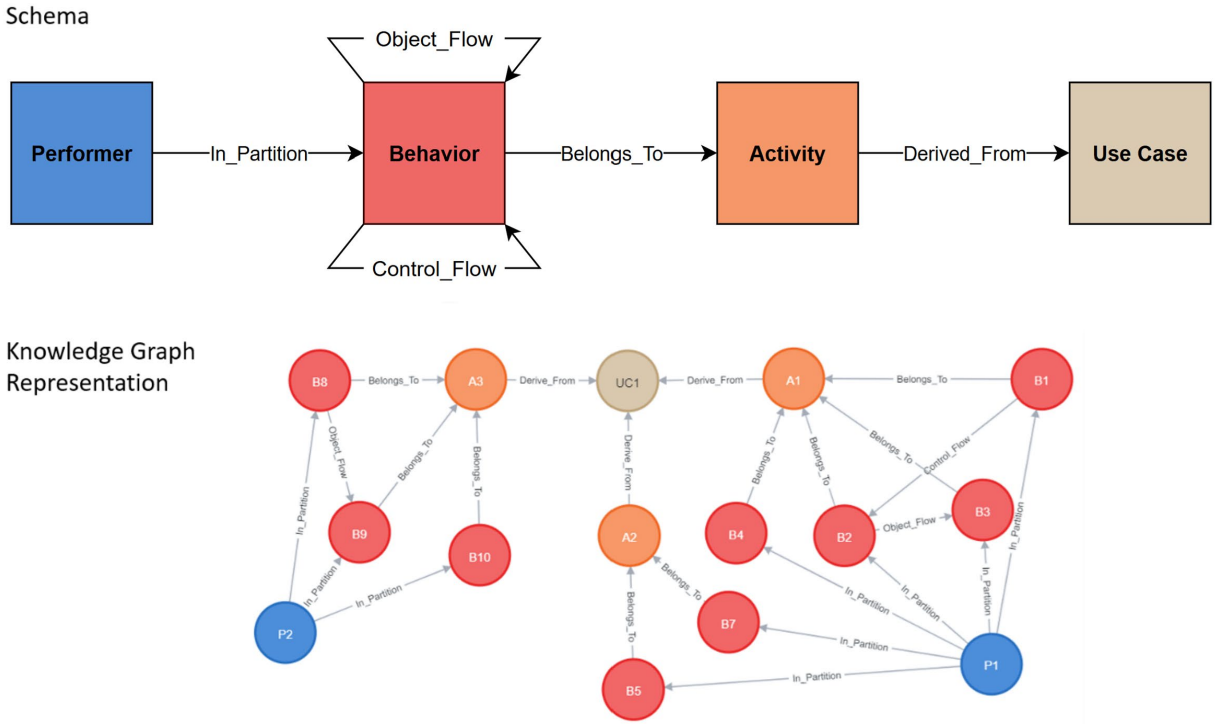



*Figure 3. Graph Database Schema*

Once the graph is projected, the GDS algorithms of particular interest are:

- **Louvain community detection** identifies clusters of densely connected nodes within the network by optimizing modularity, which measures the strength of division into distinct communities [4]. For our purposes, Louvain clustering can reveal natural groupings of behaviors that frequently occur together or share operational dependencies, such as pre-flight safety checks, air traffic coordination sequences, or emergency response protocols. These communities often correspond to functional operational phases or specialized procedural domains within AAM operations.

- **Page rank** measures the relative importance of nodes based on both the number of incoming connections and the importance of the nodes making those connections [5]. In the AAM context, behaviors with high page rank scores represent foundational operations that are frequently referenced or depended upon by other critical behaviors across multiple use cases. Identifying high page rank behaviors is useful for understanding which operations have the greatest influence across the entire system.

- **Betweenness centrality** measures how often a node acts as a bridge along the shortest path between two other nodes in the network [6]. In the AAM context, behaviors with high betweenness centrality represent critical bottlenecks or gateway operations that must be traversed to move between different operational phases. Identifying high betweenness behaviors is useful for understanding operational dependencies between sections and communities of the graph, and potential failure points, as disruption to these nodes could significantly impact the connectivity and flow of the entire AAM operation.

- **Link prediction algorithms** analyze the existing graph structure to identify potential missing relationships or predict future connections between nodes [7]. This capability is particularly useful for operational planning since it can predict which behaviors might follow from operational states, enabling contingency planning and management, as well as operational optimization.

- **Full text search** in Neo4j represents an information retrieval mechanism that extends traditional graph database capabilities by integrating Apache Lucene's indexing with the Cypher query language. This implementation allows for textual queries across graph structures while preserving the semantic relationships in the connected data. Unlike conventional relational database text searches, the full text search maintains graph context, enabling subsequent traversal operations on search results and facilitating hybrid analysis that combines textual relevance with structural relationship analysis [8].

# 3. Local Large Language Models

Large Language Models are an evolutionary step in deep neural networks, building on earlier breakthroughs in the field of natural language processing such as word and sentence vectorization, recurrent neural networks, and long short-term memory networks. Thanks to a new technique known as attention, which allows weights to be assigned to each word based on their relevance, LLMs are able to learn language with words in proper, natural sequence, greatly increasing the naturalness of the language generated [9]. As LLMs continue to increase in size, often measured by the number of "parameters" within the model, they gain new, sometimes unexpected capabilities, including the ability to answer questions of varying levels of complexity, translate from one language to another, and generate software source code in a number of languages based only on their extensive reading of material in that language.

There are many commercial LLMs available through online services in the consumer space, many of these names will probably sound familiar such as Open AI's GPT, Anthropic's Claude, and xAI's Grok models [10][11][12]. These enterprise models consist of hundreds of billions of parameters, requiring entire datacenters to run. While these enterprise models are extremely powerful, their size and restrictive licenses can make them difficult to customize for domain specific tasks, such as the Cypher query translational layer that is the focus of this paper. Additionally, there are concerns with running proprietary and sensitive information through these services since all processing occurs external to the user's local machine. However, it is worth mentioning that as these services continue to evolve, some have implemented the ability to fine-tune models through their developer platforms, making them more adaptable to domain specific problems [10], though this still does not address the issues of data sensitivity and privacy.

Local Large Language Models provide similar capabilities as their commercial counterparts while giving users the option to download the models and run them on their personal computers through open-source inference software libraries such as llama.cpp or vLLM [13][14]. Many of these local LLMs are small enough in size, after quantization methods are applied, where they can run on a single mid-range consumer GPU, such as a GTX 3060. There are hundreds of options to choose from in the local LLM space, our team prefers to use models that are licensed under the Apache 2.0 license and are between 7 and 14 billion parameters in size so they will fit within the memory constraints of our GPUs. Mistral NeMo 12B, developed by Mistral AI, meets both aforementioned requirements, has a large context window of 128K tokens [15], and provided adequate Cypher query generation performance. Furthermore, since it is a locally deployed model, all data sent to the LLM stays on the machine that the LLM is deployed to, allowing the user to keep sensitive information behind their organization's firewall.

Most of the content in this paper focuses on models either developed directly by Mistral AI or models that use Mistral AI models as the base. Starting from late 2023, we consistently found that for our application, the Mistral models either outperformed all other similarly sized local LLMs or provided an optimal licensing structure when compared to the other licenses on models that would have provided better performance. The content in this paper can be generalized well beyond this family of models and should be applicable to most local LLMs.

# 4. QLoRA Fine-Tuning

Traditional fine-tuning methods load all parameters of the model and perform optimization computations on all parameters during training. However, these traditional methods require significant amounts of memory and compute resources. LoRA is a fine-tuning method that freezes a pre-trained model's weights and optimizes a subset of parameters from each layer of the model's transformer architecture. By only optimizing a subset of parameters, LoRA fine-tuning requires significantly less memory and compute resources during training [16]. A variant of LoRA, called QLoRA, was used during the fine-tuning process detailed in this paper. QLoRA allows quantization of the pre-trained model to further reduce memory usage by encoding parameters using a 4-bit NormalFloat type [17]. The rationale

behind LoRA and QLoRA fine-tuning methods is to fine-tune in limited compute or limited budget environments. Due to its efficiency, QLoRA fine-tuning can be performed on a single GPU such as an Nvidia A100 80GB GPU or an Nvidia RTX A6000 48GB GPU, depending on the selected model and training parameters.

Despite the resource usage benefits, LoRA and QLoRA fine-tuning do not match traditional fine-tuning methods in performance. Both fine-tuning methods result in models that perform worse than full fine-tuned models in the target domain of the training data. However, LoRA and QLoRA fine-tuning presents a much lower risk of negatively impacting the model's core capabilities that are outside the target domain of the training data, such as document summarization and generalized question / answer interactions [18]. For use in the Urban Air Mobility project, this tradeoff is more optimal since the model is expected to perform other tasks outside of Cypher query generation such as analysis.

# 5. Training Data

Our team generated the training data by first manually creating three sets of 50 natural language questions (150 examples total) paired with corresponding Cypher queries, each set focused on content specific to a unique domain. Figure 4, seen below, shows a few question / Cypher pairs extracted from one of the training sets that focuses on querying an Airbnb graph. This training set is one of two that represents a use-case outside of UAM since we intend for this model to be extensible to other domains. The original, manually generated, training data was then expanded

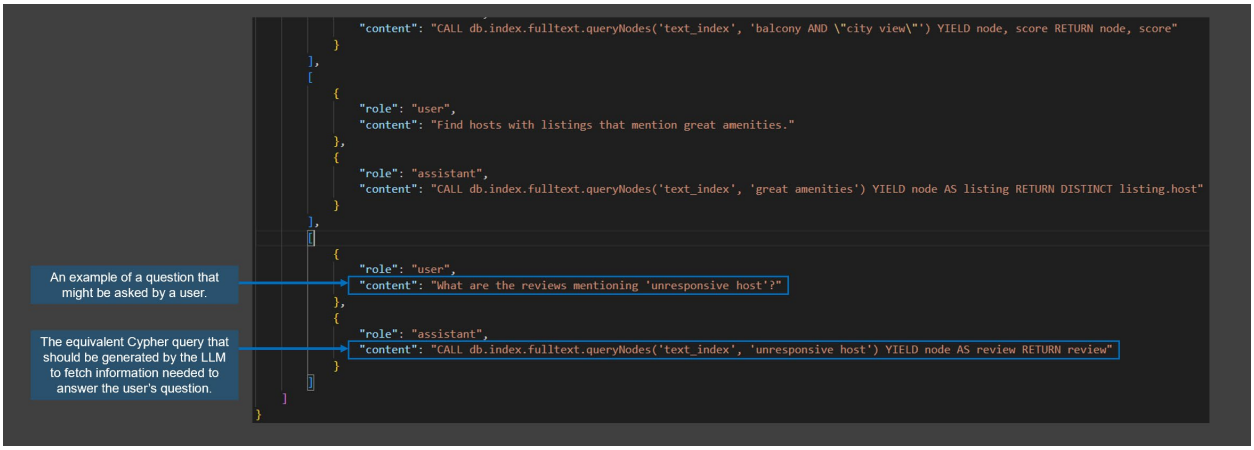



*Figure 4. A snippet from the JSON file containing question / Cypher pairs for the Airbnb dataset.*

from 150 initial examples to a total of 352 examples using another, significantly larger, commercial LLM. The commercial LLM was instructed to generate additional training sets following the structure established in the manually generated examples. Training data generated by the LLM was validated for semantic accuracy before being stored in each dataset's corresponding JSON file for processing into the final training set in a subsequent step.

The training dataset preparation for fine-tuning consists of a pipeline that aggregates information from each dataset's JSON file to form training set entries as seen in the flowchart in Figure 5. Each training set entry is

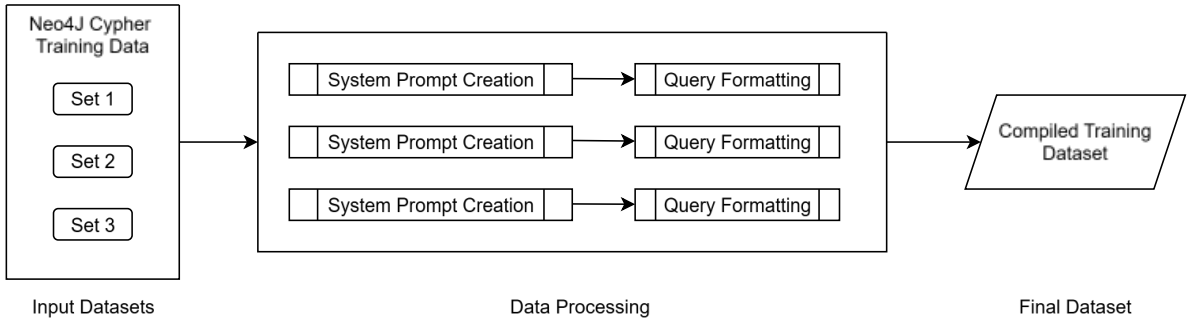


*Figure 5. A flowchart diagram of the training dataset generation pipeline.*

self-contained, consisting of the system prompt, which contains the instructions and database schema information, and the English question / equivalent Cypher query pair. A training set entry is generated for each of the 352 examples using the chat format expected by the model being fine-tuned, these examples are stored in a single JSON Lines (JSONL) file, an example of one of these entries can be seen in the below figure.

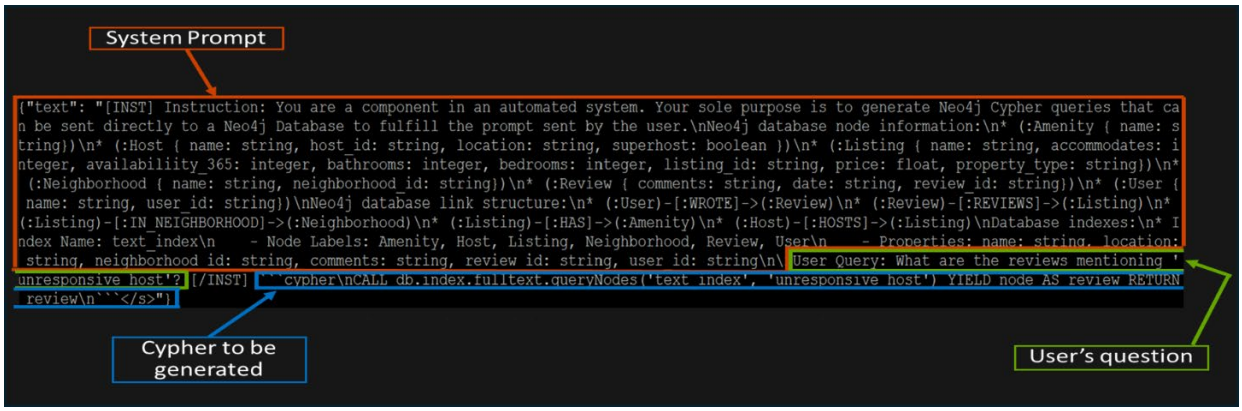



*Figure 6. A training set entry containing the system prompt and question / Cypher pair from the JSONL training data file.*

All datasets contained a mix of basic Cypher queries, fulltext search operations, and GDS library algorithms. Across all domains, the training datasets provide examples to set up the model's foundational Cypher knowledge with basic queries using traditional graph database interactions including MATCH, WHERE, WITH, ORDER BY, and LIMIT statements. In addition, these datasets teach the system how to perform fulltext search and use a variety of algorithms from Neo4j's GDS library including but not limited to Common Neighbors, Adamic-Adar Index, and Preferential Attachment.

Our primary training dataset contains Urban Air Mobility (UAM) relevant elements. Though it focuses on UAM knowledge graph data similar to that found in our AAM graph, it uses an earlier schema version that differs from the current production AAM schema. The reason these data differ from our current production data is our team's work originally focused around UAM elements before being expanded to AAM. Thanks to the extensible structure of the system prompt used to tune the model, this will be a non-issue going forward. The following figure illustrates a basic query from this dataset, showing how natural language prompts are translated to Cypher queries.

```
{
"role": "user",
"content": "Display 25 Requirements associated with the lowest rml values."
},
{
"role": "assistant",
"content": "MATCH (n:`Roadmap Requirement`) WITH n ORDER BY (n.rml) ASC LIMIT 25
RETURN n"
}
```

*Figure 7. Example of UAM training dataset showing one entry with the plain English question and the next entry with the equivalent Cypher translation.*

The two additional datasets were generated using example datasets provided by Neo4j. The second dataset makes use of the "Northwind" example dataset, which closely mirrors a traditional retail system [19]. The third dataset follows the "Airbnb listings" example dataset, which is a graph containing information on Airbnb listings, hosts, and reviews, providing examples in yet another domain context [19].

The complete training set is comprised of 352 total examples across all three domains. Dataset 1 (UAM) contained 123 examples (35%), Dataset 2 (retail) contained 141 examples (40%), and Dataset 3 (Airbnb) contained 88 examples (25%). With training datasets encompassing different domains, we standardized schema information provided to the model for extensibility. The system prompt format is identical for all training datasets. The prompt establishes the model as a Neo4j Cypher query generator. Information is then populated into the system prompt that is unique to each dataset, such as the node types, link types, and indexes found within the graph. Data types for properties are explicitly listed to enable the system to understand and handle type constraints across different schemas. Contextual information passed through the schema declarations are also all formatted identically. This consistency helps the model understand the schema and data regardless of the domain. For reference, the following structure in Figure 8 illustrates the consistent format used for schema declarations across all training domains.

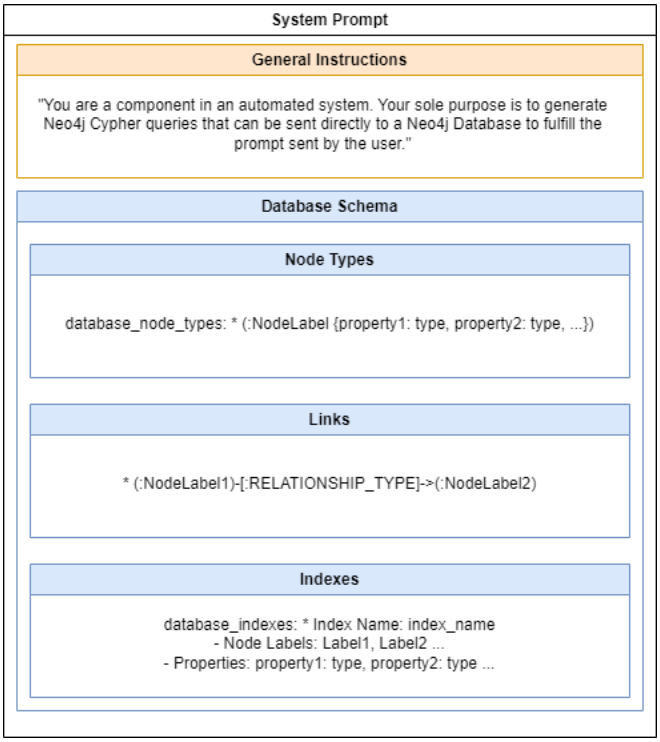



*Figure 8. Diagram of the system prompt.*

By maintaining consistent schema declaration formats the model interprets graph database structure rather than learning domain-specific patterns. This design enables the model's cross-domain flexibility without requiring retraining.

# 6. Training Pipeline

The training pipeline is designed to work on a local High-Performance Computing (HPC) system equipped with multiple GPUs. The process involves a series of steps: training data generation, training, LoRA merge, and quantization. See the previous section for the details surrounding the training dataset generation, this section will focus on the three steps following the creation of the training data. The training step utilizes the Hugging Face Transformers and Parameter-Efficient Fine-Tuning (PEFT) Python libraries. The training script starts by reading in a TOML (Tom's Obvious, Minimal Language) configuration file which contains model parameters, LoRA parameters, and training parameters displayed in Figures 9 and Table 1. Then the model to be trained is loaded using the

| Training Configuration | |
|---|---|
| Model | |
| Model | Path to LLM that will be trained |
| LoRA or QLoRA mode | An option to select LoRA or QLoRA finetuning |
| LoRA Target Modules | Modules to train during LoRA or QLoRA finetuning |
| Modules to Save | Modules to save when writing LoRA adapter to disk |
| LoRA Parameters | |
| Rank | LoRA Rank value |
| Alpha | LoRA Alpha value |
| Dropout | LoRA Dropout value |
| Data And Tokenizer Parameters | |
| Training Dataset | Path to training dataset JSONL |
| Validation Dataset | Path to validation dataset JSONL |
| Sample Length | Size for each training sample |
| Sample Padding Side | Which side to pad training samples if they do not meet the size requirement |
| Add Tokens | List of tokens to add to tokenizer |
| Add Special Tokens | List of special tokens to add to tokenizer |
| Override Special Tokens | Overrides for special tokens such as padding token, end token, etc. |
| Training Parameters | |
| Seed | Seed value to use for training to ensure results are reproducable |
| Epochs | Number of epochs to run training for |
| Batch Size | Size of each training sample batch |
| Learning Rate | Learning rate for training |
| Gradient Accumulation Steps | Number of gradient accumulation steps |
| Loss Log File Path | File to log loss values to |
| Loss Plot File Path | File to write loss value plot |

*Figure 9. Diagram of the training parameters in the training configuration.*

*Table 1. Training parameters used for the deployed Mistral Nemo Instruct 2407 model.*

| Parameter Name | Value |
|---|---|
| GPU | 2x Nvidia A100 80GB |
| Model | MistralAI Mistral Nemo Instruct 2407 |
| Training Mode | QLoRA |
| LoRA Target Modules | “all-linear” |
| Modules to Save | “lm_head”, “embed_tokens” |
| LoRA Rank | 128 |
| LoRA Alpha | 64 |
| LoRA Dropout | 0.1 |
| Sample Length | 1024 Tokens |
| Padding Side | Left |
| Added Tokens | None |
| Added Special Tokens | None |
| Overwritten Special Tokens | Pad Token: “[PAD]” |
| Seed | 25519 |
| Epochs | 5.0 |
| Batch Size | 19 |
| Learning Rate | 2e-5 |
| Gradient Accumulation Steps | 1 |

“get_peft_model()” method from the PEFT library and parameters are then passed to the Transformers “Trainer()” class to initiate the training session. Both LoRA and QLoRA fine-tuning are supported by this pipeline, however full fine-tuning is not. During the training process, the training and validation loss are logged and plotted for inspection. The loss plot for the model we are using in production can be seen in Figure 11. Multiple copies of the LoRA adapters are saved throughout the training process called “checkpoints”. These checkpoints are snapshots of the model’s state at specific points throughout training. They can be used to resume training in case the training stops unexpectedly, or they can be loaded and used for inference to test the model at different points in the training process. After the LoRA adapters are trained, the adapter from the last checkpoint is selected and is merged back into the base model. The model is then quantized, and this process differs depending on the inference engine chosen. Llama.cpp requires the model to be converted to GGUF (GGML Unified Format) then quantized using their quantization program. vLLM requires the use of a Python library called LLM Compressor for quantization. This divergence in the pipeline can be seen at

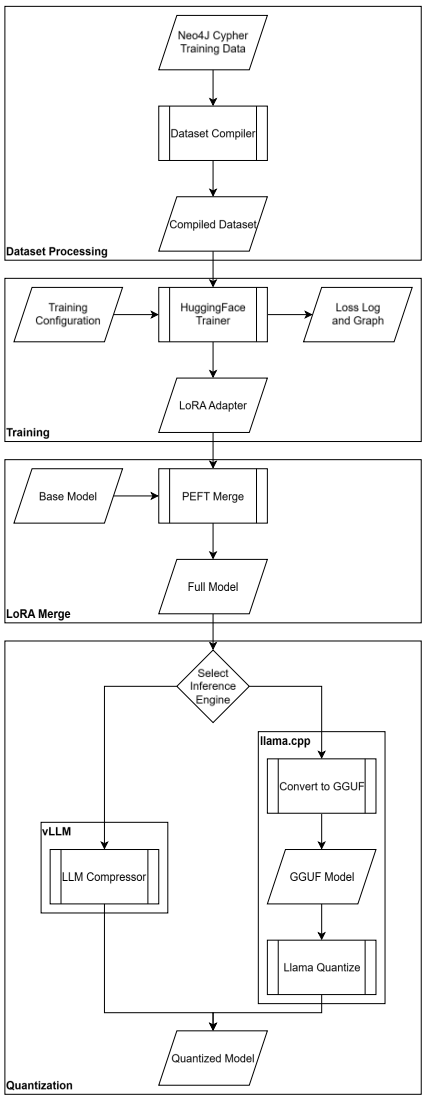



*Figure 10. A flowchart diagram of the training pipeline*

the bottom of Figure 10. The reason quantization is performed is to save memory resources when hosting the model, enabling the model to run on mid-range GPUs accessible to the everyday consumer.

The training process by default uses 2x Nvidia A100 80GB GPUs on the HPC system. After completing the training pipeline, the models are then quantized and hosted using llama.cpp on a system with an Nvidia RTX A6000 48GB GPU. The current model trained and deployed is a fine-tuned version of Mistral Nemo Instruct 2407. After training, the model was converted to the GGUF format and quantized using Q8_0 quantization from llama.cpp.

# 7. Validation

The training process records loss data and produces a graph which displays training loss and evaluation (validation) loss throughout the training session. Evaluation loss is computed using a dedicated evaluation JSONL file which was generated in the same manner and format as the training data. This evaluation set contains examples not included in the training data which the model has never seen before and is used to calculate an evaluation loss metric periodically throughout the training run. The loss graph below provides an overview on the effectiveness of the training and provides a general sense as to if the model is overfitting or underfitting.

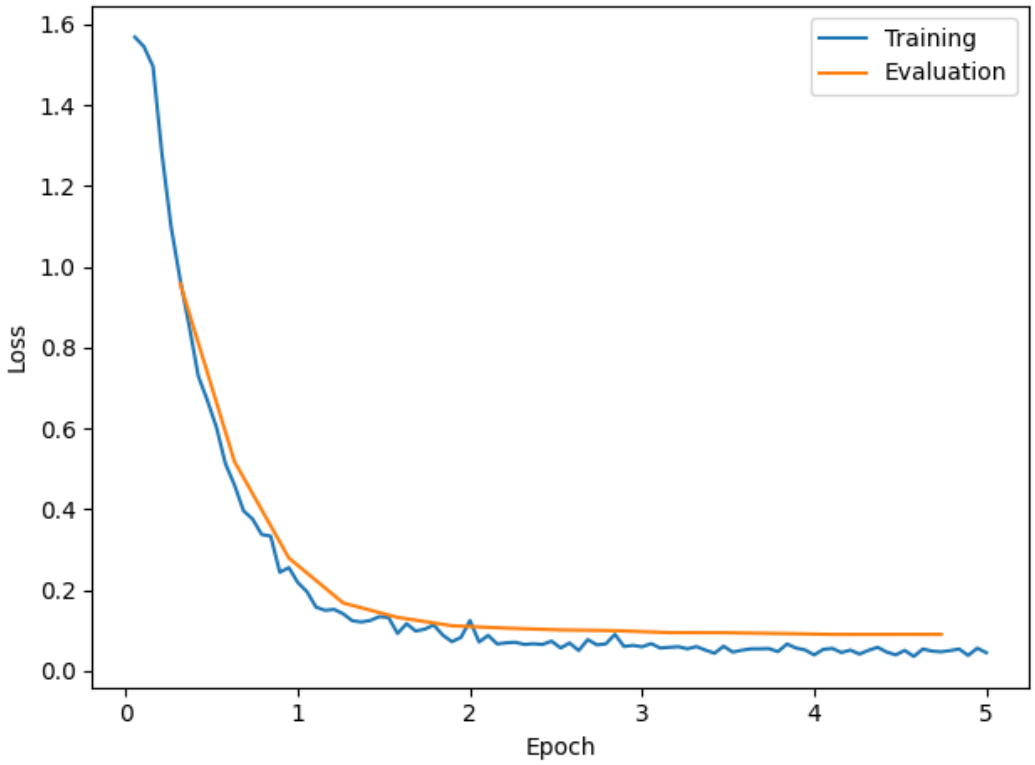



*Figure 11. Deployed Mistral Nemo Instruct 2407 training loss plot.*

An additional suite of automated tests is also performed on the resulting model to ensure the model can correctly generate complex Neo4j Cypher queries. This validation script, written in Python, requires three inputs: a test case configuration file, model-specific configuration files, and the path to the model being tested. This testing architecture enables evaluation dataset adjustment without requiring code modifications.

Testing datasets are organized across multiple CSV files, each corresponding to a specific domain. The script currently evaluates performance across four domains. There are a total of 69 evaluation test case pairs, with the UAM evaluation dataset having 39 test cases targeting our primary training domain. The Retail evaluation dataset contains seven evaluation pairs aligned with dataset 2. The Airbnb evaluation dataset has nine pairs corresponding to dataset 3. To assess the model's extensibility to new domains, the final evaluation dataset is a domain the model has never seen before. Evaluation dataset 4 consists of 14 sets using Neo4j's Stack Overflow example graph, which contains data reflecting tags, and question / answer data retrieved from the Stack Overflow website [19].

- UAM dataset (39 evaluation pairs) - corresponding to the primary training domain
- Retail dataset (seven evaluation pairs) - corresponding to training dataset 2
- Airbnb dataset (nine evaluation pairs) - corresponding to training dataset 3
- Stack Overflow dataset (14 evaluation pairs) - a domain the model has not been trained on, meant for generalization testing

Each test case is comprised of a natural language question paired with one or more acceptable Cypher query responses. This multi-response format recognizes that multiple syntactically different queries can produce equivalent results from the graph database, as demonstrated in the following examples:

**Simple query validation:**
Prompt: "Provide a list of all system actors"
Acceptable responses: MATCH (?:Actor) RETURN ? or MATCH (?:Actor) RETURN ?.name

**Complex query validation:**
Prompt: "Provide a list of 3 system actors"
Acceptable responses:
MATCH (?:Actor) RETURN ? LIMIT 3
MATCH (?:Actor) WITH ? ORDER BY ?.allocations DESC LIMIT 3 RETURN ?
MATCH (?:Actor) WITH ? ORDER BY ?.name DESC LIMIT 3 RETURN ?.name, ?.text

This multi-variant validation structure is enabled through special wildcard characters that represent flexible text sequences. These wildcards are employed when the expected query structure is known but textual variations may occur, such as different variable names or arbitrary formatting differences like spacing. Supported wildcard characters in test case CSVs are as follows:

- **Asterisk (*):** Represents unlimited alphanumeric characters, accommodating variable name differences
  Valid query 1: MATCH (user:Person) RETURN user
  Valid query 2: MATCH (person:Person) RETURN person
  Pattern: MATCH (*:Person) RETURN *

- **Question mark (?):** Represents a single character, useful for single-letter variable variations
  Valid query 1: MATCH (n:Actor) RETURN n
  Valid query 2: MATCH (a:Actor) RETURN a
  Pattern: MATCH (?:Actor) RETURN ?

- **Caret (^):** Represents optional whitespace or single characters, handling spacing inconsistencies
  Valid query 1: MATCH (n:Actor) WHERE n.name="John"
  Valid query 2: MATCH(n:Actor)WHERE n.name="John"
  Pattern: MATCH^(n:Actor)^WHERE n.name="John"

The validation script evaluates the model by loading the specified model's configuration and processing each test case. Each test case is evaluated by comparing the model's generated output against all acceptable responses for a given prompt. A response is considered correct if it matches any of the acceptable queries, accommodating semantically equivalent but syntactically different query structures. The primary evaluation metric is accuracy, calculated as the percentage of test cases where the model produced an acceptable response:

$$\text{Accuracy} = (\text{Number of Acceptable Responses} / \text{Total Test Cases}) \times 100$$

*Equation 1. Accuracy calculation used within the validation script*

The validation script generates output CSV files containing the prompts, model responses, acceptable responses, and the accuracy score. This structured output enables the tracking of improvement across model versions and failure pattern identification.

# 8. Results

When our team started this work, it was found that fine-tuning was a necessity to allow earlier LLMs to understand how to generate syntactically correct Cypher queries. Looking at an early model, the Instruct version of Mistral V0.1, released September of 2023, was only able to perform the most basic English to Cypher query conversions. This model scored a meager 21.4% when ran through our testing pipeline. Due to its low performance out of the box compared to other models that were also available at the time of testing, we did not attempt to generate a fine-tuned version of this model. Subsequent model iterations provided significant improvements regarding out of the box understanding of the Cypher query language. Mistrallite, a fine-tuned version of Mistral-7B-v0.1 by Amazon, was released in October, 2023 [20], shortly after Mistral V0.1, and boosted Cypher query generation performance significantly compared to the original model. Fine-tuning Mistrallite using our pipeline increased the accuracy from 78.6% to 96.4% and enabled the model to perform complex queries such as link prediction. The instruct versions of Mistral V0.2, Mixtral 8x7B, and Mistral Nemo 12B all provided better out of the

box Cypher generation than their earlier counterparts, and all models, except for Mistral Nemo 12B, saw benefit from the fine-tuning process as seen in the figure below.

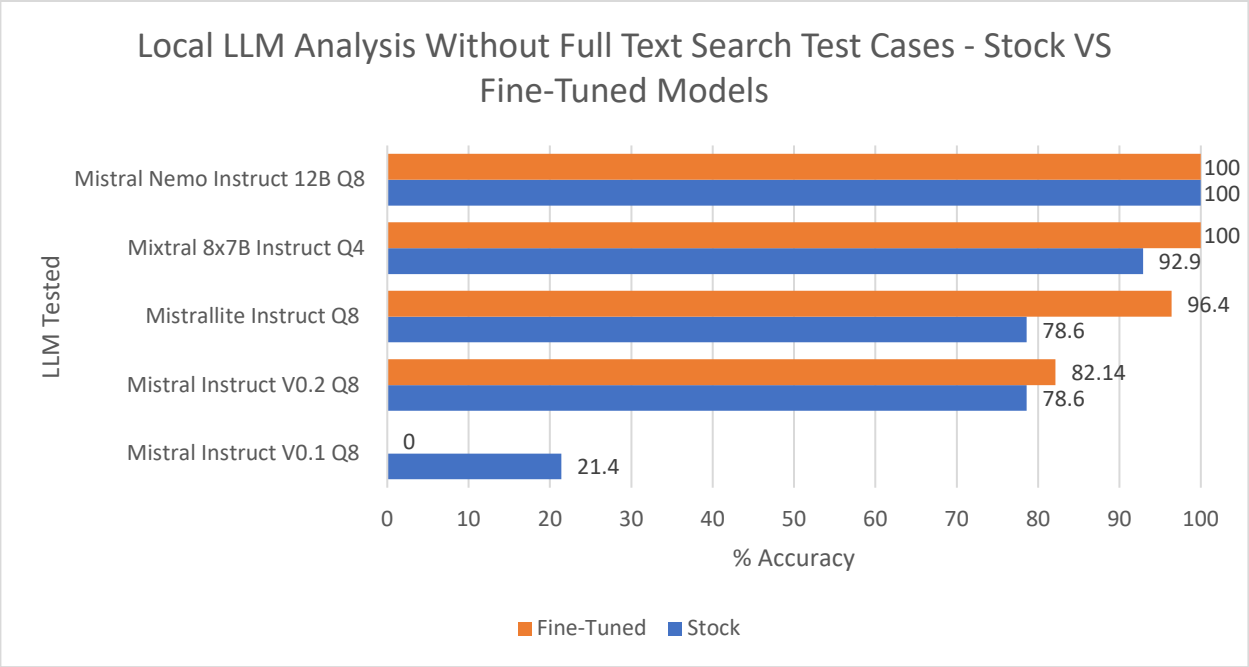



*Figure 12. Testing results from both the stock and fine-tuned Mistral AI models. Note that the "Q#" notation indicates the level of quantization applied to the model being tested. "Q8" indicates 8-bit quantization and "Q4" indicates 4-bit quantization. This test set does not include full text search cases.*

As seen in the figure above, modern LLMs, like Mistral Nemo Instruct 12B, needed no additional tuning to generate the queries provided in our test set. Out of the box, these newer models can be used with one-shot prompting, using a prompt structure like that in Figure 19, to generate Cypher queries that will address a wide number of use-cases without the need for any additional modification to the model itself. However, our team did run into issues when trying to utilize one-shot prompting to enable the Mistral Nemo 12B to perform full-text searches within Neo4j. Mistral Nemo tended to only use "WHERE / CONTAINS" clauses when asked a question that would have been better answered by a full text search. A user might ask "What deliverables focus on interoperability or standards compliance?" and the stock model would generate the following Cypher query depicted in Figure 13.

```
MATCH (d:Deliverable) WHERE d.name CONTAINS 'interoperability' OR d.name CONTAINS
                        'standards compliance' RETURN d
```

*Figure 13. Incorrectly generated Cypher query where full text search capabilities should have been used.*

Though this Cypher statement does in some way return results that are relevant to the user's question, it does not utilize the semantic similarity based full-text search provided by Neo4j, instead the above query only focuses on exact matches which will not capture all deliverables that are relevant to the user's question. The correct form of the Cypher query utilizing full text search can be seen in Figure 14.

```
  CALL db.index.fulltext.queryNodes("full_text_search", '"interoperability" OR
   "standards compliance"') YIELD node, score WHERE node:Deliverable RETURN
          node.name, node.text, node.element, node.component, score
```

*Figure 14. Correctly generated full text search query.*

To enable this capability, our team added full text search examples to the data used for both our training and testing pipelines. Seen in the figure below, adding test cases for the full text search brought the test score down from 100% to 67% with the out of the box model. Fine-tuning using the new training data with full text search examples

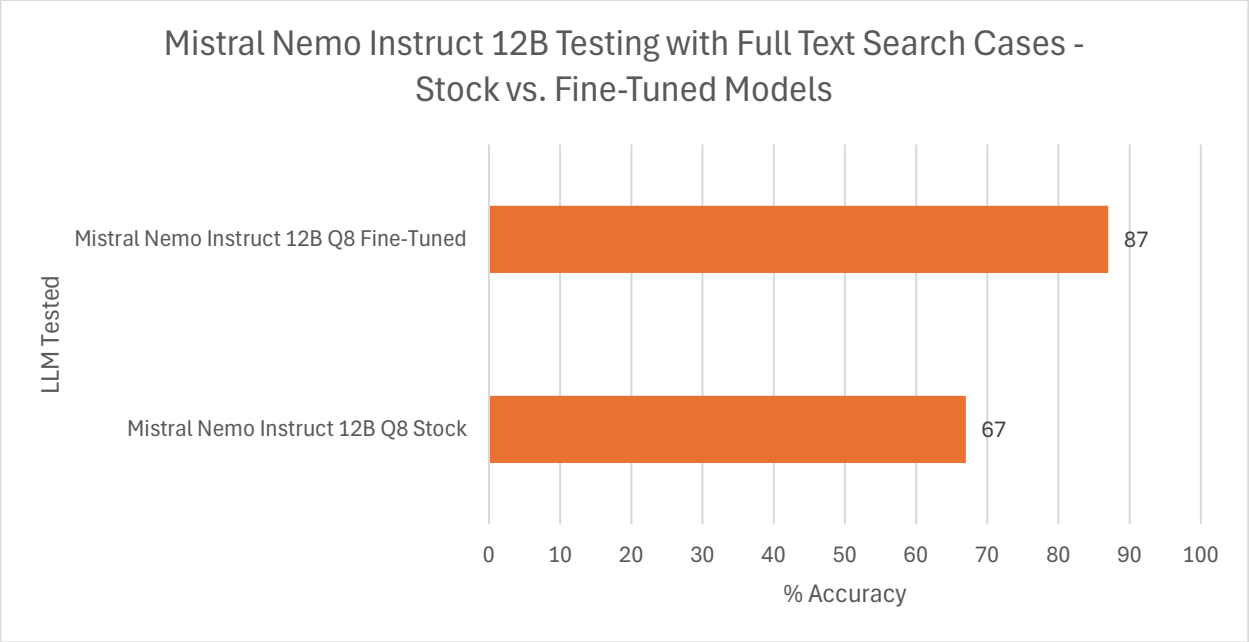



*Figure 15. Testing results from Mistral Nemo Instruct with full text search test cases included in the test set.*

increased the score by 20%. Following the fine-tuning, in most cases Mistral Nemo will use the full text search capability when appropriate, however there are still some instances where it will use "WHERE / CONTAINS" clauses instead of the full text search as noted by the 13% gap in the model's test accuracy. This "confusion" is most likely caused by the fact that the cases where one method should be used over the other are very similar, and even subjective in some situations.

## 9. Deploying the Fine-Tuned Model

Following fine-tuning and validation, the model is then deployed to the AAM Architecture Digital Assistant to serve as an intermediary layer between the user and the information housed within the graph database. The system architecture, which is visualized below in Figure 16, consists of a frontend deployed as a Python Flask web application, where the user can enter their questions in a chatbot interface, the locally deployed LLM server, which runs the fine-tuned model, and the graph database server, which provides access to the interconnected node data and the algorithms to perform analysis.

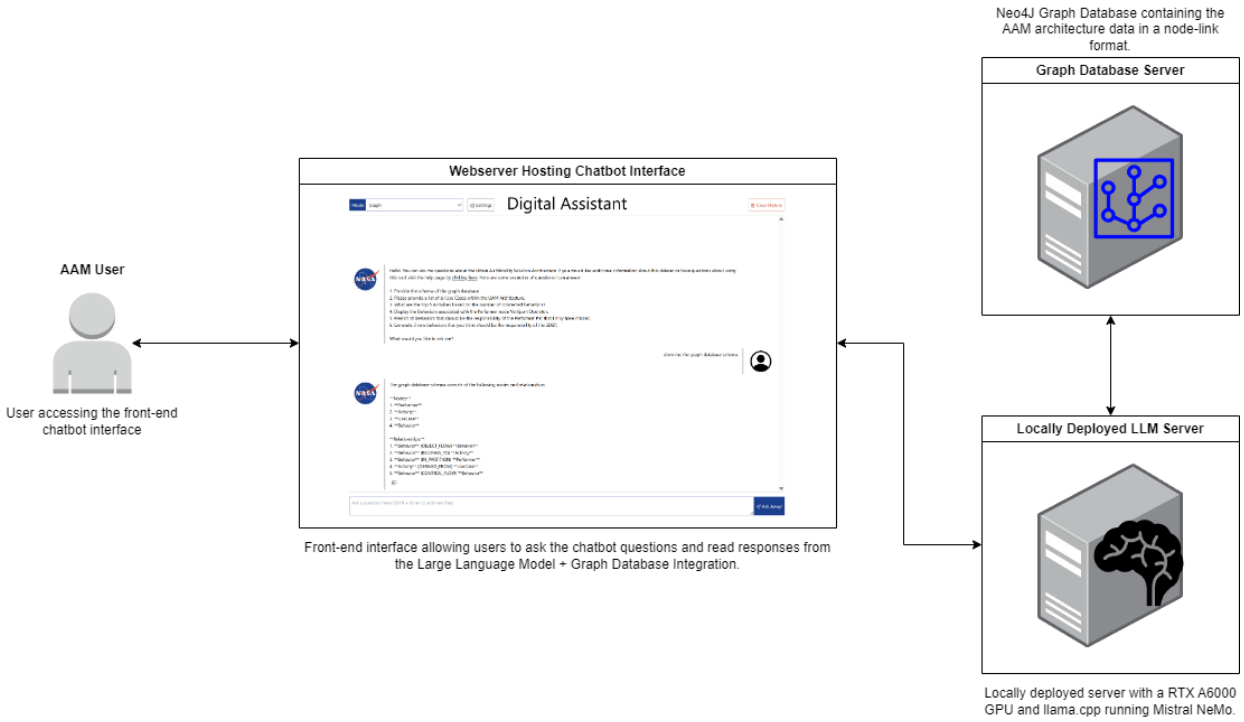



*Figure 16. AAM digital assistant system architecture*

With the LLM serving as an English to Cypher query translator, the user simply asks questions and receives responses in plain English, allowing the user to harness the power of the graph database by simply asking the Digital Assistant questions about the data. The translation process is detailed in Figure 17, and starts with a plain English question from the user, i.e. “How many behaviors are connected to the PIC?”, where the PIC in this case is a Performer node in the AAM architecture known as the Pilot in Command. The system then forwards this question to the LLM which converts the question into an equivalent Cypher query that the graph database understands. This Cypher query is sent to the graph database, executed, and then relevant information is returned to the LLM. At this point, the information returned from the database is in a “JSON-like” format. To get a final, human readable, response, the “JSON-like” list of data is passed back into the LLM and is then converted into a plain English response. This response is displayed on the Digital Assistant interface for the user to read as seen in Figure 18.

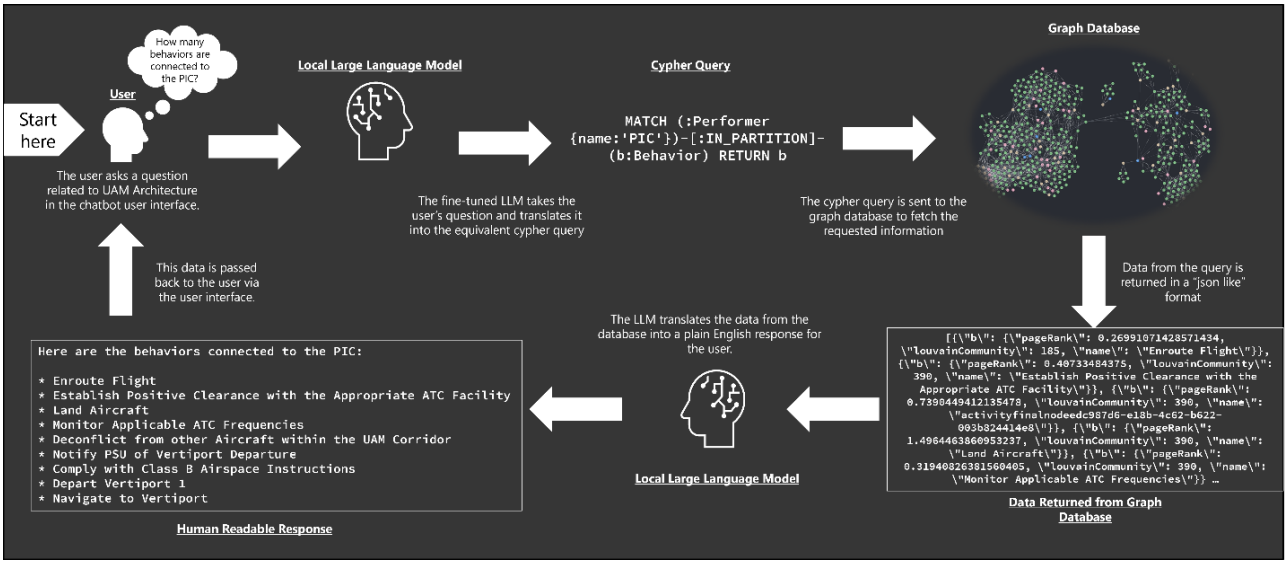



*Figure 17. LLM translation process, from English to Cypher and then back to English*

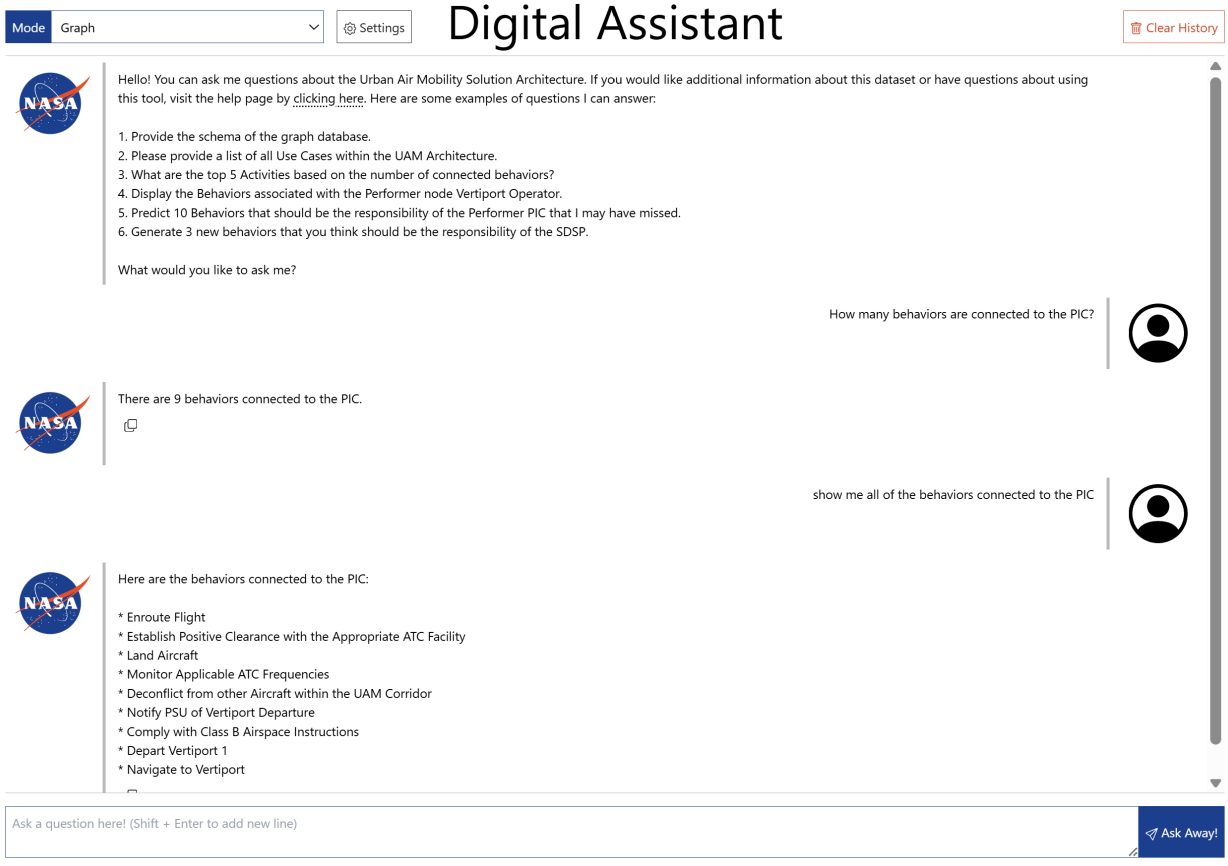



*Figure 18. AAM Digital Assistant user interface*

To make the LLM perform these translations reliably, significant prompt engineering had to occur, specifically with regards to the English to Cypher Query conversions. The specific format of the prompt was developed as part of the fine-tuning process to make the model extensible to use-cases beyond the niche field of AAM. For additional details surrounding this format, please see the "Training Data" section of this paper. Since this custom prompt format was consistently used as part of the fine-tuning process, the resulting model understands the custom prompt format and can be adapted to other graph database schemas beyond the scope of AAM if the required information is provided in the prompt. Figure 19 breaks down the prompt structure into four different sub-components.

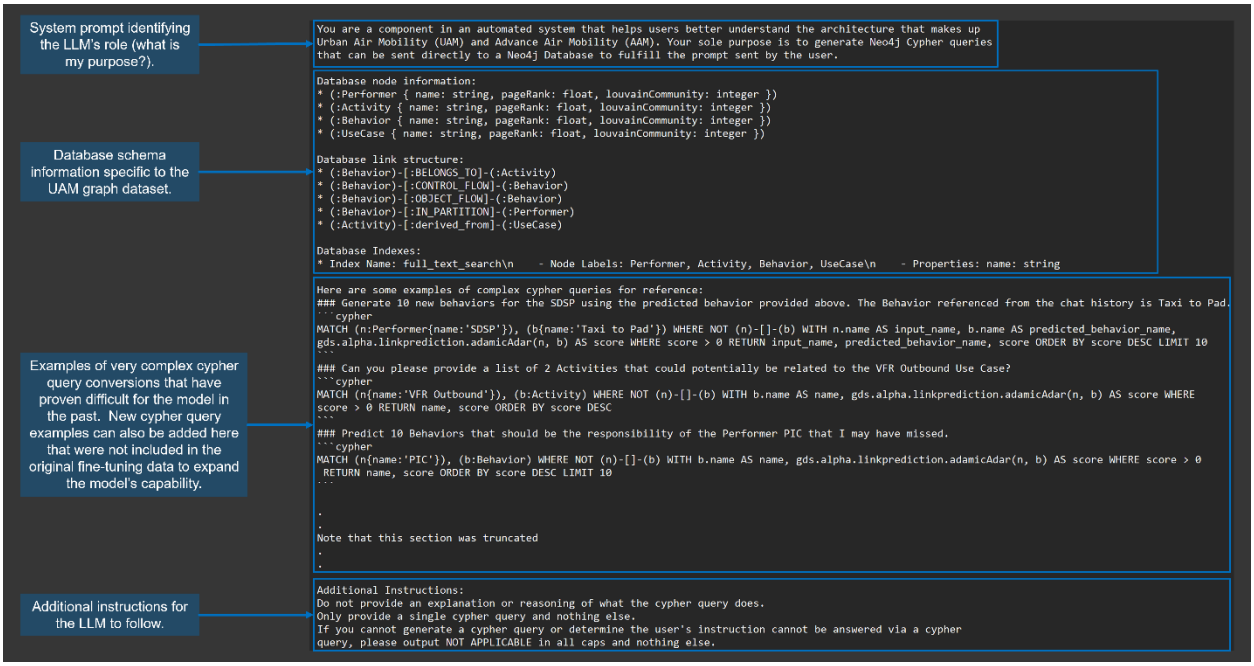



*Figure 19. Large language model prompt for English to Cypher query conversions.*

The first part of the prompt is commonly referred to as the "system prompt", this text defines the purpose of the LLM and what its role is in the larger system. The second section defines the database schema, including the node types and associated properties, and it also defines the expected link types between the different nodes. This section also allows for the definition of database indexes. An index is a copy of specified primary data in a Neo4j database, such as nodes, relationships, or properties. In the case of Figure 19, the index is used to store fields that can be used to perform a full text search in Neo4j. The third section provides examples of complex Cypher queries that were found to be problematic even for the fine-tuned version of the model. These included things like calling on functionality within the GDS toolkit and performing GDS operations using information from questions previously asked by the user and using the full text search functionality within Neo4j. Using this section, it is also possible to add examples to enable new functionality within the LLM. For example, if you were trying to add the ability to find the shortest path from one node to another using the A-Star algorithm, you would add an example similar to the below pseudo query. Though depending on the LLM utilized, additional fine-tuning may be required before the

```
### Show me the shortest path from "node_a" to "node_b"
```
Cypher
MATCH (source:nodeType{name: 'node A'}), (target:nodeType {name: 'node_b'})
CALL gds.shortestPath.astar.stream('myGraph', {
sourceNode: source,
targetNode: target,
latitudeProperty: 'latitude',
longitudeProperty: 'longitude',
```
```

```
relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
index,
gds.util.asNode(sourceNode).name AS sourceNodeName,
gds.util.asNode(targetNode).name AS targetNodeName,
totalCost,
[nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
costs,
nodes(path) as path
ORDER BY index
```

*Figure 20. Pseudo code showing a potential case where shortest path analysis could be performed using the A-Star algorithm.*

model can consistently use such a query correctly. The final section provides some additional instructions to the LLM, like what to do if the user's question is not a good candidate to be answered using the graph database and to only return the relevant query and no additional explanation surrounding what it does.

The prompt for the step where the LLM converts the "JSON-like" response from the graph database to plain English requires only a small instruction-set displayed in Figure 21. There is only one section in this prompt which indicates how the model should translate the structured data into a response that can be displayed back to the user.

```
You are an AI assistant that is part of an automated system with Neo4j integration that helps generate text
to form nice and human understandable answers based on the information in a JSON-like response from the database.
The latest prompt contains the information, and you need to generate a human readable response based on the
given information. Make it sound like the information is coming from you, but don't add any information.
If the database query does not return anything and there is not enough information in the chat history, tell
the user you do not have enough information to answer their question.
If multiple database items are returned, try to organize them into a numbered list. Do not add any additional
information that is not explicitly provided in the latest prompt.
Do not expand ANY acronyms. Acronyms are automatically expanded by the automated system.
Note that the rml_sum field is NOT a monetary value, do not put dollar signs in front of it
when responding. Avoid extraneous repetition in your response.
```

*Figure 21. Large language model prompt for "JSON-like" data to plain English conversion.*

## 10. Conclusion

QLoRA fine-tuning in combination with a well-structured prompt provides a feasible means for adding new capability to a local LLM using readily available consumer hardware. The fine-tuning and prompt engineering methodology mentioned here have the potential to be applied to other niche, novel, or obscure, languages such as Graph QL, Mongo DB QL, SysML 2.0, and UML to name a few. Our team also believes that this work may be applied to even smaller models, such as Google's Gemma 2 2B, to enable fast translation capabilities using lower end GPUs or even CPUs.

The model our team created using QLoRA fine-tuning aided Systems Engineers in navigating the complex AAM ontology by allowing them to "speak" to the data in plain English and enabled them to perform analysis using the graph data science capabilities within Neo4j that they otherwise would not have had access to through traditional System Engineering tools. Our Systems Engineers have reported that this system has increased their speed of requirement analysis by 7x, additionally, they also reported requirements and rationale generated by this system are comparable to those created by human users. The extensibility of the model also proved itself by allowing our team to quickly pivot to an updated ontology, with a significantly altered node-link structure, without the need to perform

an additional fine-tune of the model. Finally, since our team is using a locally deployed, quantized, Mistral Nemo model, we are able to run this model on a shared local server, allowing for significant cost savings to our organization and the assurance that all sensitive data will only be seen by authorized personnel.

# Nomenclature

*AAM* = Advanced Air Mobility
*AMP* = Air Mobility Pathfinder
*CSV* = Comma-separated values
*eVTOL* = Electric Vertical Take-off and Landing
*LLM* = Large Language Model
*LoRA* = Low-Rank Adaptation
*GDMS* = Graph Database Management System
*GDS* = Graph Data Science
*GGUF* = GGML Unified Format *(note, there is some ambiguity around the correct expansion of this acronym, this is the most likely expansion based off our research.)*
*GPU* = Graphics Processing Unit
*HPC* = High-Performance Computing
*JSON* = JavaScript Object Notation
*JSONL* = JSON Lines
*MBSE* = Model Based Systems Engineering
*OCARI* = Operational Concepts, Architecture, and Requirements Integration
PEFT = Parameter-Efficient Fine-Tuning
*QLoRA* = Quantized Low-Rank Adaptation
*SQL* = Structured Query Language
*TOML* = Tom's Obvious, Minimal Language
*UAM* = Urban Air Mobility

# References


[1] Levitt, I., Phojanamongkolkij, N., Horn, A., & Witzberger, K. (2023). UAM Airspace Research Roadmap Rev. 2.0. NASA-TM-20230002647

[2] Phojanamongkolkij, N., Levitt, I. M., & Barnes, P. D. Overview of Model-Based Systems Engineering Efforts to Evolve the Airspace Research Roadmap. In AIAA AVIATION 2022.

[3] VanGundy, B., Phojanamongkolkij, N., Polavarapu, R., Brown, B., Bonner, J. (2024, July). Requirement Discovery Using Embedded Knowledge Graph with ChatGPT. In INCOSE International Symposium: Volume 34, Issue 1.

[4] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008(10), P10008. Doi: 10.1088/1742-5468/2008/10/P10008

[5] Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, 30(1-7), 107-117. Doi: 10.1016/S0169-7552(98)00110-X

[6] Freeman, L. C. (1977). A set of measures of centrality based on betweenness. Sociometry, 40(1), 35-41. doi: 10.2307/3033543

[7] Adamic, L. A., & Adar, E. (2003). Friends and neighbors on the Web. Social Networks, 25(3), 211-230. doi: 10.1016/S0378-8733(03)00009-1

[8] Neo4j. (2025). Fulltext search in Neo4j. Neo4j Knowledge Base. Retrieved from https://neo4j.com/developer/kb/fulltext-search-in-neo4j/

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

[10] OpenAI. (n.d.). OpenAI - fine-tuning models. OpenAI Platform Documentation - Model Optimization. https://platform.openai.com/docs/guides/model-optimization

[11] Anthropic. (n.d.). Claude. Claude AI by Anthropic. https://claude.ai/

[12] xAI. (n.d.). *Grok*. Grok AI. https://grok.com/

[13] llama.cpp - LLM inference in C/C++, Software Package, Ver. b6713, https://github.com/ggml-org/llama.cpp

[14] vLLM - fast and easy-to-use library for LLM inference and serving, Software Package, Ver. 0.11.0, https://github.com/vllm-project/vllm

[15] Mistral AI. (n.d.). *Mistral Nemo*. Mistral AI. https://mistral.ai/news/mistral-nemo

[16] Biderman, D., Portes, J., Ortiz, J. J. G., Paul, M., Greengard, P., Jennings, C., ... & Cunningham, J. P. (2024). Lora learns less and forgets less. arXiv preprint arXiv:2405.09673.

[17] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. Advances in neural information processing systems, 36, 10088-10115.

[18] Shuttleworth, R., Andreas, J., Torralba, A., & Sharma, P. (2024). Lora vs full fine-tuning: An illusion of equivalence. arXiv preprint arXiv:2410.21228.
[19] Neo4j. "Example Datasets." Getting Started – Appendix: Example Data, Neo4j, 2025, https://neo4j.com/docs/getting-started/appendix/example-data/
[20] Hugging Face. (n.d.). *Amazon/mistrallite · hugging face*. amazon/MistralLite · Hugging Face. https://huggingface.co/amazon/MistralLite