AI Chatbot – LLM for Creating Draw.io Editing

Chaimaa El Argoubi

SS 2025

1 Abstract

In AI research, software engineering, and teaching, creating high-quality diagrams—such as deep learning architectures, UML diagrams, flowcharts, and use case models—takes a lot of work but is crucial. Though they have demonstrated potential in reasoning and code generation, Large Language Models (LLMs) such as GPT, Qwen, LLaMA, Gemini, and DeepSeek have trouble producing legitimate, structured **Draw.io diagrams**, particularly when handling intricate revisions, incremental updates, or domain-specific layouts.

In this project, we present an intelligent chatbot that leverages LLMs to generate, edit, and explain `.drawio` diagrams from natural language instructions. Our solution enables both educators and students to rapidly prototype or modify diagrams without requiring manual XML manipulation, saving time and improving accessibility to visual knowledge.

We use a specially constructed high-quality dataset of around 720 samples to investigate and assess three methods: retrieval-augmented generation (RAG), few-shot prompting, and fine-tuning. We demonstrate that, in both qualitative and quantitative evaluations, optimizing Qwen outperforms prompt- and RAG-based approaches in terms of output validity, structural consistency, and instructional precision. This work demonstrates the practical feasibility of diagram-centric LLM systems and opens the door to scalable, explainable, and real-time diagram generation in educational and professional settings.

2 Introduction

Diagrams are essential for understanding and communication in many fields, including data science, software engineering, and education. A lot of people utilize tools like draw.io to make system architectures, neural networks, flowcharts, and UML diagrams. However, creating these diagrams by hand takes a lot of time, is prone to mistakes, and necessitates knowledge of the graphical tool as well as the subject. Furthermore, it is not easy to update and interpret already-existing diagrams, particularly when their complexity rises.

State-of-the-art LLMs like GPT-4, Claude, Gemini, DeepSeek, and Mistral have advanced quickly, but they are not well-suited for creating or modifying structured representations like draw.io XML. Current models frequently fail to maintain structural integrity, produce erroneous or incomplete XML, or experience hallucinations. Furthermore, they are unable to dynamically modify already-existing diagrams or provide an understandable explanation of their structure. This emphasizes the necessity of quick engineering and domain-specific fine-tuning, which our solution demonstrates.

To address these gaps, we developed an intelligent chatbot system powered by a fine-tuned LLM that can generate, edit, and explain draw.io diagrams from natural language instructions. Our system goes beyond prompt-based generation by incorporating Retrieval-Augmented Generation (RAG) and custom finetuning, ensuring accurate and well-structured output across various diagram types. This enables users—including students, engineers, and researchers—to automate complex diagram creation with minimal effort.

2.1 Problem Definition

Designing and maintaining Draw.io diagrams by hand is a laborious and time-consuming process, especially in technical and academic settings. In order to depict neural network architectures, data pipelines, UML models, or system operations, professors frequently invest hours in drawing intricate diagrams that they then laboriously update or improve. Comparably, without interactive or dynamically generated versions that correspond with the changing content of lectures or projects, students find it difficult to understand complex diagrams. Despite its potential for automated content creation, Large Language Models (LLMs) such as GPT, LLaMA, Qwen, Gemini, DeepSeek, and Claude have significant drawbacks when it comes to creating organized diagrams:

- **Syntactic fragility**: These models frequently generate erroneous or distorted XML that is incompatible with.drawio, particularly when it comes to intricate diagrams.
- **Inability to edit**: Smaller adjustments, like adding nodes or layers to an already-existing diagram, are frequently handled incorrectly or lead to structural irregularities.
- **Semantic gaps**: Diagrams often have illogical groupings or disconnected components, making their logical flow erroneous.
- **Inconsistency**: Diagrams created from comparable prompts frequently provide inconsistent results, which precludes their application across the curriculum.

These challenges make LLMs unreliable in real-world situations where diagram accuracy, reusability, and interaction are crucial, particularly in software engineering, research, and education. This demonstrates the pressing need for a strong, LLM-based system that can accurately and fine-grainedly generate, edit, and explain.drawio diagrams.

2.2 Objectives

This project aims to build an LLM-powered chatbot that can:

- **Generate draw.io-compatible diagrams** from natural language instructions, covering formats such as flowcharts, neural networks, UML diagrams, and more.
- **Edit existing diagrams** through follow-up instructions (e.g., "add a convolution layer," or "remove the login step").

- **Explain diagram structures and components** in plain language to enhance understandability.

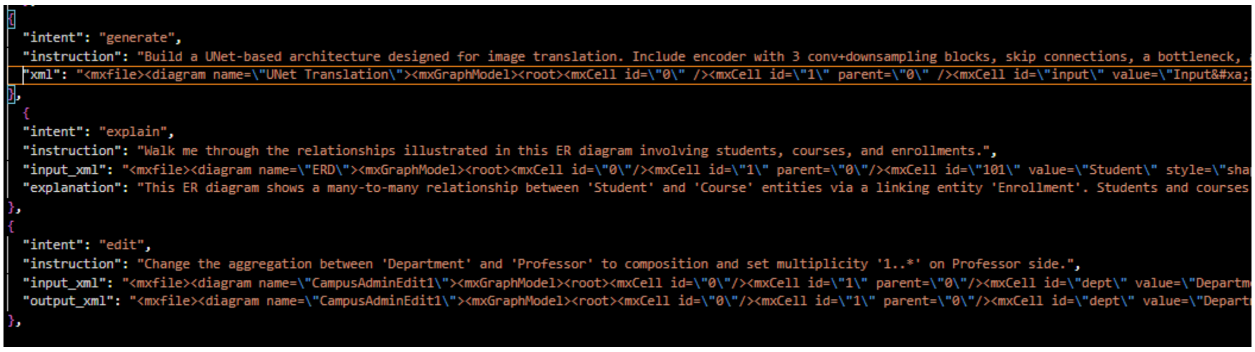
We aim to offer a tool that bridges the gap between textual intent and visual representation, delivering diagrams that are accurate, reusable, and easy to manipulate.

2.3 Scope and Limitations

This project aimed to develop an LLM-powered chatbot capable of generating, editing, and explaining a wide range of `.drawio` diagrams (neural network architectures, UML models, flowcharts, etc.) using natural language instructions. While we succeeded in creating a functional and intelligent system, the following limitations and constraints shaped the scope of our work:

- **Manual Dataset Creation:** There were no publicly available datasets mapping natural language instructions to valid `.drawio` XML. As a result, we had to build the dataset entirely from scratch — including writing instructions, crafting valid XML representations, and manually validating each example. This was extremely time-consuming, particularly in reaching over 719 high-quality samples.
- **No XML Data Augmentation:** XML diagrams are extremely structured, in contrast to image or text databases, and even little alterations might compromise syntactic validity or change the meaning of the diagram. We were forced to rely entirely on hand-crafted, high-quality examples because data augmentation was no longer practicable.
- **Limited Dataset Size vs. Large Models:** Models like Qwen, DeepSeek, or GPT variants are pre-trained on massive corpora. Fine-tuning them effectively on a relatively small, specialized dataset is challenging and prone to overfitting. Careful preprocessing, formatting, and hyperparameter tuning were essential.
- **Containerized Training & Serving:** We were unable to immediately serve or fine-tune the model on local computers due to hardware constraints. GPUs were used to train and deploy the model in a containerized environment. Scalability was made possible, but portability was restricted and complexity was added.
- **Slow Inference During Development:** During system testing, our local application connected to the model container to receive outputs. Testing for real-time interactions, iteration, and development were all slowed down by the latency this configuration introduced.

Despite these limitations, our work demonstrates that fine-tuning LLMs on high-quality diagram data can significantly outperform prompting- and RAG-based methods in structure-aware tasks like `.drawio` generation.



```
{
  "intent": "generate",
  "instruction": "Build a UNet-based architecture designed for image translation. Include encoder with 3 conv+downsampling blocks, skip connections, a bottleneck, 
  "xml": "<mxfile><diagram name=\"UNet Translation\"><mxGraphModel><root><mxCell id=\"0\" /><mxCell id=\"1\" parent=\"0\" /><mxCell id=\"input\" value=\"Input&#xa;
},
  {
  "intent": "explain",
  "instruction": "Walk me through the relationships illustrated in this ER diagram involving students, courses, and enrollments.",
  "input_xml": "<mxfile><diagram name=\"ERD\"><mxGraphModel><root><mxCell id=\"0\"/><mxCell id=\"1\" parent=\"0\"/><mxCell id=\"101\" value=\"Student\" style=\"sha
  "explanation": "This ER diagram shows a many-to-many relationship between 'Student' and 'Course' entities via a linking entity 'Enrollment'. Students and courses
},
{
  "intent": "edit",
  "instruction": "Change the aggregation between 'Department' and 'Professor' to composition and set multiplicity '1..*' on Professor side.",
  "input_xml": "<mxfile><diagram name=\"CampusAdminEdit1\"><mxGraphModel><root><mxCell id=\"0\"/><mxCell id=\"1\" parent=\"0\"/><mxCell id=\"dept\" value=\"Departm
  "output_xml": "<mxfile><diagram name=\"CampusAdminEdit1\"><mxGraphModel><root><mxCell id=\"0\"/><mxCell id=\"1\" parent=\"0\"/><mxCell id=\"dept\" value=\"Depart
},
```

Figure 1: Sample JSON structure showing fields used for each intent type (Generate, Edit, Explain).

3 Dataset Design

3.1 Overview

Our dataset was carefully curated to support three key functionalities of our system: diagram generation, diagram editing, and diagram explanation. Each example is annotated with one of three distinct intents:

- **Generate:** Produce a new `.drawio` XML diagram from a natural language instruction.
- **Edit:** Modify an existing diagram based on a prompt.
- **Explain:** Provide a textual explanation for a given diagram.

3.2 Data Format and Example Structure

Each data entry is stored in JSON format. The dataset follows an intent-specific schema:

- **Generate** samples include: `intent`, `instruction`, and `xml`.
- **Edit** samples include: `intent`, `instruction`, `input_xml`, and `output_xml`.
- **Explain** samples include: `intent`, `instruction`, `input_xml`, and `explanation`.

For each intent, we created unique prompt templates to mimic actual user-assistant dialogues. We were eventually able to use instruction-following behavior to refine LLMs like Qwen thanks to this format.

3.3 Diagram Complexity and Diversity

The dataset covers a broad range of diagram kinds and complexities to guarantee generalization, including:

- Deep learning models such as ResNet, Transformer, U-Net, and GANs
- UML class and activity diagrams

- Flowcharts and system workflows
- Use case and ER diagrams

To demonstrate robustness under different levels of complexity, we included both basic and multi-component diagrams.

3.4 XML Normalization and Validation

Making sure that all XML outputs are syntactically correct and semantically interpretable by `draw.io` was one of the main problems. To deal with this:

- We applied a custom normalization function to standardize malformed or partial XML into a valid `<mxfile>` format.
- We used Python's `xml.etree.ElementTree` to validate well-formedness.
- To avoid rendering problems, root wrappers and default structure were introduced when needed.

3.5 Manual Data Creation and Cleaning

No pre-existing datasets were suitable for our goal. By hand, we produced more than 719 examples, devoting a great deal of time to timely authoring, XML production, and verification. The procedure for cleaning entailed:

1. Stripping invalid characters and removing Markdown artifacts (e.g., ‘‘‘`xml` tags).
2. Applying consistent formatting via regex and XML parsing.
3. Tagging invalid or broken examples for exclusion.

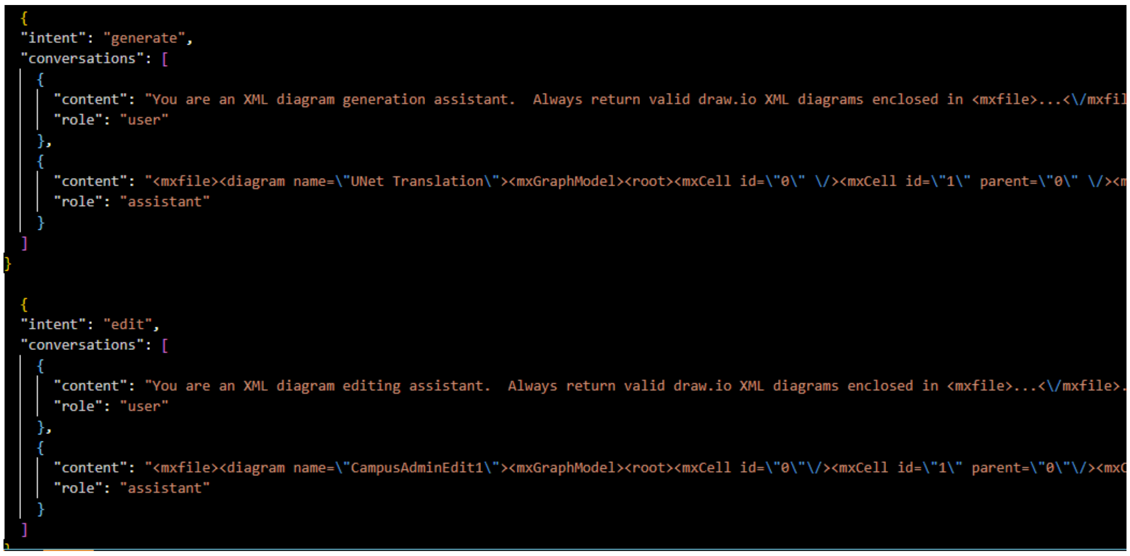
The cleaning and conversion pipeline is illustrated by the following Python functions:

- `normalize_drawio_xml`: Standardizes the XML structure into a valid `<mxfile>` format, ensuring that diagrams conform to draw.io syntax.
- `is_valid_xml`: Parses and validates the XML content to check for structural correctness.
- `clean_dataset`: Loads the raw JSON data, applies normalization, and outputs a cleaned version of the dataset with consistent and valid XML.
- `load_data`: Reads the cleaned dataset and transforms each item into a structured two-turn conversation (user prompt and assistant response) suitable for fine-tuning a language model.

Each conversation entry contains:

- A system-style user instruction that clearly specifies the task and context (e.g., generating, editing, or explaining a diagram).
- A corresponding assistant response in XML or natural language, depending on the intent.

An example of the resulting training conversation format is shown in Figure **??**.



```
{
"intent": "generate",
"conversations": [
  {
    "content": "You are an XML diagram generation assistant.  Always return valid draw.io XML diagrams enclosed in <mxfile>...<\/mxfil
    "role": "user"
  },
  {
    "content": "<mxfile><diagram name=\"UNet Translation\"><mxGraphModel><root><mxCell id=\"0\" \/><mxCell id=\"1\" parent=\"0\" \/><m
    "role": "assistant"
  }
]
}

{
"intent": "edit",
"conversations": [
  {
    "content": "You are an XML diagram editing assistant.  Always return valid draw.io XML diagrams enclosed in <mxfile>...<\/mxfile>.
    "role": "user"
  },
  {
    "content": "<mxfile><diagram name=\"CampusAdminEdit1\"><mxGraphModel><root><mxCell id=\"0\"\/><mxCell id=\"1\" parent=\"0\"\/><mxC
    "role": "assistant"
  }
]
```

Figure 2: Formatted conversation samples after dataset cleaning and conversion. Each entry includes a user prompt and assistant response tailored to the task intent.

3.6 Challenges in Dataset Design

This section of the project posed several real-world engineering challenges:

- **No public datasets:** All examples had to be authored from scratch.
- **No XML data augmentation:** XML's structural sensitivity prevented the application of data augmentation techniques.
- **High-quality demands:** Each sample needed to be thoroughly checked because LLM fine-tuning is susceptible to noise.

4 System Design and Architecture

The system is designed as a full-stack application that enables users to generate, edit, and understand draw.io diagrams through natural language instructions. The architecture integrates a frontend chat interface, a FastAPI backend for request handling, and a fine-tuned large language model (LLM) for diagram synthesis. The major components and their interactions are outlined below.

4.1 High-Level Workflow

The overall workflow follows a simple, user-friendly pattern:

1. User Input: The user types a natural language instruction into a chatbox (e.g., "Create a CNN model with three convolution layers and a softmax output").
2. Backend Processing: The instruction is passed to the FastAPI backend, where it is analyzed and forwarded to the LLM.

3. LLM Inference: A fine-tuned version of the Qwen 2.5 model, optimized using LoRA (Low-Rank Adaptation), processes the prompt and generates or edits draw.io-compatible XML based on template-guided techniques.
4. Diagram Handling: The backend parses, validates, and, if needed, modifies the XML structure to ensure compliance with draw.io formatting rules.
5. Output Delivery: The generated or updated diagram is returned to the frontend and displayed in a visual format.

This loop allows iterative refinement through follow-up prompts, enabling dynamic editing and explanation of diagrams.

4.2 Components Overview

4.2.1 Frontend (React)

- Provides an intuitive chat-based user interface.
- Sends user messages and receives generated diagrams or explanations from the backend.
- Renders the visual output using embedded draw.io viewer components or XML-to-image renderers.

4.2.2 Backend (FastAPI)

- Acts as the intermediary between the frontend and the model.
- Handles API routing, input validation, session management, and XML parsing logic.
- Performs pre- and post-processing of prompts and model outputs (e.g., XML formatting, template merging, element insertion/removal).

4.2.3 Language Model (Qwen 2.5 + LoRA)

- Fine-tuned using LoRA on a dataset consisting of natural language prompts paired with draw.io XML samples.
- Generates well-structured XML guided by predefined templates for different diagram types (e.g., flowcharts, UML, neural networks).
- Supports in-context editing by modifying existing XML structures while preserving diagram logic and layout.

4.2.4 Diagram Engine

- Validates XML outputs to ensure they conform to draw.io standards.
- Uses a parser to selectively update existing diagrams (e.g., add/remove nodes, rename components).
- Handles diagram structure consistency and syntactic integrity.

4.3 Key Design Considerations

- **Template-Guided Generation:** By grounding the model in structural templates, the system ensures that generated XML maintains logical flow and adheres to draw.io schema.
- **Dynamic Editing:** Instead of regenerating entire diagrams, the system supports targeted XML manipulation for efficient updates.
- **Modular Design:** Each system component (UI, API, model, parser) is loosely coupled to allow easy upgrades or replacement.

5 Approaches Tried

5.1 Few-shot with LLaMA

Before fine-tuning a model for diagram generation and editing, we explored few-shot prompting using the LLaMA-3.1 model deployed via a local inference API. This approach aimed to evaluate how well a general-purpose LLM could handle diagram tasks with minimal supervision.

We structured the experiment to test two core capabilities:

- **Diagram generation** from a natural language instruction.
- **Diagram editing** based on an existing draw.io XML and a follow-up instruction.

5.1.1 Pipeline Workflow

We implemented an automated script to:

- Send prompts to the model via the REST API.
- Clean the raw model output by:
  - Removing formatting artifacts.
  - Extracting valid mxfile or mxGraphModel blocks using regex.
  - Decoding any escaped characters.
- Validate the structure using Python's xml.etree.ElementTree.
- Compare against expected outputs using whitespace-insensitive and XML-normalized comparisons.
- Classify results as:
  - **match:** XML matches expected output.
  - **minor_diff:** XML differs only in formatting.
  - **structural_diff:** Logical mismatch in structure or content.

  - **invalid_xml:** Model output was not well-formed XML.

- Generate live draw.io links for quick visual inspection of model outputs.

5.1.2 Observations

- **Syntactic Fragility:** The LLaMA model frequently produced XML with minor syntax errors or malformed tags, especially in editing tasks.
- **Structural Inaccuracy:** While the model sometimes captured basic intent (e.g., "add a layer"), the resulting diagrams often had disconnected elements or illogical flows.
- **Inconsistency:** Outputs varied widely even for prompts with slight variations, indicating low reliability for curriculum-scale reuse.
- **Lack of Context Awareness:** In editing tasks, the model struggled to maintain or adapt the existing structure, often re-generating unrelated diagrams.

5.1.3 Outcome

This experiment confirmed the limitations of general-purpose LLMs when applied directly to diagram-related tasks, even with structured prompting. It highlighted the need for:

- Template-guided control.
- Fine-tuning on domain-specific diagram data.
- XML-aware validation and manipulation tools.

As a result, we moved toward fine-tuning with LoRA on the Qwen 2.5 model, coupled with template-based post-processing, which significantly improved performance in later stages of the project.

5.2 RAG System with Qwen

We used a Retrieval-Augmented Generation (RAG) system built on top of the Qwen 2.5 large language model for one of our primary strategies. This system efficiently manages tasks requiring diagram creation, explanation, and revision by utilizing generative capabilities in conjunction with the retrieval of pertinent materials.
With the specific fields and structure already described in the dataset design section, we used the dataset we prepared in JSON format, which contains three unique intents: *generate*, *edit*, and *explain*.
The technical components of the system are as follows:

- **Embeddings and Vector Store:** Documents were embedded using the sentence-transformers/paraphrase-MiniLM-L6-v2 model. These embeddings were indexed with FAISS for efficient similarity search.
- **Retriever:** FAISS returns the top 5 relevant documents for a given query, enabling the system to ground its generation on semantically similar prior examples.

- **Language Model:** In order to ensure low-latency and controlled environment inference, we used the Qwen 2.5 model, which was accessed via a LangChain `ChatOpenAI` wrapper linked to a custom inference endpoint that our professor provided.
- **Pipeline:** Through LangChain's `RetrievalQA` chain, the retrieval and generation are combined, and Qwen uses the retrieved documents as context to provide outputs that are specific to the user's instructions.
- **Supported Tasks:** The system can generate diagrams from instructions, explain diagram structure in natural language, and perform editing operations on existing diagrams, always producing valid draw.io XML.

This RAG-based system provided a solid baseline and foundation before applying fine-tuning on Qwen with additional training examples, leading to improved performance across all evaluation metrics.

5.3 Finetuning Qwen

To further enhance the model's ability to understand and generate diagram-related intents accurately, we adopted a fine-tuning approach on the Qwen 2.5 language model. This step was essential for customizing the base pretrained model to the particular activities and conversational patterns outlined in our project.

5.3.1 Model and Tokenizer Initialization:

We loaded and optimized the pre-trained Qwen 2.5 model (unsloth/Qwen2.5-Coder-7B-Instruct) using the unsloth library. With optimizations including low-bit quantization and parameter-efficient fine-tuning techniques, the unsloth library is a specialized toolset made to effectively handle big language models. It enables us to effectively manage the resource consumption of large models, such as Qwen 2.5, on the hardware that is currently available. We used 4-bit quantization, which lowers the precision of model weights from 16/32-bit floating-point numbers to 4-bit integers, in order to manage the model's large size (7 billion parameters). This keeps performance close to the original model while significantly reducing memory utilization and speeding up training and inference.
We used Parameter-Efficient Fine-Tuning (PEFT), a sophisticated method for fine-tuning that modifies a limited subset of the model's parameters rather than all seven billion. In particular, we employed a technique called LoRA (Low-Rank Adaptation), which adds trainable low-rank matrices to specific essential layers. In our instance, these levels were the projection layers that handled queries, keys, values, and outputs inside the model's attention method. LoRA significantly reduces the memory and computing load needed for fine-tuning by altering these layers with very few additional parameters.
By using this technique, we can successfully modify the potent Qwen 2.5 model to fit our unique conversational dataset without having to pay exorbitant prices for extensive fine-tuning. Additionally, it preserves the knowledge and generalization capabilities of the base model while enabling quicker experimentation and deployment.

5.3.2 Data Formatting and Tokenization:

As detailed in the data design section, we created and standardized the conversational exchange dataset that we used for fine-tuning. A list of the user-assistant conversation turns is included in each data example.
We used the get_chat_template function from the unsloth library to apply a particular chat template to the data in order to make it compliant with the Qwen 2.5 chat model. By maintaining discussion context and the desired response format, this template formats the talks into an understandable organized prompt.
After that, we used the model's tokenizer to tokenize these structured texts, truncating and padding them to a maximum sequence length of 2048 tokens. This length respects memory constraints while accommodating the majority of discussion scenarios.
The tokenization process outputs input IDs, attention masks, and labels, where labels are a copy of input IDs for causal language modeling. With this configuration, the model may be trained to anticipate the subsequent token in the conversation sequence.

5.3.3 Training Setup:

We used the Hugging Face `Trainer` API to manage the fine-tuning process with the following key hyperparameters and settings:

- **Batch Size:** To stabilize training, 2 samples per device with gradient accumulation over 4 steps are used, which essentially simulates a batch size of 8.
- **Learning Rate:** Set to 5×10^{-5}, balancing convergence speed and training stability.
- **Epochs:** 10 epochs were used to train the model, which allowed for enough data runs to adjust parameters without overfitting.
- **Weight Decay:** To avoid overfitting and regularize model weights, a tiny decay rate of 0.01 was used.
- **Warmup Steps:** For seamless optimization, the learning rate was progressively increased at the beginning of training using 20 warmup steps.
- **Learning Rate Scheduler:** During training, a cosine decay scheduler was used to gradually lower the learning rate.
- **Precision:** To speed up training and lower memory usage without compromising accuracy, mixed precision training with `bf16` was enabled.
- **Optimizer:** To ensure optimal memory utilization and training performance, the `paged_adamw_8bit` optimizer was employed.
- **Gradient Checkpointing:** Enabled to save GPU memory by recomputing intermediate activations during backpropagation.
- **Random Seed:** To guarantee that training results can be replicated, it was fixed to 42.

- **Model Saving:** Checkpoints were saved after each epoch with a limit of 3 to manage storage efficiently.

These configurations ensured efficient, stable, and reproducible fine-tuning of the large Qwen 2.5 model using our conversational dataset.

5.3.4 Significance:

The Qwen 2.5 model was able to effectively internalize and leverage domain-specific knowledge derived from our meticulously created conversational dataset thanks to the fine-tuning technique used in this work. We significantly improved task accuracy, response relevance, and overall output quality by combining Parameter-Efficient Fine-Tuning (PEFT), specifically LoRA, with careful prompt formatting and a representative dataset. This method significantly decreased the memory and processing costs that are usually involved in fine-tuning big language models with billions of parameters. As a result, we were able to efficiently tailor a state-of-the-art language model to our specialized diagram generation and editing use cases, demonstrating a practical pathway for adapting large-scale models in resource-constrained environments without compromising performance.

6 Evaluation Strategy and Results

We used a set of evaluation measures according to each intent's type to evaluate our optimized LLM's performance on various tasks.

6.1 Explain Intent: BLEU and ROUGE

The **Explain** intent involves generating natural language explanations for a given diagram. Since this is a text generation task, we used standard NLP metrics:

- **BLEU (Bilingual Evaluation Understudy):** Compares the generated explanation to a reference explanation to determine the n-gram precision. It is appropriate for structured technical explanations because it rewards precise phrase overlap and penalizes brevity.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** Measures the amount of the reference explanation that is captured by the generated text in order to concentrate on memory. We employ ROUGE-L, which provides a more flexible similarity evaluation by taking into account the longest common subsequence.

These metrics provide a quantitative measure of the model's ability to produce accurate and fluent textual explanations.

6.2 Edit and Generate Intents: Structural Comparison

For the **Edit** and **Generate** intents, the outputs are XML files representing diagrams in `draw.io` format. Since these diagrams are structural (not textual), we used the following metrics:

- **Precision, Recall, and F1-Score:** We compared the expected output element-by-element with the reference XML after tokenizing and normalizing the XML tags and structure. This method assesses how well the model creates or modifies important diagram elements.
  - **Precision:** Measures how many of the generated XML elements are correct.
  - **Recall:** Measures how many of the expected elements were successfully generated.
  - **F1-Score:** Harmonic mean of precision and recall — provides a balanced score especially useful in partial correctness.
- **Graph Edit Distance (GED):** Given that diagrams are graphs, we also used GED to assess the structural similarities between the reference and forecast diagrams. It determines how few graph operations—such as insertions, deletions, and substitutions—are required to change one graph into another. Even in cases when the XML differs syntactically, this captures semantic similarities.

We ensured that our evaluation captured both syntactic accuracy and graphical fidelity across all intent types by combining token-level and structure-aware metrics.

6.3 Quantitative Results

Before deciding on which model to fine-tune, we compared the performance of a few-shot prompting baseline using LLaMA 2 and a retrieval-augmented generation (RAG) setup using Qwen. Given Qwen's superior performance across all tasks, we selected it for further instruction tuning using our custom dataset. This allowed us to move beyond few-shot capabilities and achieve better generalization.

6.3.1 Explain Intent (Textual Output)

- **Few-Shot (LLaMA 2):** BLEU = 0.63, ROUGE-L = 0.65
- **RAG (Qwen):** BLEU = 0.71, ROUGE-L = 0.76
- **Fine-Tuned (Qwen):** BLEU = **0.94**, ROUGE-L = **0.96**

The fine-tuned Qwen model significantly outperforms both baselines, producing more accurate and semantically rich explanations.

6.3.2 Generate and Edit Intents (Diagram XML Output)

- **Few-Shot (LLaMA 2):** Precision = 0.60, Recall = 0.65, F1 = 0.62, GED = 0.50
- **RAG (Qwen):** Precision = 0.72, Recall = 0.77, F1 = 0.74, GED = 0.67
- **Fine-Tuned (Qwen):** Precision = **0.86**, Recall = **0.86**, F1 = **0.86**, GED = 0.89

The improvements are evident across both element-level accuracy (Precision, Recall, F1) and structural similarity (lower Graph Edit Distance), validating the benefit of fine-tuning with high-quality, task-specific examples.

7 Future Work

While the current system demonstrates promising results in diagram-related task understanding and generation, several directions can be pursued to further improve its performance and scalability:

- **Dataset Expansion:** Increase the size and diversity of the training data by adding more intents, varied dialogue patterns, and domain-specific edge cases. This will improve the ability to generalize and decrease overfitting, especially when dealing with unclear or invisible queries.
- **Reinforcement Learning from Human Feedback (RLHF):** Implement RLHF to align the model outputs with human preferences and improve the quality, helpfulness, and factual accuracy of the responses. This technique can bridge the gap between task success and human satisfaction.
- **Full System Deployment:** Integrate the fine-tuned model into a scalable, production-ready web application. This includes building an efficient backend for low-latency inference, deploying the model via containers (e.g., Docker), and designing a user-friendly interface for real-time interaction and diagram visualization.
- **Multimodal Integration:** Extend the chatbot's capabilities to understand and generate visual diagrams directly using multimodal models or by interfacing with diagram rendering engines such as Graphviz or Mermaid.
- **Evaluation Framework:** Develop an automated assessment pipeline that includes task-specific metrics (e.g., intent classification accuracy, diagram completeness, and syntax validity) alongside human evaluation to ensure robustness.
- **Transfer Learning to Other Domains:** Explore the adaptation of the chatbot to related domains such as UML generation, flowchart automation, or educational tutoring systems by fine-tuning on domain-relevant conversations.

These enhancements would not only increase the system's accuracy and usability but also broaden its applicability across various real-world scenarios involving structured diagram generation and interpretation.

8 Conclusion

In this project, we created, optimized, and assessed a domain-specific chatbot that can comprehend and produce material relevant to structured diagrams for three major purposes: *Generate*, *Edit*, and *Explain*. We made it possible for the language model to learn and generalize across a variety of diagram jobs by utilizing a bespoke conversational dataset that was constructed from real-world Draw.io XML examples and enhanced with prompt-response interactions.
To guarantee training feasibility on constrained hardware while maintaining performance, we used the Qwen 2.5 model and parameter-efficient fine-tuning approaches (LoRA via the

Unsloth library). Structured prompt formatting, targeted PEFT modules, and 4-bit quantization were all included in our training pipeline. Using intent-appropriate metrics such as BLEU, ROUGE, precision, recall, F1 score, and Graph Edit Distance, the evaluation showed significant performance in both natural language and XML creation tasks.
We created a responsive and reliable framework that serves as the basis for intelligent diagram creation and manipulation by integrating our model with a unique inference endpoint. The project bridges the gap between technical diagram synthesis and natural language understanding by demonstrating the efficacy of fine-tuning big language models on specialized, structured domains.
Overall, this work represents a significant step toward building intelligent assistants capable of interfacing with technical diagram formats in a human-centered manner. The methodologies and tools used here serve as a replicable blueprint for similar domain-specific LLM customization efforts.

9 Contributions

This project was the result of a strong collaborative effort between both team members, Chaimaa El Argoubi and Souhail Karam. Together, we worked closely across all stages of the project, from dataset design and preprocessing to model experimentation, fine-tuning, evaluation, and interface development. The accomplishments of this project demonstrate a common dedication to investigating many architectural options, evaluating findings critically, and developing a workable system for diagram-based natural language interaction.

Chaimaa El Argoubi played a central role in the project, she was responsible for designing and implementing the Retrieval-Augmented Generation (RAG) system using the Qwen 2.5 model. She led the integration of the model via a LangChain wrapper linked to a custom inference endpoint provided by our professor. Chaimaa handled the RAG pipeline's development, designed effective prompt templates, and conducted extensive experiments on Qwen's performance. She also played a key role in applying parameter-efficient fine-tuning (PEFT) with LoRA using the Unsloth library, optimizing the training pipeline, and preparing evaluation metrics for XML-based tasks. Additionally, Chaimaa contributed to building the frontend and backend integration of the final chatbot interface.

Souhail Karam played a pivotal role in the project, spearheading the few-shot prompting experiments with the LLaMA model. He was responsible for designing structured input formats, configuring the inference pipeline, and systematically benchmarking the model's outputs against the RAG-enhanced system. His efforts were crucial in validating early hypotheses and identifying key limitations of off-the-shelf LLMs. Souhail also co-led the preparation of the training dataset, including cleaning, formatting, and transforming it into conversation-style pairs suitable for fine-tuning. He was actively involved in the LoRA-based training of the Qwen model and helped define robust evaluation metrics tailored to diagram generation and editing. In addition, he contributed significantly to frontend development, ensuring seamless user interactions and end-to-end system integration.

References

[1] Hugging Face. Qwen - hugging face. https://huggingface.co/Qwen, 2024. Accessed: 2025-07-01.

[2] Ksm00i. Fine-tuning qwen 2.5 coder 14b llm - sft + peft. https://www.kaggle.com/code/ksmooi/fine-tuning-qwen-2-5-coder-14b-llm-sft-peft, 2024. Accessed: 2025-07-01.

We used the Qwen model from Hugging Face [1] and followed a fine-tuning strategy inspired by the approach in the Kaggle notebook [2].