Machine Learning with Applications 20 (2025) 100660

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

# Unified modeling language code generation from diagram images using multimodal large language models

Averi Bates [a], Ryan Vavricka [a], Shane Carleton [b], Ruosi Shao [c], Chongle Pan [a,*]

[a] School of Computer Science, University of Oklahoma, 110 W. Boyd St., Norman, OK, US
[b] Enterprise Architecture, MapLarge, 1201 Peachtree Street NE, Building 400, Suite 1750, Atlanta, GA, US
[c] School of Communication, Florida State University, 4100 University Center, Building C, Tallahassee, FL, US

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.15103682

*Keywords:*
UML
Large language models
Machine learning
Code generation

ABSTRACT

The Unified Modeling Language is a standardized visual language widely used for modeling and documenting the design of software systems. Although many tools are available that generate UML diagrams from UML code, generating executable UML code from image-based UML diagrams remains challenging. This paper proposes a new approach to generate UML code using a large multimodal language model automatically. Synthetic UML activity and sequence diagram datasets were created to train and test the model. We compared the standard fine-tuning with LoRA techniques to optimize base models. The experiments measured the code generation accuracy across different model sizes and training strategies. These results demonstrated that domain-adapted MM-LLMs perform for UML code generation automation, whereby, at the best model, it achieved BLEU and SSIM of 0.779 and 0.942 on sequence diagrams. This will enable the modernization of legacy systems and decrease the manual effort put into software development workflows.

## 1. Introduction

In software engineering, the Unified Modeling Language (UML) is widely used to represent and analyze complex systems visually (Booch et al., 1998). UML provides a standardized framework for documenting, designing, and refining software solutions (Lu & Alexandru, 2023; Metzner, 2024). It supports various stages of the development lifecycle by structuring software design and clarifying relationships among components, encouraging collaboration among stakeholders (Koç et al., 2021). Metzner compares UML proficiency to an architect's use of blueprints, highlighting its importance in creating systematic, organized design frameworks (Metzner, 2024).

Despite its utility, UML diagrams are often stored in fixed formats such as PDFs or images, limiting their usability in automated workflows. Static formats complicate iterative design tasks, such as design verification or consistency checks, particularly in agile development. Moreover, in legacy systems, UML diagrams and corresponding code may become inaccessible, leading to outdated documentation and increasing manual reconstruction efforts. For instance, a bank modernizing its legacy transaction processing system may only have outdated PDFs and scattered notes of its original UML diagrams. Without tools to automate conversion into editable, machine-readable formats, developers must painstakingly reconstruct designs, risking overlooked dependencies, security vulnerabilities, and higher maintenance costs. Similar challenges arise in collaborative environments where static UML diagrams lose relevance over time as project teams evolve. The need for automated UML-to-code generation is becoming increasingly urgent as systems grow in complexity. Manual translation of diagrams into code is labor-intensive and error-prone. Automated tools that extract and convert UML diagrams into editable formats can streamline development, reduce errors, and preserve design integrity.

### 1.1. Addressing the problem

To address the limitations of rule-based tools, this research leverages Multimodal Large Language Models (MM-LLMs), which integrate advanced machine learning techniques to process visual and textual data. MM-LLMs analyze complex UML diagram components without relying on static rule sets, such as class relationships, dependencies, and hierarchical structures. MM-LLMs can handle non-standard, hand-drawn, or domain-specific diagrams by dynamically interpreting visual UML constructs and corresponding annotations. The multimodal approach offers significant advantages for scalability and adaptability. As UML diagrams evolve during iterative development, MM-LLMs can

* Corresponding author.
*E-mail addresses:* averi.j.bates-1@ou.edu (A. Bates), ryan_vavricka@ou.edu (R. Vavricka), shane.carleton@maplarge.com (S. Carleton), rs24bl@fsu.edu (R. Shao), cpan@ou.edu (C. Pan).

https://doi.org/10.1016/j.mlwa.2025.100660
Received 31 December 2024; Received in revised form 12 April 2025; Accepted 22 April 2025
Available online 30 April 2025

extract meaningful information from incomplete or partially updated diagrams, ensuring consistency across workflows like design validation, automated code generation, and documentation.

Automated code generation from visual representations has gained traction in recent years. For example, pix2code demonstrated the conversion of GUI screenshots into executable code across platforms like iOS, Android, and the web (Beltramelli, 2018). Using CNNs and LSTM networks, it bypasses complex heuristics by interpreting GUI elements directly from images. Building on this work, Cai et al. (2023)'s GUICG enhances component localization and classification through CNN-based target detection, achieving notable accuracy improvements on large-scale GUI datasets. Beyond GUIs, Ellis et al. explored code generation from freehand sketches using a two-stage model combining CNNs with program synthesis techniques (Ellis et al., 2018). Identifying geometric primitives and constructing structured code representations highlights the potential of machine learning in bridging visual and code-based workflows. Researchers have also started developing models that extract elements from UML diagrams stored in non-editable formats. For instance, ReSECDI identifies core UML components like classes and relationships across varying formats (Chen et al., 2022). However, while ReSECDI provides syntactic extraction, it does not yet generate executable code, underscoring the need for machine learning techniques to bridge this gap (Conrardy & Cabot, 2024).

Large Language Models (LLMs) have demonstrated impressive capabilities in understanding and generating human-like text. Advanced transformer-based architectures such as BERT (Devlin et al., 2018), GPT (Radford et al., 2018), and T5 (Raffel et al., 2020) capture contextual relationships within text, enabling tasks like summarization, question answering, and code generation. However, LLMs process only textual data, limiting their utility in domains requiring multimodal understanding, such as image or video analysis. Researchers developed MM-LLMs that incorporate visual, auditory, and textual data into unified frameworks to address these limitations. Models like CLIP (Radford et al., 2021) align visual embeddings with textual embeddings, enabling cross-modal tasks such as visual question answering (VQA) and zero-shot image classification. For example, CLIP (Contrastive Language–Image Pretraining) learns shared visual and textual semantics from large-scale datasets of image-text pairs, effectively bridging linguistic context and visual perception. MM-LLMs leverage these multimodal capabilities to tackle tasks that traditional LLMs cannot address, such as interpreting UML diagrams and generating corresponding executable code. MM-LLMs align diagram structures with text-based outputs by combining visual encoders with language models, enhancing accuracy and flexibility for applications like UML-to-code automation.

### 1.2. Research objectives and contributions

Our research aims to enhance the capabilities of MM-LLMs for UML-to-code generation by fine-tuning them to handle activity and sequence diagrams—key behavioral diagrams in software design. By leveraging MM-LLMs' ability to interpret visual UML components and generate executable code, this work addresses the limitations of existing tools. Specifically, the study develops a scalable framework to evaluate model performance across dataset sizes and UML complexities. The key contributions include a framework for UML diagram-to-code automation, insights into factors affecting MM-LLM generalization, and recommendations for future enhancements, such as improving dataset diversity and multimodal alignment. This research bridges the gap between static UML documentation and modern, automated workflows, offering practical solutions for improving software design efficiency.

### 1.3. UML scope

UML serves as a standard framework for visualizing and analyzing system designs. It consists of various diagram types categorized into structural and behavioral groups. Structural diagrams, such as class diagrams, capture static system elements like classes, attributes, and relationships. On the other hand, behavioral diagrams focus on dynamic interactions and workflows within systems. Our research focuses on behavioral diagram types—activity and sequence diagrams—because they are critical in representing workflows and communication flows, which are essential for UML-to-code automation.

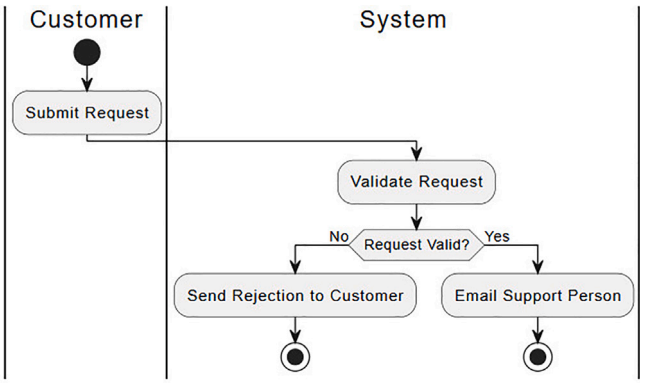


**Fig. 1.** Activity diagram showing the decision paths for valid and invalid requests.

```
@startuml
|Customer|
start
:Submit Request;

|System|
:Validate Request;
if (Request Valid?) then (No)
  :Send Rejection to Customer;
  stop
else (Yes)
  :Email Support Person;
  stop
endif
@enduml
```

**Fig. 2.** PlantUML code that represents the activity diagram structure, including the decision points and action paths.

Activity diagrams excel at modeling control flow and decision-making processes within a system, providing a high-level overview of workflows. Each action or task is depicted as a rounded rectangle, with arrows (control flows) indicating the progression of tasks. Decision nodes handle conditional branches, while swimlanes assign responsibilities to actors or components. These features make activity diagrams particularly valuable for visualizing multi-actor systems and concurrent processes. Fig. 1 provides an example of an activity diagram for a request validation process, where customer requests are either approved or rejected based on conditions. The corresponding PlantUML code seen in Fig. 2 defines this workflow and its components.

In contrast, sequence diagrams focus on temporal interactions between system components, emphasizing communication order and dependencies. A lifeline represents each component, while horizontal arrows signify message exchanges or method calls. Sequence diagrams excel in modeling client–server interactions, API calls, and service workflows, offering developers insight into component dependencies

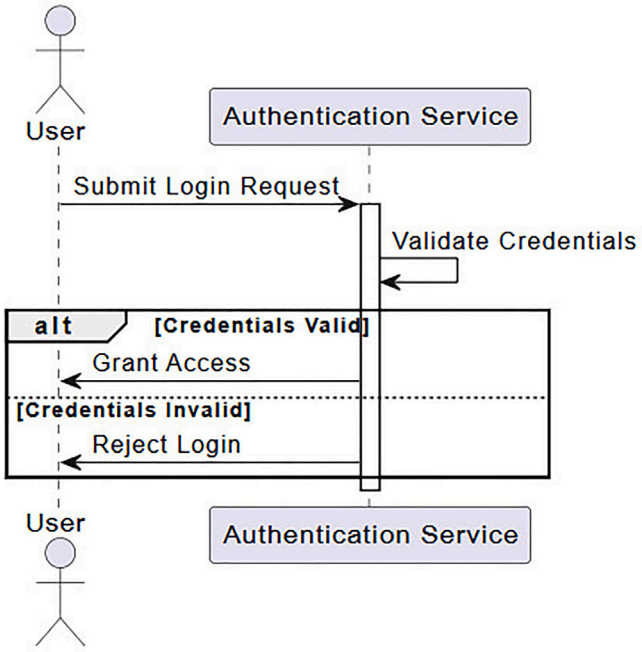


**Fig. 3.** Sequence diagram representing validating user information and granting access.

over time. Figs. 3 and 4 present a user log-in process captured in a sequence diagram. It models how users interact with the authentication service, highlighting decision paths for valid and invalid credentials. The corresponding PlantUML code details these interactions, focusing on the sequence of events and activation periods.

Activity diagrams and sequence diagrams are inherently complementary. While activity diagrams provide a broad, workflow-centric perspective suited for process automation, sequence diagrams emphasize fine-grained, event-driven interactions between components. Focusing on these two diagram types, allows actionable insights into UML-to-code automation. Activity diagrams support the generation of control-flow logic and parallel execution, while sequence diagrams align with detailed communication mapping.

### *1.4. Automating code generation*

LLMs have significantly advanced automated code generation, providing tools that analyze textual and structural information to generate functional code. For example, Meta's CodeCompose offers developers real-time code suggestions and streamlining routine coding tasks through context-aware predictions (Murali et al., 2024). Similarly, OpenAI's Codex, fine-tuned from GPT-3 on GitHub datasets, achieves a notable success rate in solving coding challenges through iterative sampling strategies (Chen et al., 2021). These tools demonstrate the potential of LLMs to improve productivity by reducing manual intervention in code generation. Despite their success, traditional LLMs are limited in diagram-to-code automation. Converting UML diagrams into executable code demands both linguistic and visual comprehension. Thus, the use of MM-LLMs fine-tuning techniques play a critical role in enhancing MM-LLMs for UML-to-code generation. Instruction-based fine-tuning and Chain-of-Thought prompting improve the models' ability to generate contextually accurate and logically consistent code. Recent studies have shown that fine-tuned models generalize effectively across benchmarks, maintaining structural fidelity in generated outputs (Chung et al., 2022). Studies like Plot2Code and Design2Code highlight the challenges of converting structured visuals into functional code. For instance, while GPT-4 achieved high accuracy in generating executable code from scientific plots, its performance declined with increasing visual complexity (Shi et al., 2024; Wu et al., 2024). These findings underscore the importance of tuning MM-LLMs to handle intricate UML diagrams, where maintaining relational and sequential accuracy is critical. In this research, we extend these advancements by fine-tuning MM-LLMs to generate code from activity and sequence diagrams autonomously. The models are trained to align diagram structures with functional outputs, ensuring high syntactic and structural fidelity.

## 2. Methodology

Automating UML code generation requires models capable of processing visual and textual data. We use the Large Language and Vision Assistant (LLaVA) (Liu, Li, Wu et al., 2024), specifically its improved version, LLaVA-1.5 (Liu, Li, Li, Lee, 2024), a foundational model. Both LLaVA and LLaVA-1.5 integrate a CLIP-based visual encoder (Lu et al., 2024) with the Vicuna language model (LMSYS, 2023) via a vision-language connector, enabling the extraction of visual features from UML diagrams and translating them into accurate textual outputs. LLaVA uses a linear projection layer to align visual embeddings with the language model. In contrast, LLaVA-1.5 introduces a multi-layer perceptron as a more complex connector, enabling nuanced interactions between visual and textual representations. The architectural enhancements, combined with expanded visual instruction tuning on VQA and diagram datasets, improve LLaVA-1.5's performance for tasks requiring detailed diagram analysis, such as UML-to-code generation.

### *2.1. Fine-tuning and LoRA fine-tuning*

The base model, the 'original model' or 'baseline model', is the pre-trained LLaVA model without fine-tuning. Although this base model has strong general capabilities, it struggles with the task-specific syntax and structure required to accurately interpret and generate UML diagrams, particularly given the detailed nature of some UML formats. Fine-tuning addresses this limitation by training the model on task-specific data, enabling it to understand better the unique demands of UML diagrams, such as their precise syntax and structural rules. Our study uses 'full fine-tuning' and 'standard fine-tuning' interchangeably to describe tuning that updates all model weights using additional, task-specific data. The baseline model, which remains unaltered, serves as the starting point for the fine-tuning process. Fine-tuning adapts the model to address UML diagram-specific requirements better, improving its performance and accuracy in diagram-to-code generation tasks.

Four models were trained using two fine-tuning strategies: standard fine-tuning and LoRA (Low-Rank Adaptation) fine-tuning. The models included two sizes: one with 7 billion parameters (7B) and one with 13 billion parameters (13B). The model sizes represent different trainable parameters, with the large model having more. Standard fine-tuning updates all model parameters, while LoRA introduces trainable low-rank matrices that reduce the computational burden by focusing on updating only selected layers. LoRA allows for efficient adaptation of large language models, such as LLaVA-1.5.

LoRA reduces the number of trainable parameters by introducing trainable low-rank matrices into the architecture. Specifically, instead of updating all parameters in the model, LoRA modifies only a subset of the parameters in the form of low-rank matrices, significantly decreasing memory consumption during training. The low-rank approximation minimizes the number of trainable parameters by focusing only on updating specific components, leading to substantial reductions in memory usage and computational overhead (Hu et al., 2021). Despite this simplification, LoRA can demonstrate performance comparable to full fine-tuning in several large-scale language models. Given the scale of models like LLaVA-1.5, incorporating LoRA enables the system to maintain computational efficiency without compromising its ability to adapt to the unique requirements of tasks such as UML diagram analysis. This balance between resource efficiency and adaptability makes LoRA a compelling approach for fine-tuning large models on domain-specific tasks.

```
@startuml
actor User
participant "Authentication Service" as AuthService

User -> AuthService: Submit Login Request
activate AuthService

AuthService -> AuthService: Validate Credentials
alt Credentials Valid
    AuthService -> User: Grant Access
else Credentials Invalid
    AuthService -> User: Reject Login
end

deactivate AuthService
@enduml
```

**Fig. 4.** PlantUML code that defines the interactions for the user login process shown in the sequence diagram.

**Table 1**
Hyperparameters for LoRA and standard fine-tuning.

| Hyperparameter | LoRA fine-tuning | Standard fine-tuning |
|---|---|---|
| lora_enable | True | None |
| lora_r | 128 | None |
| lora_alpha | 256 | None |
| mm_projector_lr | $2 \times 10^{-5}$ | None |
| learning_rate | $2 \times 10^{-4}$ | $2 \times 10^{-5}$ |
| per_device_train_batch_size | 16 | 8 |
| gradient_accumulation_steps | 4 | 4 |

### 2.2. Hyperparameters

The fine-tuning process utilized several hyperparameters. Common hyperparameters, such as the *learning rate* and *batch size*, applied to both standard and LoRA fine-tuning, while others were specific to LoRA. The differentiation ensured that each fine-tuning strategy was optimized for its specific methodology. The process adhered to the original implementation of LLaVA-1.5 to represent the architecture accurately. Table 1 summarizes the hyperparameters applied in both fine-tuning strategies.

The hyperparameters *lora_r* and *lora_alpha* are central to LoRA fine-tuning. *lora_r* determines the rank of the low-rank matrices integrated into the model. Increasing *lora_r* enhances the expressive power of these matrices, enabling more nuanced adaptations but requiring greater memory and computational resources (Hu et al., 2021). *lora_alpha*, a scaling factor, adjusts the influence of the low-rank matrices during training. Increasing *lora_alpha* amplifies the impact of the matrices on the model output, enhancing their prominence in the learning process. The *mm_projector_lr* hyperparameter sets the learning rate for the projector layer. Since LoRA fine-tuning updates only a subset of parameter values, the projector learning rate ensures balanced updates concerning the rest of the model. LoRA fine-tuning employs a higher learning rate and larger batch size, as updating fewer parameters than standard fine-tuning allows the model to converge more efficiently. A batch size of 8 was used for standard fine-tuning, while LoRA fine-tuning utilized a batch size of 16 to accommodate its more efficient parameter updates. Despite the differences in batch size, the training times and computational performance metrics provide valuable insights into each approach's efficiency and scalability.

### 2.3. Synthetic data generation

Two types of synthetic UML diagrams — activity and sequence diagrams — were programmatically generated using PlantUML, a text-based diagramming tool widely adopted for its ease of integration and automation. The diagrams were constructed using randomized text strings drawn from a curated vocabulary of the 25,286 most popular words found in English. These strings were inserted into standard PlantUML syntax structures, thereby generating a wide variety of visually valid UML diagrams that are structurally diverse but semantically nonsensical. Importantly, the diagrams do not follow any coherent application logic; rather, they serve as syntactic approximations of real UML diagrams. This approach was chosen to isolate the visual parsing and language modeling capabilities of the system without introducing domain-specific priors.

PlantUML enables direct conversion of textual UML descriptions into graphical formats, supporting both rasterized PNG and vectorized SVG outputs. Each synthetic diagram was exported in PNG format. The sequence diagrams emphasize participant interactions and message flows, as illustrated in Fig. 5. The corresponding PlantUML code showcases the structured textual representation of visual relationships in Fig. 6. In a similar fashion, activity diagrams were synthesized to reflect control flows including branching, merging, and parallel operations, represented visually in **Fig. S1** and in PlantUML text format (**Fig. S2**). Their labels and process steps are composed of randomly selected words, resulting in workflows that lack meaningful interpretation. Nevertheless, they adhere to the syntactic rules of UML activity diagramming, making them suitable for testing model robustness to surface-level structural variation.

#### 2.3.1. Dataset breakdown

The synthetic data set consisted of activity and sequence diagrams split into four sizes, 'small', 'medium', 'large', and 'extra large', each containing various diagrams and the corresponding UML code files. Each one had varying amounts of training and testing data. Table 2 outlines the distribution of the data set for training and testing. The training was conducted separately on the types of activity and sequence diagram to optimize the model's performance, ensuring task-specific optimization.

Each dataset entry contains:

1. UML diagram image (PNG/SVG format).
2. Corresponding UML code (PlantUML syntax text file).
3. Pairing of prompt templates.

All generated diagrams and corresponding PlantUML source files were stored with unique identifiers to ensure dataset traceability and to prevent unintentional duplication. Additionally, each image-code pair was serialized into a JSON format (see Supplemental **Fig. S3**), which was used as the primary input format during model training. The dataset was partitioned using a simple random sampling procedure

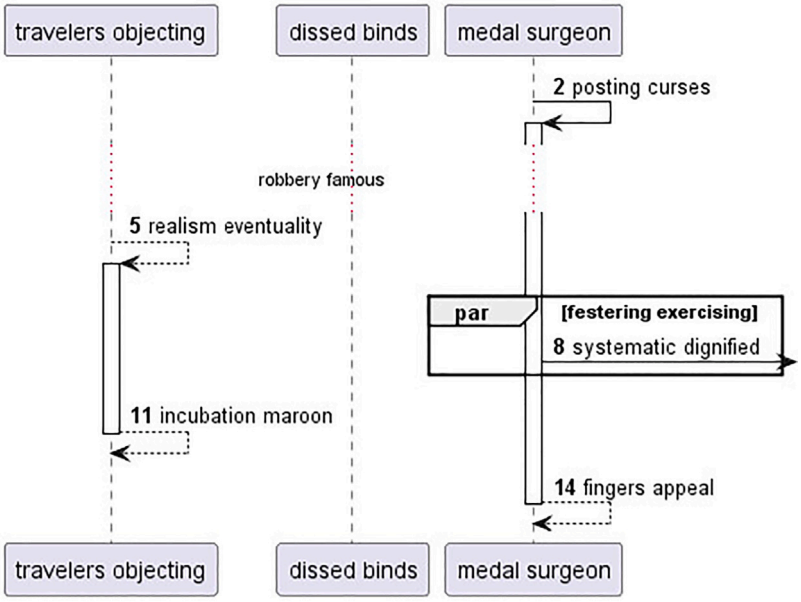


**Fig. 5.** Sequence diagram depicting participants, messages, and activation events.

```
@startuml
participant "travelers objecting" as 0
participant "dissed binds" as 1
participant "medal surgeon" as 2
autonumber 2 3
2 -> 2++ : posting curses
...robbery famous...
0 --> 0++ : realism eventuality
par festering exercising
2 ->] : systematic dignified
end
return incubation maroon
return fingers appeal
@enduml
```

**Fig. 6.** PlantUML code generating the sequence diagram.

**Table 2**
Dataset sizes for training and testing.

| Dataset size | Diagram type | Training size | Testing size |
|---|---|---|---|
| Small | Activity | 6000 | 1500 |
| Medium | Activity | 12,000 | 3000 |
| Large | Activity | 24,000 | 6000 |
| Extra large | Activity | 120,000 | 30,000 |
| Small | Sequence | 6000 | 1500 |
| Medium | Sequence | 11,994 | 2998 |
| Large | Sequence | 23,994 | 5998 |
| Extra large | Sequence | 119,994 | 29,998 |

to produce an 80/20 train-test split. Testing examples were selected without regard to structure or content, ensuring unbiased evaluation across the synthetically generated samples. The testing split had an additional step where data was organized in a question-answer format and added as another JSON file. The structured format supports effective multimodal learning, as demonstrated below.

#### 2.3.2. Real-world testing evaluation

While the majority of our dataset is synthetically generated to ensure controlled variability and sufficient volume for training, it is important to assess how well models perform on naturally occurring data. To this end, we curated a small set of real-world UML diagrams — 29 activity diagrams and 28 sequence diagrams — totaling 57 examples. These were sourced from publicly available resources and serve as a benchmark for evaluating the generalization capabilities of models trained exclusively on synthetic data. The real-world data is used solely for testing and enables comparison across models with varying levels of training exposure. We evaluate the original model, a minimally trained small LoRA variant, two standard models (7B and 13B) trained on the small dataset, and finally, models trained extensively on the extra-large

**Table 3**
Real-world testing data distribution.

| Data type | Testing size |
|---|---|
| Activity diagrams | 29 |
| Sequence diagrams | 28 |

dataset using both LoRA and full parameter tuning. This setup allows us to examine the influence of training intensity — from no training to full-scale training — on the model's ability to generalize to naturally structured UML diagrams. Including this evaluation provides valuable insights into the robustness and practical applicability of our models. It helps determine whether large-scale synthetic training introduces limitations or enhances the model's ability to interpret real-world inputs. The distribution of the real-world testing dataset is summarized in Table 3.

The real-world UML diagrams were sourced from publicly available, educational, and technical resources that include software engineering tutorials, documentation repositories, and open-source guides. Specifically, we collected data from three primary sources: the official PlantUML guide (PlantUML, 2025), as well as tutorial-based resources on activity and sequence diagrams from WebDevTutor (Web Dev Tutor, 2023a; Web Dev Tutor, 2023b). Selection criteria focused on clarity, diversity of structure, and representational fidelity to practical use cases in software design. All chosen diagrams were fully rendered and accompanied by the corresponding PlantUML source code, ensuring their suitability for both image-based and code-based model evaluation. Unlike the synthetic diagrams, which were generated by inserting randomized text strings into grammatically correct UML syntax templates, the real-world diagrams were authored by domain experts or educators to illustrate concrete workflows and interactions. These real diagrams typically exhibit meaningful semantic content, coherent control or message flows, and intentional labeling conventions that reflect real application logic or business processes. For instance, activity diagrams in the real-world set often include process names suggesting a clear operational sequence. Sequence diagrams show realistic inter-object communications such as login procedures or database queries, in contrast to the semantically nonsensical but syntactically valid interactions found in synthetic examples.

### 2.4. Hardware and pipeline development

We completed all fine-tuning and evaluation tasks on the University of Oklahoma's supercomputing infrastructure, leveraging an SLURM-managed HPC cluster. Depending on node availability, training was conducted on NVIDIA A100 GPUs, either 40 GB or 80 GB versions. For standard full-parameter fine-tuning — required for larger models like Vicuna-13B and LLaVA-13B — we utilized four A100 GPUs to meet the memory and compute demands. In comparison, LoRA fine-tuning was successfully executed using just two A100 GPUs. All jobs were submitted via SLURM scripts, which specified GPU resources, runtime configurations, and distributed training parameters.

Our approach followed the official LLaVA training pipeline from the project's GitHub repository, incorporating full model and LoRA-based fine-tuning scripts. The process involved two key stages: feature alignment and visual instruction tuning. Depending on the experiment, we used either the standard (fine-tune.sh) or LoRA (finetune_lora.sh) script, with only minimal modifications tailored to our HPC setup and resource allocations (Liu, Li, Li, Lee, 2024).[1] Model checkpoints were saved at predefined intervals, and SLURM output was used to monitor logs in real-time. We performed all inference runs on a single A100 GPU per model for consistent evaluation, applying this to both our trained checkpoints and baseline models. The setup ensured uniform memory and compute conditions across all experiments. All associated pre- and post-processing — such as JSON parsing and results aggregation — was handled through Python 3.11 scripts within a Linux terminal environment.

### 2.5. Evaluation metrics

We hypothesize that using large datasets and larger model sizes will improve BLEU and SSIM metrics performance, reflecting enhanced textual accuracy and visual fidelity in generated UML code. Simultaneously, we expect these configurations to yield lower error rates, indicating fewer syntax errors, structural misalignments, and overall diagram inaccuracies. While both LoRA fine-tuning and standard fine-tuning are predicted to produce comparable BLEU and SSIM improvements, LoRA is anticipated to offer superior computational efficiency. Specifically, it should achieve faster training times and require fewer computational resources while maintaining similar output quality. This hypothesis underpins the study's goal of balancing accuracy and efficiency to optimize UML-to-code generation workflows.

The BLEU metric, introduced by Papineni et al. (2002), is widely used for evaluating text generation tasks. It measures the similarity between generated text and a reference by comparing n-grams, offering a quantitative measure of syntactic accuracy. BLEU's simplicity, scalability, and independence from specific programming languages make it well-suited for large-scale evaluation of generated UML code. In prior studies, BLEU has been successfully applied to similar tasks such as evaluating large-scale language models for code generation (Chen et al., 2021; Zhang et al., 2023). A higher BLEU score indicates stronger alignment between the generated output and the ground truth, ensuring that syntactic fidelity is maintained during UML-to-code translation.

The Structural Similarity Index Measure (SSIM) is a common image quality metric that evaluates the visual similarity between generated and reference images (Reznik, 2023; Wang et al., 2004). Unlike pixel-based metrics, SSIM considers luminance, contrast, and structural features to align with human visual perception. This makes SSIM especially relevant for UML diagrams, as it prioritizes the layout and relationships between elements over minor pixel variations. SSIM produces scores ranging from −1 to 1, where higher values reflect closer visual resemblance. When evaluating UML diagrams, SSIM helps ensure that structural elements, such as object relationships, decision nodes, and component alignment, remain intact. **Figs S4** and **S5** demonstrate a high-scoring diagram, ground truth and generated, respectively. **Figs S6** and **S7** demonstrate a low-scoring diagram, ground truth and generated, respectively. A high SSIM and BLEU score signifies that the generated UML diagram aligns visually and textually with the reference, preserving spatial arrangement and semantic content. Conversely, low scores indicate discrepancies, such as misaligned components or incorrect labels, compromising structural fidelity. These examples underscore the critical role of SSIM and BLEU in validating the performance of models tasked with UML code generation. Maintaining high visual and textual accuracy is essential to ensure the diagrams remain functional for software development workflows.

Several metrics were incorporated to evaluate our models' computational efficiency, including Floating-Point Operations (FLOS), training time, samples per second, steps per second, and evaluation time. FLOS represents the cumulative number of operations during training, offering a measure of computational demand. As the model size and dataset complexity increase, FLOS and training time also rise, revealing scalability challenges. Samples per second and steps per second measure throughput and gradient updates, respectively, providing insights into how efficiently the models handle large datasets. Evaluation time measures the latency of generating a response to input image-prompt pairs, with longer times highlighting potential bottlenecks in handling more complex UML diagrams.

The error analysis follows a structured evaluation process inspired by methodologies from prior studies (Conrardy & Cabot, 2024). Three

[1] Code available at: https://github.com/haotian-liu/LLaVA.

main error types are examined: syntax errors, UML code absence, and diagram mismatches. Syntax errors refer to missing symbols or incomplete code blocks that prevent the diagram from rendering correctly. These errors are flagged automatically when the PlantUML rendering tool fails to generate valid diagrams. UML absence errors occur when the generated output is irrelevant or incomplete and contains placeholder text or non-UML responses. The diagram mismatch score quantifies logical inconsistencies, such as producing an incorrect diagram type (e.g., a class diagram instead of a sequence diagram). These error metrics comprehensively evaluate the model's reliability, with lower scores indicating better performance.

The combined use of BLEU, SSIM, and error metrics provides robust diagram-to-code generation models' accuracy of UML diagram-to-code generation models. The study analyzes cost and output quality by analyzing these metrics across data fine-tuning approaches to identify the optimal balance between computational cost and output quality, as well as fine-tuning approaches. Structured evaluation ensures that the textual and visual components align with the ground truth, while error analysis highlights areas that require further refinement.

## 3. Results

This section evaluates how the LLaVA-1.5 models perform across various datasets and scenarios, focusing on both computational efficiency and the quality of the generated diagrams compared to the ground truth. The computational analysis considers the total time spent and the overall training loss. The methodology states that BLUE and SSIM are used to assess model performance. These metrics provide insight into how well the models generate diagrams, examining aspects such as fluency, accuracy, and structural similarity to their original counterparts. Beyond performance evaluations, an error analysis was performed on the types of mistakes that emerge in diagram generation. Error analysis focuses on syntax issues, the absence of UML code, and mismatching diagram types.

We propose an experimental hypothesis to investigate the impact of model size, dataset scale, and fine-tuning techniques on the accuracy and efficiency of UML diagram-to-code generation. Building on the overarching hypothesis outlined in Section 1, which is that fine-tuned models will outperform the baseline, the experiments aim to assess how these factors influence the performance of BLEU and SSIM metrics. We hypothesize that increasing model and dataset sizes, from small to extra-large, will significantly enhance performance, resulting in greater textual and visual fidelity, fewer syntax errors, and improved alignment with ground truth diagrams. Furthermore, full and LoRA-based fine-tuning is expected to reduce errors such as syntax issues, UML omissions, and diagram mismatches while ensuring stronger adherence to the original UML structure.

LoRA fine-tuning results are anticipated to be comparable to full fine-tuning, with the added benefit of computational efficiency. Additionally, we predict that sequence diagrams, due to their more straightforward structure, will yield higher BLEU scores than activity diagrams. However, given the inherent complexity of activity diagrams, the latter is expected to benefit more significantly from fine-tuning. Finally, we hypothesize that expanding the dataset size will enhance the models' scalability and generalization, driving improvements across all major performance metrics. This hypothesis builds upon and complements the overarching premise, serving as the foundation for the experimental evaluation presented in the subsequent sections.

### 3.1. Model performance

BLEU scores tend to be lower at small dataset sizes. As seen in Fig. 8, the original 7B model's BLEU score was only 0.009 for the small sequence dataset, which is notably lower than both the full 7B model (0.113) and the LoRA 7B model (0.153). The original 13B model also performed poorly in the same small sequence dataset, registering a BLEU score of 0.011. A similar pattern appears in the small activity dataset, seen in Fig. 7, where low BLEU scores underscore the challenge of generating accurate outputs from such limited data. Nonetheless, even minimal fine-tuning appears to be more effective than the original model at recreating UML diagrams. As the dataset size increases, the same general trends emerge, as shown in Figs. 9, 10, 11, and 12. LoRA models usually outperform — or at least match — the full models, while the baseline tends to perform the worst. One exception is that the original 13B model outperformed the full 13B sequence data model. Moving on to the extra-large dataset, seen in Fig. 13, the LoRA 13B model achieved BLEU scores of 0.779 for sequence diagrams and 0.350 for activity diagrams, far surpassing earlier results. The full 13B model also showed comparable gains, scoring 0.769 and 0.347, respectively. The higher scores illustrate the significant benefits of using larger datasets.

SSIM scores followed a similar trajectory. In the small activity dataset, the baseline 7B model only reached 0.279, whereas the full and LoRA 7B models exceeded 0.600. Although the 7B model performed slightly better on the small sequence dataset, sequence SSIM scores remained below most of the activity. The lower scores are not seen the extra large dataset where SSIM scores outperform activity. With medium and large datasets, SSIM values continued to rise. Finally, the extra-large dataset delivered the highest SSIM scores overall. As seen in Fig. 13, the LoRA 13B model achieved 0.942 on sequence diagrams and 0.891 on activity diagrams, with the full 13B model closely matching these values. Such results suggest that larger datasets and fine-tuning strategies enable models to capture structural details more effectively, resulting in higher BLEU scores and stronger SSIM performance.

Across all dataset sizes, models trained on sequence data consistently outperformed those trained on activity data, achieving higher BLEU and SSIM scores. The LoRA 13B model achieved the highest scores, with 0.779 for BLEU and 0.942 for SSIM on sequence data in the extra-large dataset seen in Fig. 13. This discrepancy reflects the relative simplicity of sequence data compared to activity diagrams, which involve more intricate structural and syntactic complexity. While LoRA fine-tuning consistently enhanced performance across metrics, its benefits were more pronounced in sequence datasets. The performance gap between full and LoRA models was smaller for activity diagrams, suggesting that LoRA's targeted parameter updates are particularly effective for less complex data. These findings reinforce the critical role of fine-tuning in improving performance across different UML diagram types and dataset sizes.

#### 3.1.1. Real-world evaluation and limitations

Performance on real-world diagrams remained limited, as shown in Figs. 14, 15, and 16. While increasing model size and training duration led to slight gains—BLEU scores for the activity test set improved from 0.012 of the orignial 7B model to 0.038 for the extra large LoRA training data model, and SSIM increased from 0.213 to 0.643—these improvements were modest. Results showed an even smaller increase in sequence data from the small training models to the extra large training models. The relatively small improvements underscore the gap between training conditions and real-world application.

Supplementary figures, **Fig. S8** through **Fig. S11**, visualize these limitations, contrasting model-generated outputs with actual UML diagrams. The differences in terminology, formatting, and syntactic structure make it clear that synthetic data lacks realism in key areas. These observations point to the need for more representative training data that better mirrors real-world patterns. Given the difficulty of scaling real-world data collection, a practical path forward involves refining synthetic data generation to better simulate real-world use. Designing datasets around specific application domains — such as authentication, e-commerce, or user management — and adopting realistic naming conventions could significantly enhance generalization.

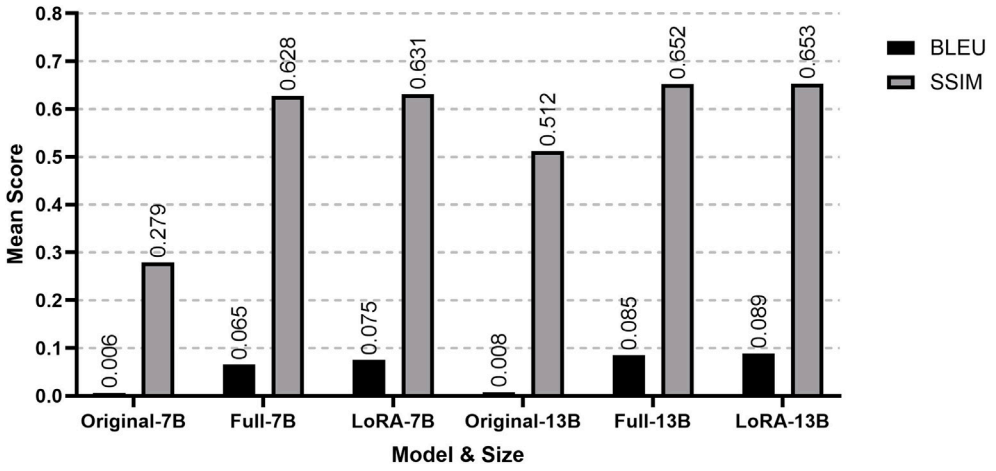


**Fig. 7.** Model performance on the small activity dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

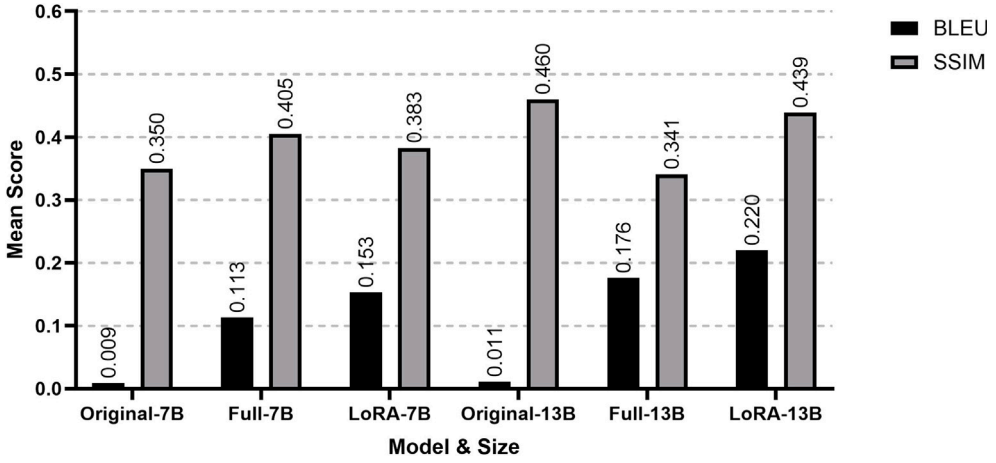


**Fig. 8.** Model performance on the small sequence dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

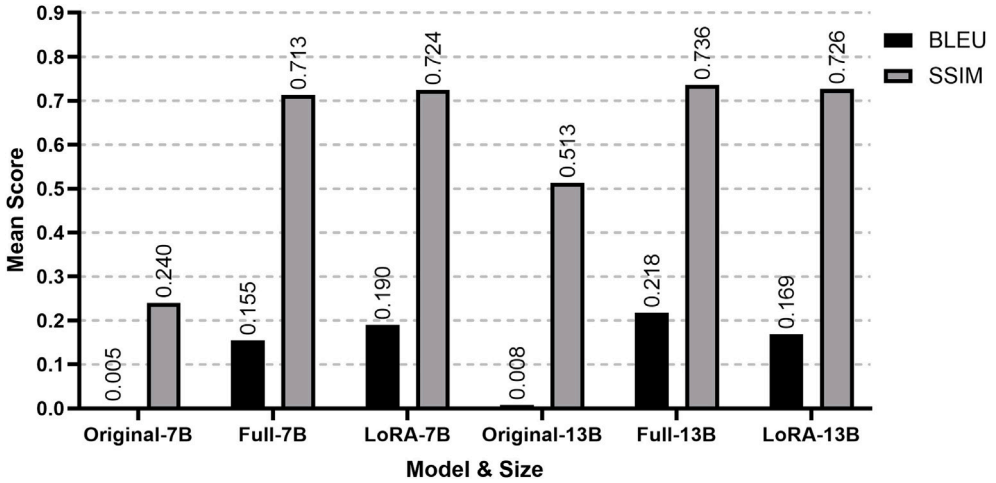


**Fig. 9.** Model performance on the medium activity dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

### 3.2. Computational efficiency

#### 3.2.1. Training trends

The computational demands of machine learning model training depend on the size of the dataset, the complexity of the dataset, and the model architecture. Tables 4, 5, 6, and 7 detail the metrics for training time and computational performance across activity and sequence datasets of varying sizes (small to extra-large) for both the 7B and 13B model architectures. While activity and sequence datasets share trends in step rates and throughput, differences in complexity lead to notable disparities in computational demands. Step rates decrease

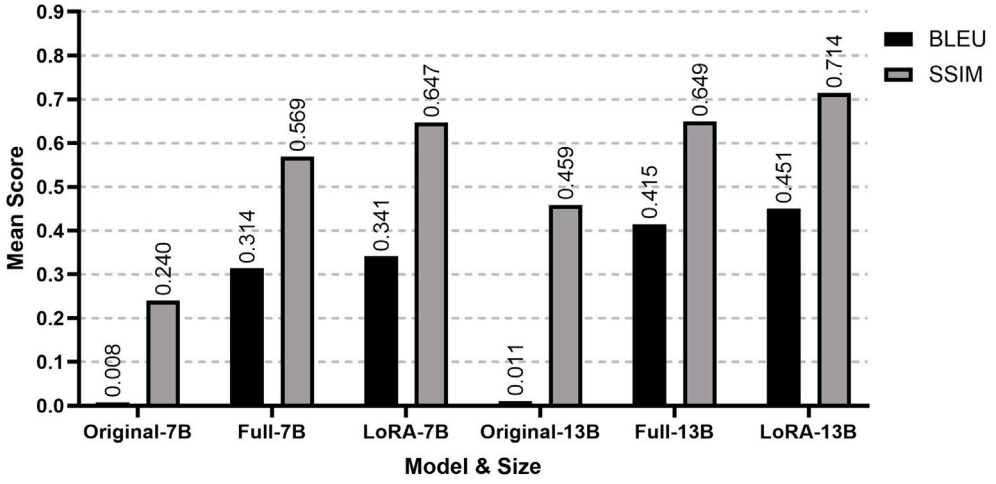


**Fig. 10.** Model performance on the medium sequence dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

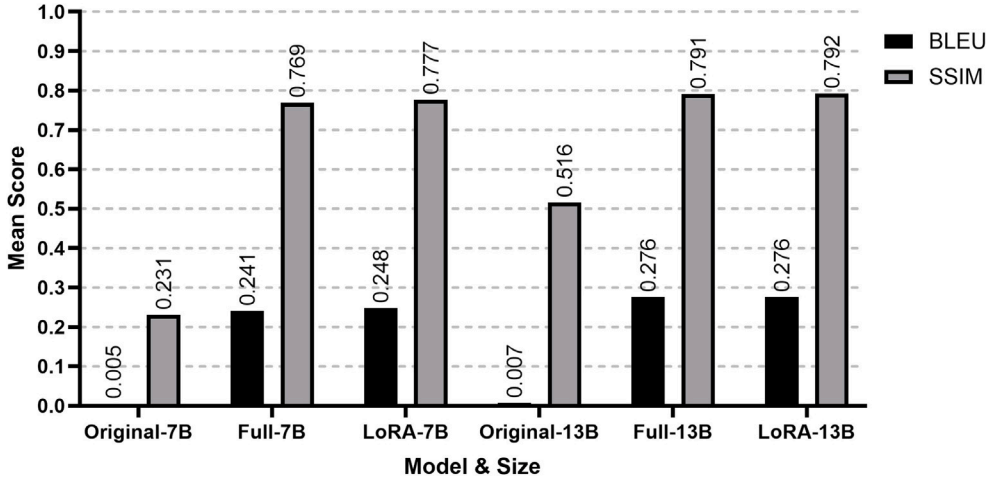


**Fig. 11.** Model performance on the large activity dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

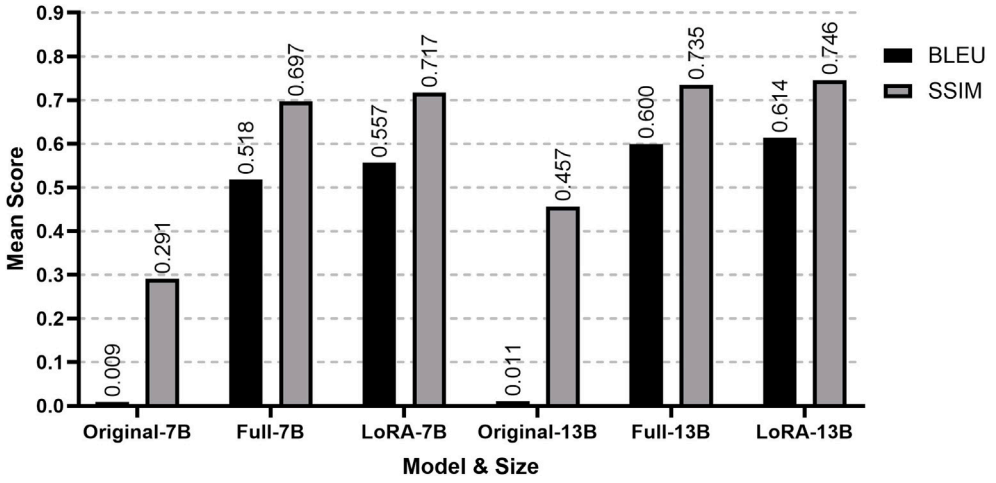


**Fig. 12.** Model performance on the large sequence dataset, grouped by model type and size, with BLEU and SSIM scores for each configuration.

slightly as the dataset size grows. For instance, the 7B full model processes 0.081, 0.080, and 0.079 steps per second for the small, medium, and large activity datasets. Small reductions in step rates reflect the increasing computational demands of larger datasets while demonstrating the model's consistency. Similarly, the 7B full model maintains a steady 0.108 steps per second across small, medium, and large sequence datasets, highlighting the lighter computational burden of sequence data. These results confirm the robustness of the model

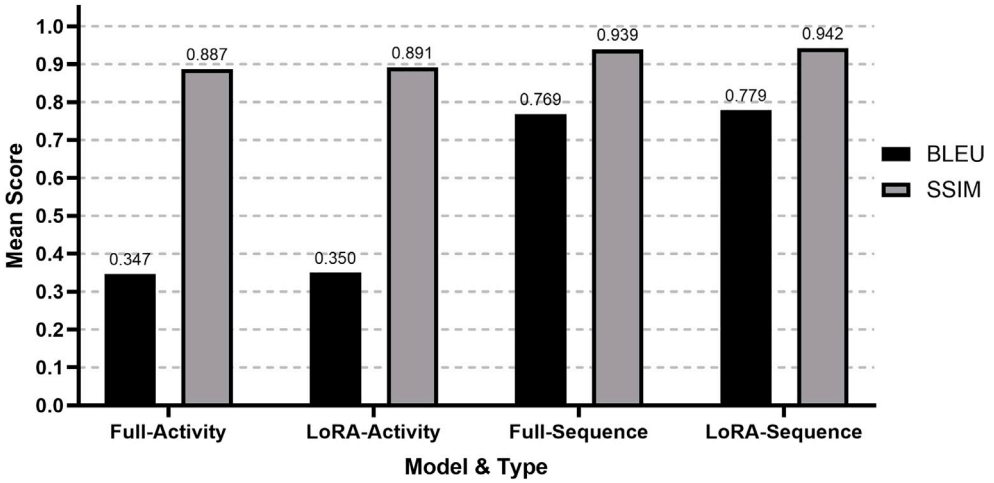


**Fig. 13.** Model performance on the extra-large dataset for activity and sequence diagrams, grouped by BLEU and SSIM scores across different model types and data types.

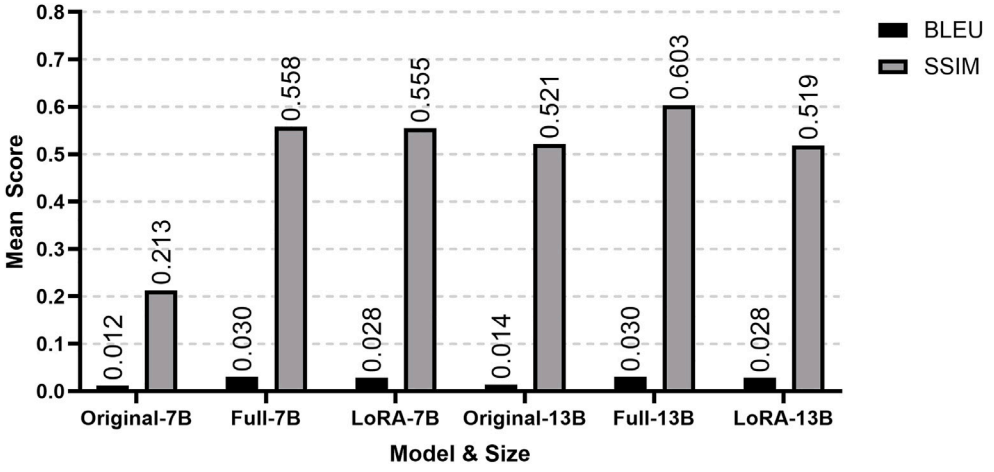


**Fig. 14.** Model performance on the real-world activity diagram dataset, showing BLEU and SSIM scores across baseline, full fine-tuning, and LoRA configurations with the small dataset models. Compared to synthetic data, improvements from fine-tuning are more modest.

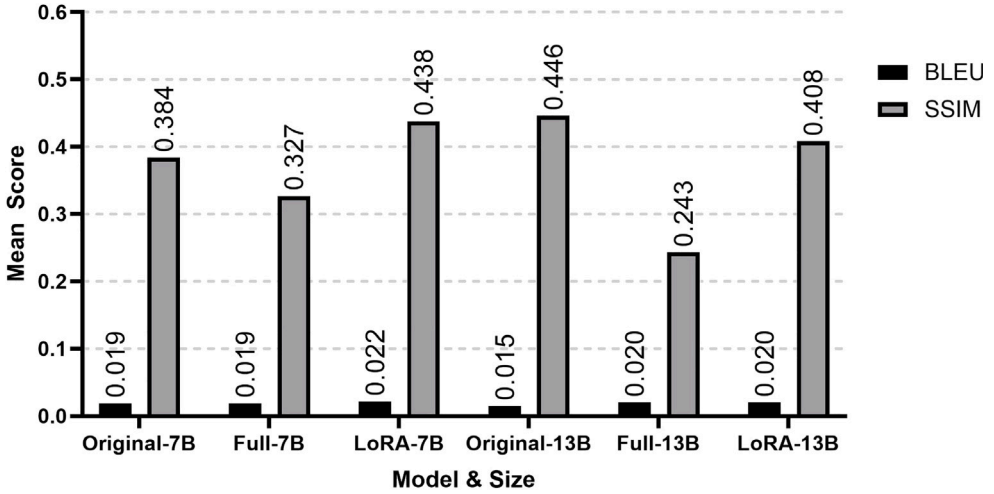


**Fig. 15.** Model performance on the real-world sequence diagram dataset. Fine-tuned models, for the small dataset models, show only minor gains over the baseline, suggesting that visual mismatches between training and test data affect generalization.

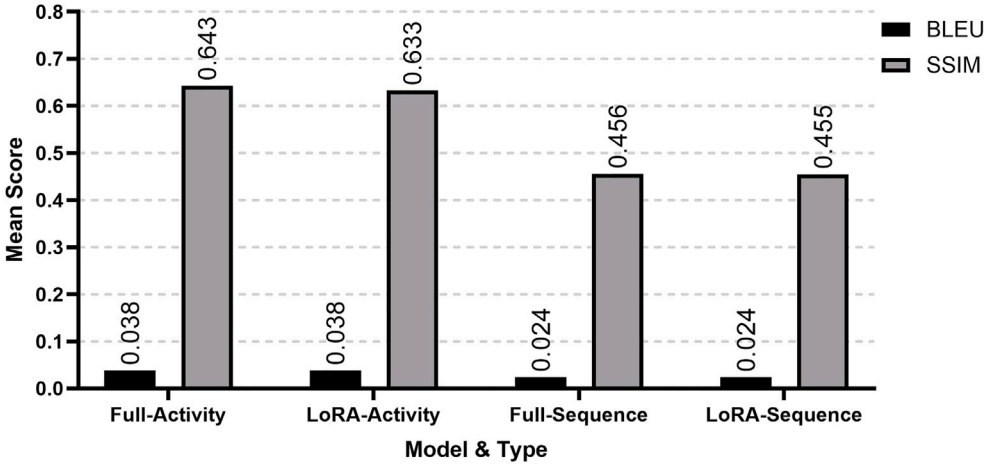


**Fig. 16.** Comparison of BLEU and SSIM scores across real-world activity and sequence diagrams. Results indicate that both full and LoRA fine-tuning yield limited improvements over the baseline, underscoring the importance of aligning synthetic training data with real-world visual styles.

**Table 4**
Training time metrics for activity datasets by model size: 7B model.

| Dataset size | Model type | Training time (h) | Train samples per sec | Train steps per sec | Train loss | Total FLOS |
|---|---|---|---|---|---|---|
| Small | Full | 0.161 | 10.335 | 0.081 | 2.020 | $8.721 \times 10^{13}$ |
| Small | LoRA | 0.252 | 6.626 | 0.052 | 2.022 | $9.731 \times 10^{13}$ |
| Medium | Full | 0.323 | 10.305 | 0.080 | 1.924 | $1.857 \times 10^{14}$ |
| Medium | LoRA | 0.507 | 6.578 | 0.051 | 1.924 | $2.066 \times 10^{14}$ |
| Large | Full | 0.658 | 10.127 | 0.079 | 1.830 | $3.880 \times 10^{14}$ |
| Large | LoRA | 1.038 | 6.423 | 0.050 | 1.800 | $4.326 \times 10^{14}$ |

**Table 5**
Training time metrics for activity datasets by model size: 13B model.

| Dataset size | Model type | Training time (h) | Train samples per sec | Train steps per sec | Train loss | Total FLOS |
|---|---|---|---|---|---|---|
| Small | Full | 0.284 | 5.862 | 0.046 | 1.945 | $1.090 \times 10^{14}$ |
| Small | LoRA | 0.432 | 3.858 | 0.030 | 1.953 | $1.217 \times 10^{14}$ |
| Medium | Full | 0.577 | 5.774 | 0.045 | 1.867 | $2.321 \times 10^{14}$ |
| Medium | LoRA | 0.891 | 3.743 | 0.029 | 1.871 | $2.583 \times 10^{14}$ |
| Large | Full | 1.180 | 5.650 | 0.044 | 1.777 | $4.851 \times 10^{14}$ |
| Large | LoRA | 1.784 | 3.736 | 0.029 | 1.769 | $5.408 \times 10^{14}$ |
| Extra Large | Full | 5.948 | 5.604 | 0.044 | 1.569 | $2.484 \times 10^{15}$ |
| Extra Large | LoRA | 9.002 | 3.703 | 0.029 | 1.553 | $2.772 \times 10^{15}$ |

**Table 6**
Training time metrics for sequence datasets by model size: 7B model.

| Dataset size | Model type | Training time (h) | Train samples per sec | Train steps per sec | Train loss | Total FLOS |
|---|---|---|---|---|---|---|
| Small | Full | 0.121 | 13.738 | 0.108 | 0.615 | $4.003 \times 10^{13}$ |
| Small | LoRA | 0.190 | 8.758 | 0.069 | 0.574 | $9.731 \times 10^{13}$ |
| Medium | Full | 0.240 | 13.864 | 0.108 | 0.502 | $8.121 \times 10^{13}$ |
| Medium | LoRA | 0.374 | 8.897 | 0.069 | 0.474 | $8.545 \times 10^{13}$ |
| Large | Full | 0.483 | 13.802 | 0.108 | 0.380 | $1.699 \times 10^{14}$ |
| Large | LoRA | 0.753 | 8.852 | 0.069 | 0.357 | $1.777 \times 10^{14}$ |

architecture in maintaining efficiency across different scales and types of datasets.

Activity datasets typically exhibit greater complexity, resulting in slower step rates than sequence datasets. For example, the 7B full model processes 10.335 samples per second on the small activity dataset versus 13.738 samples per second on the corresponding sequence dataset. Activity datasets require more operations per sample due to their richer feature sets, which slows processing rates and lowers throughput. However, the consistent step rates within each dataset type indicate that the architecture handles increasing dataset size without introducing performance bottlenecks.

Training time increases significantly with dataset size for both activity and sequence datasets, though activity datasets demand more time due to their complexity. For instance, training the 7B full model on the small activity dataset takes 0.161 h, compared to 0.121 h for the equivalent sequence dataset. This trend holds across all dataset sizes, with the 13B full model requiring 5.948 h for the extra-large activity dataset and 4.290 h for the corresponding sequence dataset. The additional time required for activity datasets stems from their slower step rates and lower throughput. For example, the 7B full model processes 10.335 samples per second for the small activity dataset but achieves 13.738 samples per second for the small sequence dataset. Despite

**Table 7**
Training time metrics for sequence datasets by model size: 13B model.

| Dataset size | Model type | Training time (h) | Train samples per sec | Train steps per sec | Train loss | Total FLOS |
|---|---|---|---|---|---|---|
| Small | Full | 0.214 | 7.805 | 0.061 | 0.544 | $5.005 \times 10^{13}$ |
| Small | LoRA | 0.334 | 4.995 | 0.039 | 0.515 | $5.285 \times 10^{13}$ |
| Medium | Full | 0.427 | 7.801 | 0.060 | 0.433 | $1.015 \times 10^{14}$ |
| Medium | LoRA | 0.642 | 5.189 | 0.040 | 0.417 | $1.068 \times 10^{14}$ |
| Large | Full | 0.950 | 7.819 | 0.061 | 0.324 | $2.124 \times 10^{14}$ |
| Large | LoRA | 1.313 | 5.075 | 0.040 | 0.314 | $2.222 \times 10^{14}$ |
| Extra Large | Full | 4.290 | 7.770 | 0.061 | 0.178 | $1.075 \times 10^{15}$ |
| Extra Large | LoRA | 6.514 | 5.117 | 0.040 | 0.171 | $1.123 \times 10^{15}$ |

these differences, the processing consistency within each dataset type showcases the scalability of the 7B and 13B models.

Activity datasets consistently require higher FLOS than sequence datasets of the same size. For instance, training the 7B full model on the small activity dataset involves $8.721 \times 10^{13}$ FLOS, compared to $4.003 \times 10^{13}$ FLOS for the equivalent sequence dataset. This disparity becomes more pronounced with larger datasets, as the 13B full model demands $2.484 \times 10^{15}$ FLOS for the extra-large activity dataset but only $1.075 \times 10^{15}$ FLOS for the sequence dataset. Sequence datasets are less computationally intensive, supporting faster convergence and reduced training time, making them advantageous for applications requiring quicker deployment. Training loss trends reveal that activity and sequence datasets yield lower loss values as dataset and model sizes grow. For instance, the 7B full model achieves a loss of 2.020 on the small activity dataset, while the equivalent sequence dataset achieves a lower loss of 0.615. Larger models and datasets amplify this pattern, with the 13B full model attaining a loss of 1.553 for the extra-large activity dataset and 0.178 for the corresponding sequence dataset. The lower loss values associated with sequence datasets reflect more efficient learning and potentially better generalization. Conversely, the higher loss values for activity datasets may challenge optimization but offer richer feature spaces. These features could capture more nuanced patterns, benefiting specific applications despite higher computational costs.

#### 3.2.2. Evaluation time

Fig. 17 illustrates the evaluation time for various models, including a non-fine-tuned baseline. Evaluation time primarily depends on the model's response length to image-prompt pairs and the total number of pairs rather than the inherent complexity of the data. For sequence models, evaluation time scales with dataset size, increasing from 2.708 h for the small dataset to 54.167 h for the extra-large dataset. This increase stems from the larger number of image-prompt pairs, requiring the model to generate more responses. The relatively straightforward structure of sequence diagrams results in concise outputs, helping to keep evaluation times relatively lower compared to activity models. In contrast, models fine-tuned on activity data demonstrate longer evaluation times due to the intricate nature of activity diagrams, which necessitate more detailed and extended responses. Evaluation time begins at 6.042 h for the small dataset and rises to 120.833 h for the extra-large dataset. The gap between sequence and activity models widens as dataset size increases, highlighting the greater computational demands of processing more complex diagram types.

The non-fine-tuned baseline model shows the longest evaluation times across all dataset sizes, starting at 9.167 h for the small dataset and reaching 183.333 h for the extra-large dataset. This model generates verbose and often inaccurate responses, extending evaluation time beyond fine-tuned models for sequence and activity data. These findings emphasize that evaluation time, rather than training time, becomes the primary bottleneck when working with large datasets. Optimizing evaluation processes — such as reducing unnecessary response lengths — is crucial for efficiently deploying models. Fine-tuning not only improves response accuracy but also curtails evaluation time.

### 3.3. Error analysis

The original 7B model exhibits the highest syntax error rates when generating sequence diagrams, particularly for small and medium datasets. Specifically, it produces rates of 0.343 for the small dataset and 0.522 for the medium dataset before stabilizing at 0.439 for larger datasets. In contrast, the full and LoRA-tuned versions of the 7B model show marked improvements, with the LoRA variant reducing syntax errors to 0.297. The 13B models follow a similar pattern but with consistently lower syntax error rates. The original 13B model starts at 0.133 for small datasets, with reductions observed as dataset size increases. Notably, the 13B LoRA model achieves the lowest syntax error rate, dropping to 0.031 for large datasets and 0.006 for extra-large datasets. This highlights the benefits of larger model sizes and fine-tuning strategies to reduce syntax errors effectively.

More complex activity diagrams initially exhibit higher syntax error rates, particularly for the baseline 7B model. The syntax error rate reaches 0.623 for small datasets, increasing slightly to 0.683 for larger datasets. However, fine-tuning mitigates these errors. The full 7B model achieves a syntax error rate of 0.012, and the LoRA-tuned 7B model follows closely with 0.019 for the small dataset. The 13B LoRA model again demonstrates the best performance, achieving an error rate of 0.006 for the extra-large dataset. These results indicate that fine-tuning, particularly LoRA-based strategies, is highly effective for complex activity diagrams (see Figs. 18 and 19).

The UML absence score measures instances where models fail to generate UML code. Table 8 shows that baseline models display slow but non-negligible absence rates, while fine-tuned models eliminate this timely For the small dataset, the baseline 7B model records a UML absence score of 0.0007 for both sequence and activity diagrams. The baseline 13B model avoids absence errors for small datasets but introduces minor errors for medium and large datasets. A notable exception occurs with the 7B model on the medium-sized sequence dataset, where the absence score increases to 0.0023.

The diagram mismatch score measures how often the generated diagram type differs from the prompt. Baseline models again demonstrate higher mismatch rates, particularly for activity diagrams. For example, the 13B baseline model records a mismatch score of 0.75 for small datasets, rising slightly to 0.7538 for large datasets. Sequence diagrams also show problems in baseline models, where the 7B model generates mismatches at a rate of 0.3567 for small datasets and 0.5254 for medium datasets. Fine-tuned models resolve this problem entirely, producing diagrams that consistently match the requested type. The reduction in mismatch scores highlights the effectiveness of fine-tuning to adhere to prompt instructions and improve the accuracy of the model output.

Comparing the performance of sequence and activity diagrams reveals that activity diagrams generally exhibit fewer syntax errors after fine-tuning. Baseline models perform better on sequence diagrams initially, as their simpler structure reduces the likelihood of errors. However, sequence diagrams are more susceptible to corrupted outputs caused by syntax errors, often producing black or green error images. These corrupted outputs degrade the SSIM scores significantly, even when the BLEU scores remain relatively high. In contrast, fine-tuned models excel at handling the complexity of activity diagrams. The

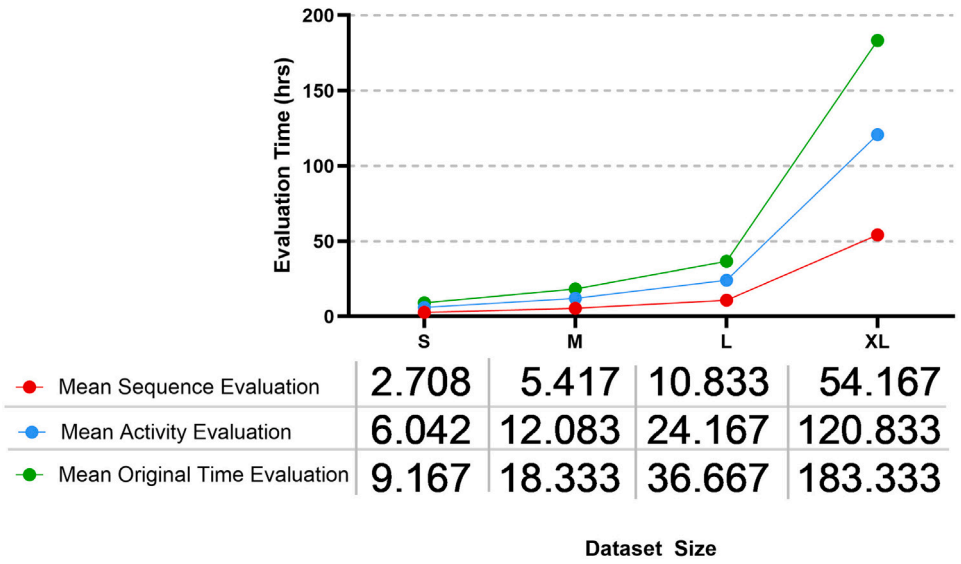


**Fig. 17.** Evaluation time across various model configurations and dataset conditions, illustrating different setups' computational efficiency and response behavior.

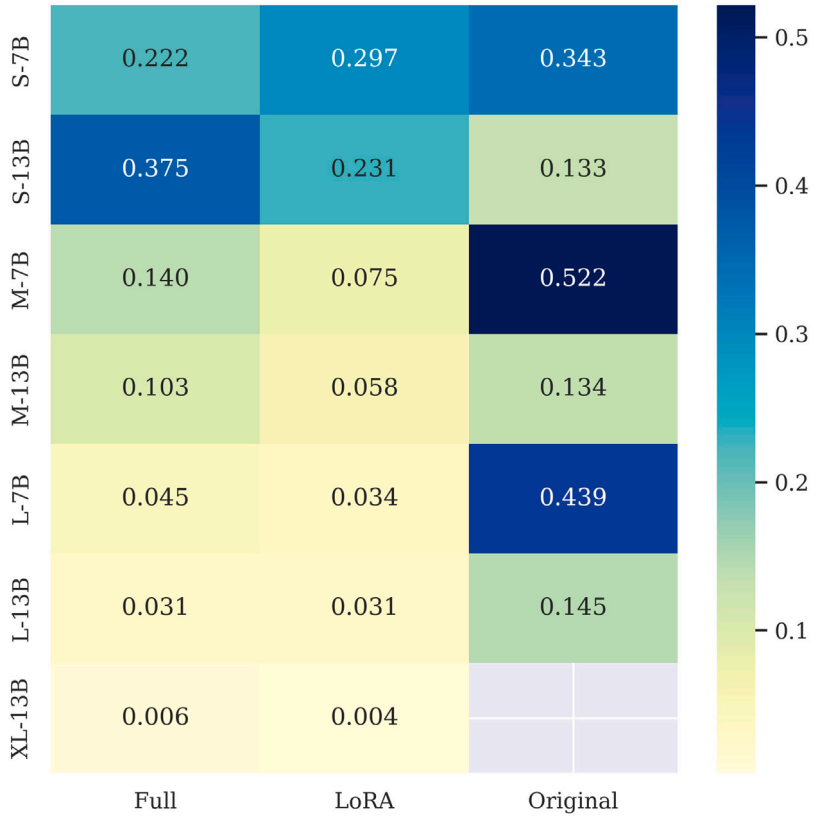


**Fig. 18.** Syntax error rates for sequence diagrams across all dataset sizes and model configurations. Darker shades represent higher error frequencies.

**Table 8**
UML absence and diagram mismatch scores by dataset size and model configuration.

| Dataset | Model size | Sequence absence | Activity absence | Sequence mismatch | Activity mismatch |
|---|---|---|---|---|---|
| Small | 7B | 0.0007 | 0.0007 | 0.3567 | 0.3633 |
| Small | 13B | 0 | 0 | 0.526 | 0.75 |
| Medium | 7B | 0.0023 | 0.001 | 0.5254 | 0.367 |
| Medium | 13B | 0.0003 | 0.0007 | 0.4693 | 0.7687 |
| Large | 7B | 0.001 | 0.0003 | 0.3141 | 0.3433 |
| Large | 13B | 0.0003 | 0 | 0.56 | 0.7538 |

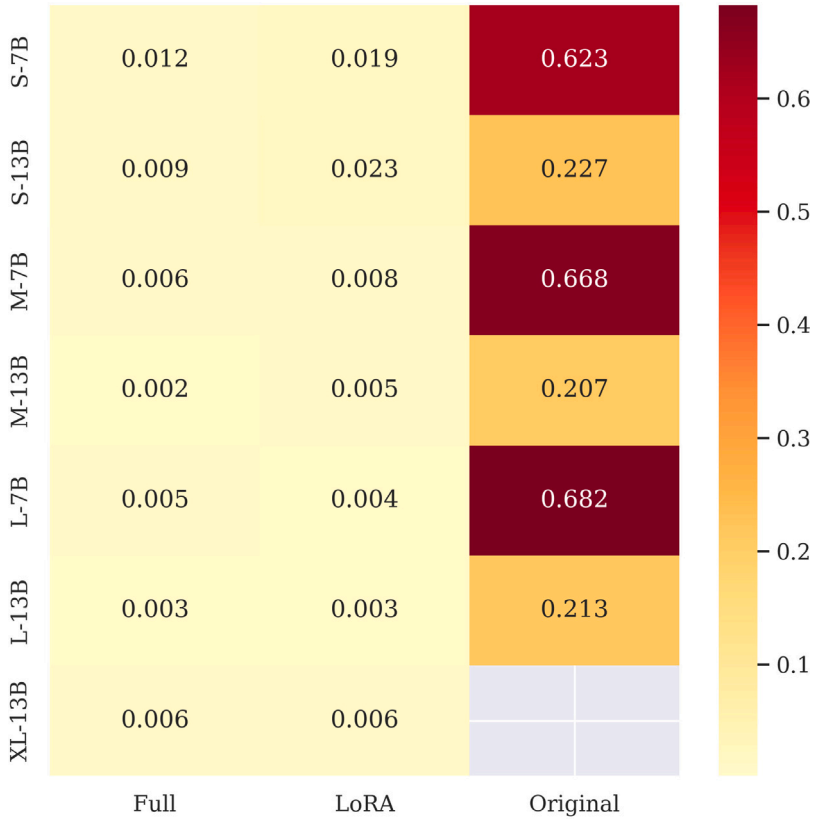


**Fig. 19.** Syntax error rates for activity diagrams across all dataset sizes and model configurations. Darker shades represent higher error frequencies.

improvements observed in syntax error rates and diagram mismatch scores demonstrate the effectiveness of larger datasets and fine-tuning techniques. The 13B LoRA-tuned model achieves the lowest error rates overall, reinforcing the importance of both model size and training strategy in improving UML diagram generation accuracy.

### 3.4. Result overview

The experiments confirmed that larger models and datasets improve BLEU and SSIM metrics, reduce syntax errors, and enhance diagram accuracy. Larger models, particularly the 13B configurations, outperformed smaller models across most metrics, demonstrating lower syntax error rates and greater fidelity to ground truth diagrams. Sequence diagrams achieved higher BLEU scores, but their SSIM scores suffered due to syntax errors that degraded visual fidelity. Being more complex, activity diagrams benefited significantly from fine-tuning, exhibiting reduced syntax errors and improved image quality. Fine-tuned models trained on larger datasets consistently demonstrated better generalization and alignment with reference diagrams. Fine-tuning eliminated UML absence errors and reduced diagram mismatches, with only baseline models displaying non-zero UML absence scores.

LoRA fine-tuning yielded results comparable to full fine-tuning but revealed trade-offs related to hardware constraints. While LoRA fine-tuning significantly reduces the number of trainable parameters and lowers memory consumption, our findings highlight an important trade-off: increased training time and higher total FLOPs compared to full fine-tuning. Despite requiring fewer GPUs (two versus four for full fine-tuning), LoRA exhibited slower step rates and longer convergence times. This was largely due to hardware and batch size constraints. The LLaVA training pipeline is designed to maintain a consistent global batch size. To match the original global batch size of 128 (as used in the official 8-GPU setup), we configured LoRA training on two GPUs with a per-device batch size of 16 and four accumulation steps. In contrast, standard fine-tuning used four GPUs with a batch size of 8 per device and the same accumulation steps. While the global batch size was preserved, using fewer GPUs with larger per-device batches increased memory pressure and reduced per-step efficiency. For example, across the 13B model on extra-large activity datasets, LoRA fine-tuning consumed approximately $2.772 \times 10^{15}$ FLOS over 9 h, compared to $2.48 \times 10^{15}$ FLOS in 5.95 h for full fine-tuning. This suggests that LoRA's efficiency gains in memory and GPU requirements come at the cost of computational throughput—likely due to differences in batch size and limited GPU resources. However, the flexibility it offers — especially for resource-constrained setups — often outweighs the extended runtime, making it a valuable strategy for scalable domain adaptation.

Although real-world data was not used during training, we evaluated model generalization on a small set of naturally sourced UML diagrams. The results, while showing small improvements with training, revealed a notable performance gap compared to synthetic test data. The discrepancy is likely due to differences in naming structure and semantic coherence, as the synthetic data was generated using randomized labels. Nonetheless, the experiment demonstrates the feasibility of using synthetic data for this task and highlights opportunities for improving generalization. The extensive synthetic datasets reveal that, regardless of other factors, with sufficient data the models can effectively learn to represent UML code from images.

## 4. Discussion

This study investigates the role of the LLaVA-1.5 model in automating UML diagram-to-code generation, expanding on prior work by Conrardy and Cabot. While their research prioritizes syntactic precision and prompt design for models like GPT-4 and CogVLM (Conrardy & Cabot, 2024; Wang et al., 2024), our broader approach asses both structural fidelity and computational efficiency. By incorporating BLEU

and SSIM metrics, we evaluate textual and visual accuracy, offering a more comprehensive performance assessment across various datasets and model configurations. Results emphasize the critical role of dataset scale and fine-tuning strategies. Larger LoRA-tuned models, particularly the 13B configurations, consistently outperform baseline models by achieving higher BLEU and SSIM scores. These models excel at generating structurally accurate UML diagrams, effectively reducing syntax errors and mismatches. Fine-tuning, especially with LoRA, demonstrates how targeted parameter updates enhance performance without overwhelming computational costs, making advanced models accessible to resource-limited environments.

Compared to Conrardy and Cabot's focus on prompt refinement, our study addresses computational scalability, a key factor in adoption. While they achieved impressive results with structural accuracy for class diagrams, we highlight the importance of scalability and model adaptability for complex behavioral diagrams like sequence and activity workflows. Both studies demonstrate the growing potential of multimodal models in software engineering. Still, our inclusion of error analysis — quantifying mismatches and syntax issues — offers deeper insights into areas where models can improve further. Our research also reveals that model misinterpretations, such as defaulting to incorrect diagram types, are more prevalent in smaller datasets. However, fine-tuned models correct these issues by adhering more closely to prompts and minimizing mismatches. These findings demonstrate that larger datasets and well-tuned models enable MM-LLMs to handle dynamic workflows effectively. By addressing key challenges such as structural errors, prompt misinterpretations, and computational bottlenecks, our work sets the stage for scalable and efficient UML diagram automation.

### 4.1. Avenues for further study

The BLEU metric has been a foundational tool for evaluating code generation; however, its reliance on syntactic $n$-gram matching may limit its applicability for UML tasks, where visual fidelity and structural relationships are also critical. CodeBLEU extends BLEU by incorporating abstract syntax tree analysis and data flow consistency, aligning evaluations more closely with functional correctness. However, its current design focuses on traditional programming languages such as Python or C (Ren et al., 2020). Exploring how similar techniques might be adapted for UML tasks could open avenues for aligning evaluations with textual and visual aspects. Furthermore, approaches such as pass@k, which evaluates correctness across multiple outputs, and frameworks such as EvalPlus, which uses mutation-based testing to identify potential model vulnerabilities (Liu et al., 2023), present interesting opportunities for advancing UML evaluation methods by integrating textual and diagrammatic assessments.

Emerging multimodal models such as LLaVA-NeXT (Liu, Li, Li, Li et al., 2024) and LLaVA-OneVision (Li et al., 2024), which incorporate advanced OCR, multi-resolution processing, and cross-modal knowledge transfer, offer promising directions for improving UML automation by better handling various types of diagram and maintaining accuracy at varying resolutions. While our experiments centered on LLaVA-based models, we acknowledge the importance of incorporating additional benchmark models to broaden the comparative landscape. Our primary objective was to demonstrate the feasibility of using a multimodal LLM for large-scale UML code generation. Within this scope, LLaVA-1.5 at both 7B and 13B scales proved sufficient for validating our approach. Nevertheless, including models such as Deepseek R1 (DeepSeek-AI et al., 2025), Claude, or CogVLM in future work would offer a more comprehensive perspective on architectural differences, visual reasoning capabilities, and generalization.

Our models achieved strong performance on synthetic datasets, gains on more realistic UML samples were notably more modest. As illustrated in Figs. 14 and 15, fine-tuned models demonstrated only marginal improvements over the baseline when evaluated on real-world diagrams. This discrepancy stems from synthetic diagrams' structure, which fail to fully capture the artifacts commonly found in typical UML diagrams. To bridge the realism gap, future iterations of our synthetic dataset could categorize diagrams into common application domains such as banking, e-commerce, authentication workflows, and web development. The thematic structuring would align synthetic diagrams more closely with real-world use cases, thereby improving their semantic fidelity. Additionally, incorporating domain-specific terminology and realistic naming conventions within these categories would further enhance the authenticity of the synthetic data. Nevertheless, our results strongly suggest that synthetic data can serve as a powerful training signal with sufficient volume and variation: MM-LLMs exhibit a clear capacity to learn to accurately reconstruct UML code from images. The result highlight a promising direction—by further enhancing the visual realism and variability of synthetic training data, future models may bridge the domain gap and achieve high-fidelity UML code generation even in real-world scenarios.

## 5. Conclusion

This study introduces a scalable and efficient framework to automate UML diagram-to-code generation, directly addressing a critical need in software engineering. We demonstrate how fine-tuned multimodal large language models adapt to dynamic workflows and event-driven processes by focusing on sequence and activity diagrams. Integrating LoRA fine-tuning improves accuracy while reducing computational demands, making the approach accessible for resource-constrained environments. Our dual-metric evaluation framework combines BLEU for textual accuracy and SSIM for visual fidelity, creating a robust benchmark for assessing UML generation models. This approach ensures that generated outputs maintain structural precision and syntactic correctness, bridging the gap between diagrammatic design and machine-readable code. Fine-tuned models consistently outperform baseline versions by achieving higher fidelity and reducing structural misalignment errors.

The study highlights opportunities for future exploration, such as modifying advanced metrics like CodeBLEU and pass@k, which offer deeper insights into functional correctness, and leveraging state-of-the-art models like LLaVA-NeXT and LLaVA-OneVision to enhance reasoning and scalability. Expanding datasets to include additional UML types, such as class diagrams, and incorporating synthetic and real-world examples will strengthen model generalization and robustness. These advancements will refine UML automation tools that align with the evolving requirements of modern software engineering workflows. Our contributions demonstrate the potential of intelligent, scalable AI systems to automate complex design tasks and bridge theoretical advances with practical applications. The framework offers software engineering solutions and broader fields requiring structured and dynamic representations, including healthcare, education, and industrial automation. Establishing a strong foundation for AI-driven design and implementation creates new opportunities for innovation across diverse industries, advancing intelligent system integration and automation.

## CRediT authorship contribution statement

**Averi Bates:** Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Ryan Vavricka:** Data curation, Software, Writing – review & editing. **Shane Carleton:** Conceptualization, Supervision, Project administration, Funding acquisition. **Ruosi Shao:** Writing – review & editing. **Chongle Pan:** Conceptualization, Supervision, Project administration, Writing – review & editing.

## Funding statement

This project was funded by the U.S. National Science Foundation under the Accelerating Research Translation program 2331409 and the U.S. Air Force Small Business Innovation Research program.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Shane Carleton reports financial support was provided by US Department of the Air Force. Ryan Vavricka reports financial support was provided by MapLarge. Shane Carleton reports a relationship with MapLarge that includes: employment. Averi Bates has patent issued to University of Oklahoma. Shane Carleton has patent issued to University of Oklahoma. Chongle Pan has patent issued to University of Oklahoma. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

The authors thank the University of Oklahoma's Supercomputing Center for Education & Research for providing access to their supercomputing resources. We also thank the Data Institute for Societal Challenges for facilitating GPU resources through OSCER's infrastructure. In addition, we acknowledge AFWERX for their support and contributions. Finally, the authors extend their deepest gratitude to Dr. Andrew H. Fagg for his invaluable feedback on the thesis that formed the foundation of this work. His guidance were helped shape the direction and quality of the research.

## Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.mlwa.2025.100660.

## Data availability

The training and testing UML data supporting the findings of this study are available and can be found at https://doi.org/10.5281/zenodo.15103682. Please note that this excludes any associated prompts. Access to the data will be granted for non-commercial research purposes.

## References

Beltramelli, T. (2018). Pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems* (pp. 1–6).

Booch, G., Rumbaugh, J. E., & Jacobson, I. (1998). The unified modeling language user guide. *Journal of Database Management*, *10*, 51–52, URL: https://api.semanticscholar.org/CorpusID:27636970, [Online]. Available: https://api.semanticscholar.org/CorpusID:27636970 (Accessed 28 June 2024).

Cai, B., Luo, J., & Feng, Z. (2023). A novel code generator for graphical user interfaces. *Scientific Reports*, *13*, URL: https://api.semanticscholar.org/CorpusID:265349539, [Online]. Available: https://api.semanticscholar.org/CorpusID:265349539 (Accessed 05 September 2024).

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ...., Ryder, N. et al. (2021). Evaluating large language models trained on code. arXiv:2107.03374, URL: https://arxiv.org/abs/2107.03374, [Online]. Available: https://arxiv.org/abs/2107.03374, (Accessed 15 July 2024).

Chen, F., Zhang, L., Lian, X., & Niu, N. (2022). Automatically recognizing the semantic elements from UML class diagram images. *Journal of Systems and Software*, *193*, Article 111431. http://dx.doi.org/10.1016/j.jss.2022.111431, URL: https://www.sciencedirect.com/science/article/pii/S0164121222001340.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., ...., Valter, D. et al. (2022). Scaling instruction-finetuned language models. arXiv:2210.11416v5, URL: https://arxiv.org/abs/2210.11416v2, [Online]. Available: https://arxiv.org/abs/2210.11416v2 (Accessed 30 September 2024).

Conrardy, A., & Cabot, J. (2024). From image to UML: First results of image based UML diagram generation using LLMs. arXiv:2404.11376v2, URL: https://arxiv.org/abs/2404.11376v2, [Online]. Available: https://arxiv.org/abs/2404.11376v2 (Accessed 03 August 2024).

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., ...., Liu, A. et al. (2025). DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. arXiv:2501.12948, URL: https://arxiv.org/abs/2501.12948, [Online]. Available: https://arxiv.org/abs/2501.12948 (Accessed 17 March 2025).

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, [Online]. Available: https://arxiv.org/abs/1810.04805 (Accessed 05 July 2024).

Ellis, K., Ritchie, D., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning to infer graphics programs from hand-drawn images. *Advances in Neural Information Processing Systems*, *31*.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. arXiv:2106.09685v2, URL: https://arxiv.org/abs/2106.09685v2, [Online]. Available: https://arxiv.org/abs/2106.09685v2 (Accessed 02 June 2024)].

Koç, H., Erdoğan, A. M., Barjakly, Y., & Peker, S. (2021). UML diagrams in software engineering research: A systematic literature review. *Proceedings*, *74*(1), http://dx.doi.org/10.3390/proceedings2021074013, URL: https://www.mdpi.com/2504-3900/74/1/13.

Li, B., Zhang, Y., Guo, D., Zhang, R., Li, F., Zhang, H., Zhang, K., Zhang, P., Li, Y., Liu, Z., & Li, C. (2024). LLaVA-OneVision: Easy visual task transfer. arXiv:2408.03326v3, [Online]. Available: https://arxiv.org/abs/2408.03326v3 (Accessed 24 July 2024)].

Liu, H., Li, C., Li, Y., & Lee, Y. J. (2024). Improved baselines with visual instruction tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 26296–26306).

Liu, H., Li, C., Li, Y., Li, B., Zhang, Y., Shen, S., & Lee, Y. J. (2024). LLaVA-NeXT: Improved reasoning, OCR, and world knowledge. URL: https://llava-vl.github.io/blog/2024-01-30-llava-next/, [Online]. Available: https://llava-vl.github.io/blog/2024-01-30-llava-next/ (Accessed 29 June 2024).

Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2024). Visual instruction tuning. *Advances in Neural Information Processing Systems*, *36*.

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *vol. 36*, *Advances in neural information processing systems* (pp. 21558–21572). Curran Associates, Inc., URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf (Accessed 25 August 2024).

LMSYS (2023). Vicuna: An open-source chatbot impressing GPT-4 with 90% ChatGPT quality. URL: https://lmsys.org/blog/2023-03-30-vicuna/, [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/ (Accessed 07 July 2024).

Lu, Y., & Alexandru, C. A. (2023). Systematic review of UML diagramming software tools for higher education software engineering courses. In *Proceedings of the 2023 conference on united kingdom & ireland computing education research*. URL: https://api.semanticscholar.org/CorpusID:262487030, [Online]. Available: https://api.semanticscholar.org/CorpusID:262487030 (Accessed 20 August 2024).

Lu, S., Jiao, J., Wang, L., Qiu, H., Lin, X., Mei, H., & Li, H. (2024). Video class-incremental learning with clip based transformer. In *2024 IEEE international conference on image processing* (pp. 500–506). http://dx.doi.org/10.1109/ICIP51287.2024.10647787.

Metzner, A. (2024). Systematic teaching of UML and behavioral diagrams. In *2024 36th international conference on software engineering education and training* (pp. 1–5). IEEE.

Murali, V., Maddila, C., Ahmad, I., Bolin, M., Cheng, D., Ghorbani, N., Fernandez, R., Nagappan, N., & Rigby, P. C. (2024). AI-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proceedings of the ACM on Software Engineering*, *1*(FSE), http://dx.doi.org/10.1145/3643774.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics* (pp. 311–318). USA: Association for Computational Linguistics, http://dx.doi.org/10.3115/1073083.1073135.

PlantUML (2025). PlantUML language reference guide. URL: https://plantuml.com/guide, Version 1.2025.0.

Radford, A., Kim, J. W., Hallacy, A., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. arXiv preprint arXiv:2103.00020, [Online]. Available: https://arxiv.org/abs/2103.00020 (Accessed 04 September 2024).

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). *Improving language understanding by generative pre-training*. OpenAI.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*(140), 1–67.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., & Ma, S. (2020). CodeBLEU: a method for automatic evaluation of code synthesis. arXiv:2009.10297v2, URL: https://arxiv.org/abs/2009.10297v2, [Online]. Available: https://arxiv.org/abs/2009.10297v2 (Accessed 16 June 2024).

Reznik, Y. A. (2023). Another look at SSIM image quality metric. In *Image quality and system performance* (pp. 305–312). URL: https://api.semanticscholar.org/CorpusID:252691913.

Shi, C., Yang, C., Liu, Y., Shui, B., Wang, J., Jing, M., Xu, L., Zhu, X., Li, S., Zhang, Y., Liu, G., Nie, X., Cai, D., & Yang, Y. (2024). ChartMimic: Evaluating LMM's cross-modal reasoning capability via chart-to-code generation. arXiv:2406.09961, URL: https://arxiv.org/abs/2406.09961, [Online]. Available: https://arxiv.org/abs/2406.09961 (Accessed 30 September 2024)].

Wang, Z., Bovik, A., Sheikh, H., & Simoncelli, E. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612. http://dx.doi.org/10.1109/TIP.2003.819861.

Wang, W., Lv, Q., Yu, W., Hong, W., Qi, J., Wang, Y., Ji, J., Yang, Z., Zhao, L., Song, X., Xu, J., Xu, B., Li, J., Dong, Y., Ding, M., & Tang, J. (2024). CogVLM: Visual expert for pretrained language models. arXiv:2311.03079v2, URL: https://arxiv.org/abs/2311.03079v2, [Online]. Available: https://arxiv.org/abs/2311.03079v2 (Accessed 20 July 2024).

Web Dev Tutor (2023a). Beginner's guide to creating PlantUML sequence diagrams: Learn the basics with examples. URL: https://www.webdevtutor.net/blog/beginner-guide-plantuml-sequence-diagrams-with-examples, [Online]; (Accessed 17 March 2025.

Web Dev Tutor (2023b). A comprehensive guide to PlantUML activity diagrams: Everything you need to know. URL: https://www.webdevtutor.net/blog/comprehensive-guide-plantuml-activity-diagrams [Online] (Accessed 17 March 2025).

Wu, C., Ge, Y., Guo, Q., Wang, J., Liang, Z., Lu, Z., Shan, Y., & Luo, P. (2024). Plot2Code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. arXiv:2405.07990, URL: https://arxiv.org/abs/2405.07990, [Online]. Available: https://arxiv.org/abs/2405.07990 (Accessed 17 August 2024).

Zhang, S., Chen, Z., Shen, Y., Ding, M., Tenenbaum, J. B., & Gan, C. (2023). Planning with large language models for code generation. arXiv:2303.05510, URL: https://arxiv.org/abs/2303.05510, [Online]. Available: https://arxiv.org/abs/2303.05510 (Accessed 02 August 2024).