

Comparative Analysis of Large Language Models for Automated Use Case Diagram Generation

Nikhil Krishnan G S^a, Ambadi S^b and M G Thushara^c

Department of Computer Science and Applications, Amrita School of Computing, Amritapuri, India

Keywords: UML, Use Case Diagram, LLM, Automating UML, Automation.

Abstract: This study explores the use of Large Language Models in automating the process of generating use case diagrams from software requirements written in natural language, ensuring both syntactic correctness and semantic accuracy. The proposed methodology involves selecting a few prominent LLMs, preparing standardized inputs, and assessing outputs based on their syntactic and semantic correctness, and relationship mapping. Models were compared using a rigorous error analysis framework to identify strengths and limitations of the models. Among the tested models, gemma2 achieved the best average performance. This research contributes to advancing automated requirements processing, offering a scalable solution for software engineering workflows.

1 INTRODUCTION


Requirements engineering is the very first step in software development which defines both the functional and non-functional specifications of a software system. These specifications are usually documented in a Software Requirements Specifications (SRS) document. This document helps to translate stakeholder expectations into the technical implementation of the software and guides the development, testing, and validation processes. In the requirements, use case diagrams play an important role. It visually represents the interaction between users (actors) and the system (use cases), making the requirements clear to both technical and non-technical stakeholders and conveying the software system's expected behavior across development teams.


Manually creating these use case diagrams is a time-consuming task, particularly for complex software systems with multiple actors and use cases (Elallaoui et al., 2018). This often leads to inconsistency issues, as high skill and attention to detail are required to maintain uniformity in notations and structures in all the diagrams (Nair and Thushara, 2024a). In addition, manually created diagrams are prone to human errors, such as neglected relationships or mis-


interpreted use cases, which will affect the quality of the SRS document (Nair and Thushara, 2024b). Updating the diagrams to add any new change in requirement is a labor-intensive task and can introduce further inconsistencies leading to scalability issues. Due to these drawbacks with the manual creation of diagrams, automating the process is gaining significant attention, to streamline development and reduce error (Elallaoui et al., 2018).

Large Language Models (LLMs) are a good tool for automating the generation of use case diagrams from natural language specifications (Alessio et al., 2024) (Jeong, 2024). PlantUML compiler allows for the generation of UML diagrams from PlantUML code. But writing PlantUML code manually is a time-consuming and error-prone process (Alessio et al., 2024). LLMs when trained on specific datasets containing both text and code, become capable of generating PlantUML syntax from functional requirements (Soudani et al., 2024). However, current LLMs, especially general-purpose ones, often produce output with syntactic or semantic errors, necessitating manual corrections for precise executable code (Xie et al., 2023).

This study is motivated by the need for an automated and accurate generation of use case diagrams from functional requirements, with minimal manual intervention. By evaluating multiple base LLMs, this research aims to identify which models perform best

^a  <https://orcid.org/0009-0001-8538-2372>

^b  <https://orcid.org/0009-0003-3839-5490>

^c  <https://orcid.org/0000-0002-8325-1491>

in generating error-free PlantUML code for use case diagrams. This analysis will focus on error classification, particularly syntactic and semantic errors, and determine the suitability of each model for further fine-tuning. Identifying an optimal base model with fewer initial errors will reduce the effort required in post-generation correction and enhance the viability of automated UML generation in both educational and software development contexts.

The paper is structured as follows: Section II reviews related work in requirements engineering and automated UML generation, focusing on advances in LLM-driven code generation. Section III details the methodology, including environment setup, model selection, input data preparation, and evaluation metrics. Section IV presents the results, analyzing model performance based on error rates and code quality. Section V explains how the models were evaluated and the criteria that led to determining the best LLM model. Finally, Section VI concludes with insights from this evaluation and discusses potential directions for future research, including model fine-tuning for improved accuracy in UML generation.

2 RELATED WORKS

2.1 Automated Requirements Processing

The process of automating the generation of UML diagrams and other structural formats (Vemuri et al., 2017) from software requirements has been explored before. Previous works utilized rule-based and NLP-based methods to interpret the requirements (Veena et al., 2018) (Veena et al., 2019). These approaches relied on syntactic and semantic parsing to identify the actors and use cases in the functional requirements. But the results generated often require manual corrections (Nair and Thushara, 2024b) (Nair and Thushara, 2024a). More recent approaches leverage machine learning models, but this requires substantial domain-specific training data to generate results with accuracy (Ahmed et al., 2022). Transforming UML diagrams from one type to another has also been investigated, such as the work on deriving activity diagrams from Java execution traces (Devi Sree and Swaminathan, 2018) and transforming sequence diagrams into activity diagrams (Kulkarni and Srinivasa, 2021).

2.2 Use of Large Language Models in Software Engineering

Studies (Alessio et al., 2024) have shown that LLMs can translate descriptions in natural language into code snippets and other meaningful structural representations. Current LLMs like GPT and Codex struggle with syntax accuracy and specific domain requirements, often generating inconsistent results (Xie et al., 2023). Research in this domain shows a need for fine-tuning LLMs to generate reliable results containing fewer errors (Jeong, 2024). Additionally, frameworks leveraging retrieval-augmented generation (RAG)-based approaches have been applied to teaching UML diagram generation effectively (Ardimento et al., 2024). Recent technical advancements also highlight efficient LLM models that are capable of functioning on edge devices (Abdin et al., 2024).

2.3 Error Analysis in LLM-Generated Code

Identifying errors and correcting them is a critical step in evaluating the outputs generated by LLMs for software engineering applications. Earlier studies have identified some common error types, such as inconsistent semantics and misinterpreted requirements. Correction after generating results and fine-tuning in the specific domain are two techniques that have been proposed to address these issues. Recent research demonstrates how graph transformation approaches (Anjali et al., 2019) can model and analyze UML diagrams effectively (Hachichi, 2022). Furthermore, evaluating the effectiveness of LLMs in generating UML diagrams has been explored, with notable efforts in class diagram generation for comparative analysis (De Bari, 2024).

There is limited research on comparing LLMs specifically for PlantUML syntax generation. This gap stresses the need for comprehensive error analysis to determine the most suitable model for automated use case diagram generation.

ROUGE and BLUE are two conventional reference-based metrics for evaluating LLMs. These metrics are not preferred in this research as they have relatively low correlation with human judgements, especially in open-ended generation tasks. While these metrics are good for evaluating short and generic outputs, they fail to evaluate coherence and relevance which are critical for human-like judgements. One solution to overcome these shortcomings is to use LLM as a judge for evaluation. GEval

(Liu et al., 2023) is a framework that uses LLMs with chain-of-thoughts to evaluate the quality of LLM-generated results. By taking natural language instruction that defines the evaluation criteria as prompt, G-Eval uses an LLM to generate a chain-of-thoughts of detailed evaluation steps. The prompt along with the generated chain-of-thoughts is used to evaluate the results.

2.4 Summary and Research Gap

While prior work has explored LLMs in requirements processing and code generation, few studies have focused on systematically comparing models for use case diagram generation with PlantUML syntax (Alessio et al., 2024). Our study addresses this gap by evaluating multiple lightweight LLMs (low parameter LLMs) installed in local machines to identify those with the lowest error rates, aiming to reduce the need for manual corrections and enhance the automation potential of UML generation tools in future works.

3 METHODOLOGY

This methodology aims to systematically evaluate and compare the effectiveness of different Large Language Models (LLMs) in generating accurate PlantUML syntax directly from natural language requirements. Given the increasing use of LLMs in automating coding and diagramming tasks, understanding their strengths and limitations in this specific context is essential. This analysis seeks to identify which models produce syntactically correct and semantically meaningful use case diagrams, with minimal errors, thus reducing the need for manual corrections. By conducting a comparative analysis, we aim to determine the most suitable LLM for automating use case diagram generation, particularly in terms of accuracy, efficiency, and reliability.

The methodology is structured into key stages as shown in figure 1: model selection, data preparation, input specification, error classification, and evaluation and analysis. Each stage plays a vital role in ensuring a comprehensive and fair comparison of the models. This structured approach helps in identifying the best-performing model and highlights common areas where improvement is required.

3.1 Model Selection

The very first step in the methodology was to select a set of Large Language Models for evaluation.

We chose the models based on several factors including their capability in generating structured code outputs, understanding natural language inputs, and accessibility. The focus was on selecting models that were small in size, had smaller parameters, and were resource-efficient so that the models could be run in local machines. Each one of the selected models has varied parameters and architecture, which allows us to observe the differences in code quality, syntax accuracy, and reliability across diverse configurations. This selection process aims to ensure a well-rounded comparison of lightweight base model LLMs, to determine which could best translate natural language requirements into accurate PlantUML syntax.

3.2 Data Preparation

To assess each model's performance, a functional requirements dataset that could efficiently test each model's capabilities had to be prepared. We prepared a set of specifications based on common use case scenarios, including data processing, user administration, and e-commerce. The requirements were annotated wherever possible to emphasize specific components such as actors, actions and system interactions. This gives the models a structured input. We pre-processed the data before inputting it into the models to make the wording more uniform and eliminate ambiguities in requirements caused by any technical jargon present in the data, which the model could potentially misinterpret. Because of this uniform formatting, it became easier to evaluate each model's answer correctness.

3.3 Input Specification

The LLM models receive a CSV file as input containing ten software functional requirements, each representing a software system. These functional requirements are added as plain text with a standardized prompt designed to guide the model in generating PlantUML code for use case diagrams. The prompt contains the desired output format, including the proper semantics of "include" and "extend" relationships and specifies that the output should contain PlantUML code with valid syntax, without any additional explanations or comments that could cause syntax errors. The same prompt is used for every LLM model and input set to guarantee uniformity in input interpretation.

3.4 Evaluation Metrics

Answer correctness metric is used for evaluation. It is a numerical score between 0 and 1, 0 being the least

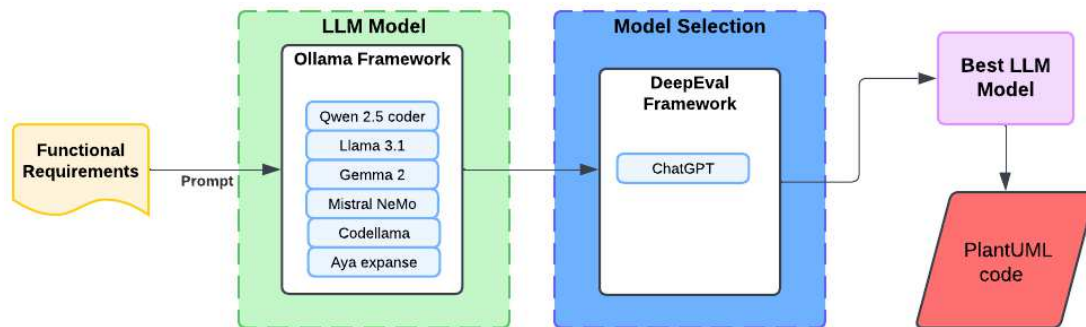


Figure 1: Structural Flow of the Proposed Model.

Table 1: Overview of Selected LLM Models for Use Case Diagram Generation.

Model	Parameters	Description
Qwen 2.5-coder	7B	Qwen2.5-Coder (Hui et al., 2024) is an advanced version of the Qwen series of large language models (formerly CodeQwen), offering notable improvements in code generation, reasoning, and bug fixing over its predecessor, CodeQwen1.5. Built on the Qwen2.5 foundation, it is trained on a vast dataset of 5.5 trillion tokens, which includes source code, text-code grounding, and synthetic data. It also supports long-context inputs of up to 128K tokens.
Gemma	9B	Google’s lightweight Gemma2 (Team et al., 2024) models build on Gemini research, with enhanced capabilities in language understanding, suitable for tasks like question-answering, summarization, and reasoning.
Llama 3.1	8B	Meta’s Llama 3 (Dubey et al., 2024) improves over Llama 2 in coherence and contextual understanding, excelling in creative writing, coding, and conversational AI. Llama 3.1 is an upgraded version that improves reasoning and context size.
Code Llama	7B	Code Llama (Roziere et al., 2024) is a code generation model built on Llama 2, designed to enhance developer workflows and assist in learning programming. It can generate both code and natural language explanations, supporting popular programming languages like Python, C++, Java, PHP, Typescript, C#, and Bash.
Mistral NeMo	12B	A 12B model by Mistral AI with NVIDIA, NeMo offers a 128k token context window, supporting multilingual applications in English, French, German, and more. The model is optimized for function calling and is similar to Mistral 7B.
Aya Expanse	8B	Aya Expanse, developed by Cohere For AI, is a family of multilingual large language models designed to close language gaps and improve global communication. It supports over 23 languages, including Arabic, Chinese, English, Hindi, and Spanish. The models use advanced techniques like supervised fine-tuning, multilingual preference training, and model merging to enhance performance across various linguistic tasks.

correct and 1 being the most correct answer. The correctness metric involves comparing the LLM’s result with the expected result based on any custom evaluation criteria defined by the user. The evaluation criteria involves:

- **Syntax Correctness:** This criteria checks if the generated PlantUML code has syntax errors and if it will compile successfully.
- **Comments and Descriptions:** Any additional comments or descriptions included with the PlantUML code will cause a syntax error. This criteria checks if there are any comments or descriptions in the result.
- **Actor Count:** To confirm that there are the same number of actors in the expected and actual out-

puts.

- **Use Case Correspondence:** This criterion ensures that the generated output matches the expected output.
- **Extend and Include Relationships:** To confirm that the relationships are accurately implemented and positioned in the diagram.
- **Hallucination Detection:** To look for any extra created content that does not fit the specified functional requirements.

These metrics provided a balanced assessment framework, helping to capture both the precision and usability of each model’s output.

4 RESULTS

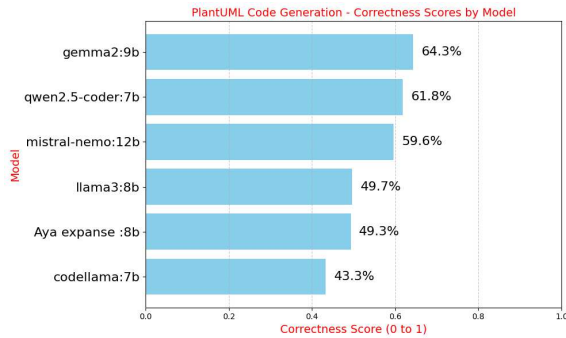


Figure 2: Evaluation results.

The identical set of carefully chosen requirements was fed into each model for testing, and the output produced by PlantUML was compared to our pre-determined metrics. We ensured that every output complied with PlantUML requirements by using automated methods to verify syntax accuracy. In order to confirm semantic accuracy, we also carried out a manual examination in which we evaluated if the relationships and elements were appropriately represented by comparing the created diagrams with the original criteria.

After testing, we ranked the models according to how well they performed across all measures. The inputs, actual outputs, expected outputs, correctness scores and correctness reasons for each LLM can be viewed here - <https://github.com/Mandalorian-way/Comparative-Analysis-of-Large-Language-Models-Result>. As seen in table 2, models with frequent errors or ambiguous outputs were scored worse, whereas those with higher syntax and semantic accuracy and lower error rates were ranked favorably. We were able to determine each model's advantages and disadvantages as well as which LLMs are best at producing precise UML diagrams straight from requirements thanks to this comparative analysis.

5 EVALUATION

The G-Eval correctness score is calculated by using a scoring function that combines the prompt, a chain-of-thought and the input context, along with the generated plantUML code. The evaluation process involves generating a set of scores based on the evaluation criteria we provide. The LLM evaluates the results and produces a probability for each score, and the final score is calculated as a weighted sum of

the probabilities and their corresponding score values. This approach normalizes the scores and provides a more continuous, fine-grained score.

The G-Eval correctness score is calculated as:

$$\text{score} = \sum_{i=1}^n p(s_i) \times s_i$$

Where:

- $p(s_i)$ is the probability of the score s_i being assigned by the LLM.
- s_i is one of the possible discrete scores in the pre-defined set of scores for evaluation criteria.

On examining the evaluation results, it is observed that Gemma2:9b-instruct-q6_K performed the best with Qwen2.5-coder:7b-instruct-q6_K and Mistral-Nemo:12b-instruct-2407-q6_K right behind the model, having comparable scores.

Gemma2:9b-instruct-q6_K performed well in modeling use cases but some relationships were mapped incorrectly, missed some functional requirements, and had mismatched use case names. Aya-Expanse:8b-q6_K received its score due to the misuse of `<include>` and `<extend>` relationships, logical inconsistencies in connecting actors and use cases, and inclusion of unnecessary text and comments that compromises syntax. Llama3:8b-instruct-q6_K correctly modeled most use cases but had used relationships incorrectly, added unnecessary comments, and omitted or misrepresented requirements, with occasional hallucinated content. Mistral-Nemo:12b-instruct-2407-q6_K followed basic syntax well, but had issues with `<include>` and `<extend>` relationships, missing or misrepresented requirements, and logical errors in mapping relationships. Qwen2.5-coder:7b-instruct-q6_K did well in modeling use cases but faced similar issues with incorrect relationship usage, missing connections, and redundant labels in relationships. Codellama:7b-instruct-q6_K struggled with similar issues, including incorrect use of relationships, logical errors in actor-to-use case connections, and extra text before or after the PlantUML code.

6 CONCLUSION AND FUTURE WORK

Automating the generation of UML diagrams holds a significant importance in reducing manual effort and improving the accuracy of the diagrams. Using Large

Table 2: Answer correctness score.

Functional Requirements	Aya expanse :8b	codellama:7b	gemma2:9b	llama3:8b	mistral-nemo:12b	qwen2.5-coder:7b
Application 1	0.524	0.314	0.864	0.003	0.577	0.722
Application 2	0.379	0.77	0.76	0.617	0.74	0.747
Application 3	0.656	0.647	0.448	0.655	0.732	0.624
Application 4	0.644	0.786	0.627	0.78	0.665	0.699
Application 5	0.387	0.308	0.47	0.517	0.469	0.646
Application 6	0.461	0.136	0.447	0.54	0.665	0.575
Application 7	0.465	0.271	0.406	0.397	0.366	0.625
Application 8	0.436	0.36	0.653	0.373	0.763	0.405
Application 9	0.479	0.382	0.865	0.484	0.532	0.665
Application 10	0.495	0.351	0.889	0.602	0.455	0.475

Language Models for code generation is a growing area of research in recent times. In this research, we have explored the capability of LLMs to create these UML diagrams. The objective of this research was to compare a few base LLMs to find out which one of them performed best for generating PlantUML code for use case diagrams.

From this research, we have identified that on average, gemma2 outperformed the other models in generating PlantUML code with least errors. However, from our observations, we see that even though gemma2 has the best average score, it did not perform the best for every input. The results produced still has a few syntactic and semantic inaccuracies. This is because base LLMs do not have any understanding of the domain and will generate inconsistent results.

In order to improve the performance of these models in this domain, extensive fine-tuning is required. A set of diverse functional requirements can be used for fine-tuning to create a robust model capable of handling complex functional requirements. Retrieval Augmented Generation can also be used to provide context on UML syntax to the LLM. Future works can use human feedback loops to iteratively refine the results and improve the semantic quality of the generated results. Using complex inputs for testing, including ambiguous and incomplete inputs, can help assess the model's adaptability to different patterns of inputs. This refined model can reduce the human effort required in the generation of UML diagrams. The final version of this improved model can be integrated into software development pipelines to save the time and effort of software analysts and architects.

REFERENCES

- Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A. A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., et al. (2024). Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.
- Ahmed, S., Ahmed, A., and Eisty, N. U. (2022). Automatic transformation of natural to unified modeling language: A systematic review. In *2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 112–119. IEEE.
- Alessio, F., Sallam, A., and Chetan, A. (2024). Model generation from requirements with llms: an exploratory study-replication package.
- Anjali, S., Meera, N. M., and Thushara, M. (2019). A graph based approach for keyword extraction from documents. In *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, pages 1–4. IEEE.
- Ardimento, P., Bernardi, M. L., and Cimitile, M. (2024). Teaching uml using a rag-based llm. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.
- De Bari, D. (2024). *Evaluating large language models in software design: A comparative analysis of uml class diagram generation*. PhD thesis, Politecnico di Torino.
- Devi Sree, R. and Swaminathan, J. (2018). Construction of activity diagrams from java execution traces. In *Ambient Communications and Computer Systems: RACCCS 2017*, pages 641–655. Springer.
- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. (2024). The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Elallaoui, M., Nafil, K., and Touahni, R. (2018). Automatic transformation of user stories into uml use case diagrams using nlp techniques. *Procedia computer science*, 130:42–49.
- Hachichi, H. (2022). A graph transformation approach for modeling uml diagrams. *International Journal of*

- Systems and Service-Oriented Engineering (IJSSOE)*, 12(1):1–17.
- Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. (2024). Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Jeong, C. (2024). Fine-tuning and utilization methods of domain-specific llms. *arXiv preprint arXiv:2401.02981*.
- Kulkarni, D. R. and Srinivasa, C. (2021). Novel approach to transform uml sequence diagram to activity diagram. *Journal of University of Shanghai for Science and Technology*, 23(07):1247–1255.
- Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., and Zhu, C. (2023). G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*.
- Nair, R. P. and Thushara, M. (2024a). Hybrid actor-action relation extraction: A machine learning approach. *Procedia Computer Science*, 233:401–410.
- Nair, R. P. and Thushara, M. (2024b). Investigating natural language techniques for accurate noun and verb extraction. *Procedia Computer Science*, 235:2876–2885.
- Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. (2024). Code llama: Open foundation models for code.
- Soudani, H., Kanoulas, E., and Hasibi, F. (2024). Fine tuning vs. retrieval augmented generation for less popular knowledge. In *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*, pages 12–22.
- Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., Mesnard, T., Shahriari, B., Ramé, A., et al. (2024). Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.
- Veena, G., Gupta, D., Lakshmi, S., and Jacob, J. T. (2018). Named entity recognition in text documents using a modified conditional random field. In *Recent Findings in Intelligent Computing Techniques: Proceedings of the 5th ICACNI 2017, Volume 3*, pages 31–41. Springer.
- Veena, G., Hemanth, R., and Hareesh, J. (2019). Relation extraction in clinical text using nlp based regular expressions. In *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, volume 1, pages 1278–1282. IEEE.
- Vemuri, S., Chala, S., and Fathi, M. (2017). Automated use case diagram generation from textual user requirement documents. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4. IEEE.
- Xie, D., Yoo, B., Jiang, N., Kim, M., Tan, L., Zhang, X., and Lee, J. S. (2023). Impact of large language models on generating software specifications. *arXiv preprint arXiv:2306.03324*.