Automated Generation of Activity Diagram using Prompt-Based Learning Technique

Vrajkumar Patel
Dhirubhai Ambani University
Gandhinagar, India
202411049@daiict.ac.in

Saurabh Tiwari
Dhirubhai Ambani University
Gandhinagar, India
saurabh_t@daiict.ac.in

***Abstract*—This project presents an automated pipeline for generating and refining UML Activity Diagrams (ADs) from natural language (NL) problem statements using large language models (LLMs). Inspired by the Chain-of-Thought prompting technique for class diagram derivation, [2] the pipeline decomposes the generation task into a sequence of reasoning steps—identifying actions, decision points, control flows, and syntactic constructs—before generating the final PlantUML representation of the diagram. To ensure semantic and syntactic accuracy, the system employs a second LLM as an evaluator, which analyzes the generated AD for completeness and correctness. Drawing on the ClarifyGPT framework, [3] the evaluator LLM identifies ambiguities, inconsistencies, or missing elements in the AD and produces targeted clarifying questions. Based on the answers—either from users or simulated responses—the first LLM regenerates a refined diagram. This iterative feedback loop ensures improved alignment with the original NL requirements. The pipeline adopts evaluation metrics and issue categorization strategies from the REW 2024 study [1]on LLM-generated UML Sequence Diagrams, focusing on adherence to UML syntax, behavioral consistency, and requirement coverage. The proposed methodology reduces manual modeling effort while promoting more accurate and standards-compliant Activity Diagram generation. The final output supports stakeholder communication and model-driven development in requirements engineering processes, with minimal human intervention.**

***Index Terms*—UML Models, Activity Diagram, Requirements engineering, Chain Of Thought (COT), Natural Language Processing (NLP), Large Language Models (LLMs), Prompt Engineering.**

I. Introduction

In software engineering, graphical models such as Unified Modeling Language (UML) diagrams serve as essential tools for visualizing system behavior, validating requirements, and improving communication among stakeholders. Among these, UML Activity Diagrams (ADs) are particularly useful for modeling the dynamic aspects of a system, capturing workflows, parallel actions, and control flows in a clear, formalized manner. However, the traditional process of constructing activity diagrams from natural language (NL) requirements is labor-intensive, error-prone, and requires substantial domain knowledge. This bottleneck poses challenges in agile and iterative development environments where rapid prototyping and requirements changes are common.

The emergence of Large Language Models (LLMs) like ChatGPT and LLaMA has revolutionized natural language processing by enabling sophisticated reasoning and generation capabilities from unstructured text. Recent research has shown the potential of LLMs to generate various UML models—including class diagrams and sequence diagrams—directly from natural language descriptions. Yet, generating UML Activity Diagrams, which require a nuanced understanding of control structures and parallel behaviors, remains an underexplored domain.

To address this gap, this project introduces an automated pipeline for generating and refining UML Activity Diagrams from natural language problem statements using LLMs. The pipeline follows a step-by-step Chain-of-Thought (CoT) [2] inspired methodology for activity diagram generation, decomposing the process into stages: identifying actions, control nodes (initial, final, decision, merge, fork, join), and establishing valid control flows. This structured reasoning approach is adapted from techniques used in class diagram generation to enhance clarity, transparency, and traceability in model creation.

Furthermore, recognizing that NL requirements often contain ambiguities, inconsistencies, or incompleteness, this project adopts a refinement loop inspired by the ClarifyGPT framework. [3] A secondary LLM evaluates the initially generated activity diagram for syntactic correctness and semantic completeness, identifying potential flaws and generating a targeted questionnaire to seek clarification. Based on the responses—either provided by the user or simulated—the primary LLM regenerates an improved diagram. This iterative, clarification-based loop ensures higher fidelity to the original requirements and improves the quality of the final model.

To assess the effectiveness of this pipeline, we employ evaluation metrics derived from the REW 2024 paper "Model Generation with LLMs: From Requirements to UML Sequence Diagrams". [1] These include syntactic conformance, semantic alignment, behavioral completeness, and coverage of requirements. Through this rigorous evaluation and iterative refinement mechanism, the generated activity diagrams demonstrate improved structural integrity, stakeholder understandability, and potential for integration into model-driven development workflows.

This work contributes to the growing body of research at the intersection of requirements engineering, LLMs, and model generation, providing a practical and scalable approach for

semi-automated UML Activity Diagram creation with minimal manual intervention.

The rest of the paper is organized as follows. Section II discusses the background and related work. Section III introduces the proposed CoT framework, and Section IV demonstrates the experiment design settings. Section V showcases the preliminary quantitative results. Furthermore, we provide a comprehensive discussion in Section VI. Finally, Section VII concludes our study.

II. BACKGROUND AND RELATED WORK

A. Uml Models

The Unified Modeling Language (UML) is a standardized visual language for modeling and documenting software systems (OMG, 2003). Like blueprints for buildings, UML models are essential for communicating system structure and behavior among teams and stakeholders.

UML combines Object Modeling Technology (OMT) and Object-Oriented Analysis and Design (OOAD) to support both structural and behavioral modeling. It includes:

- Static Diagrams (e.g., class, object, package) for system structure
- Behavioral Diagrams (e.g., use-case, activity, sequence) for dynamic behavior
- Implementation Diagrams (e.g., component, deployment) for physical architecture

Among these, activity diagrams excel at visualizing workflows and decisions, while sequence diagrams capture object interactions over time.

B. Activity Diagrams

UML Activity Diagrams are used to model the procedural flow of control within a system or process. They are particularly effective for capturing both high-level business workflows and low-level process logic, depending on the abstraction level required. At the business unit level, they can represent overarching business operations, whereas at the system level, they can describe detailed step-by-step operations involved in executing a use case.

Activity diagrams are especially beneficial in scenarios where use-case descriptions become large or complex, as they help decompose and visualize specific parts of those scenarios. By representing workflows visually, activity diagrams enhance the understandability and traceability of system behavior, especially in the early phases of software design. They are frequently employed to complement use-case specifications, offering a more detailed, structured, and actionable view of the user-system interactions.

The key elements of an activity diagram include: Initial Node: Represented as a filled circle, this denotes the starting point of the activity flow.

1) Activity (or Action) Node: Represented as a rounded rectangle, it denotes a task or operation in the workflow.
2) Control Flow: Directed arrows show the flow of control from one activity to the next.
3) Decision Node: Depicted as a diamond, it models conditional branches where different flows may be taken based on specified criteria.
4) Final Node: Represented as a bullseye symbol (a filled circle within a larger circle), it marks the termination of the process.

In complex workflows, fork and join nodes are used to model parallelism, allowing activities to be executed concurrently or synchronized appropriately. The combination of these elements enables activity diagrams to represent sequential, conditional, and parallel control flows clearly and intuitively.

C. Chain Of Thought Prompting

The emergence of Large Language Models (LLMs) has enabled a shift from manual modeling to automated workflows through techniques like Chain-of-Thought (CoT) prompting. CoT guides LLMs to reason step by step, improving output accuracy and traceability.

In this project, CoT prompting is used to systematically generate UML Activity Diagrams by first identifying actions, then control nodes (e.g., initial, decision, fork/join), and finally establishing control flows. This mimics human reasoning and improves the reliability of generated diagrams.

While CoT has shown success in other tasks, its use in behavioral model generation—particularly activity diagrams—is still underexplored. This work adapts CoT to fill that gap, enabling the generation of syntactically valid and semantically meaningful PlantUML scripts from natural language descriptions.

By structuring the generation process, CoT improves clarity, accuracy, and the practical usability of LLMs in model-driven development.

D. Related Work

Large Language Models in Requirements Engineering. Large Language Models (LLMs) are deep neural networks built on transformer architectures and trained on vast textual corpora using self-supervised learning. This enables them to capture the statistical patterns and semantic structures of natural language [4]. Modern LLMs are generative by design, capable of producing human-like responses to natural language inputs, commonly referred to as prompts. Despite their success in areas like software testing, code generation, and program repair, their application in Requirements Engineering (RE) remains relatively limited [5], [6]. Within RE, LLMs have been explored in various domains, such as legal text summarization [7] and requirements traceability [8]. Of particular relevance to this study, Chen et al. [9] assessed GPT-4's ability to generate goal models from natural language using the Goal-oriented Requirement Language (GRL). Additional work by Chen et al. [11] evaluates GPT-3.5 and GPT-4 for generating class diagrams from textual descriptions. Similarly, Camára et al. [10] conducted an exploratory study on using ChatGPT for class diagram generation, finding that iterative refinement

was necessary to achieve acceptable model quality. Other RE-focused research has proposed prompt pattern catalogs tailored to tasks like classification and traceability [12], [13].

Research Gap Most existing research on automated model generation from natural language requirements has primarily relied on rule-based approaches, which involve manually crafting complex heuristic rules to map linguistic constructs to modeling elements. While effective to some extent, such approaches are often difficult to maintain, brittle across domains, and not easily adaptable to different modeling tasks or requirement styles.

A few studies, such as Saini et al.'s work on class diagram generation, attempt to bridge this gap by combining rule-based systems with machine learning techniques. However, these hybrid approaches still fall short of leveraging the full potential of Large Language Models (LLMs) for end-to-end model generation.

Recent efforts have explored the use of LLMs for generating class diagrams and, to a lesser extent, sequence diagrams from user stories or structured use cases. However, these studies often focus on simplified or synthetic requirements and lack robustness when applied to real-world, domain-specific specifications. Moreover, very few works, if any, have addressed the challenge of generating UML Activity Diagrams—which require detailed modeling of control flows, decisions, concurrency, and synchronization—directly from natural language using LLMs.

To the best of our knowledge, this project is the first to propose a fully automated, LLM-based pipeline for generating UML Activity Diagrams from unstructured natural language problem statements. It not only decomposes the diagram generation task using Chain-of-Thought prompting, but also incorporates a ClarifyGPT-inspired feedback loop to iteratively refine the diagrams based on clarification questions generated by a secondary LLM. Furthermore, the project evaluates the generated models against real-world-inspired evaluation metrics to ensure practical applicability and alignment with requirements.

III. Methodology

This section presents the methodology adopted in our project for the automated generation and refinement of UML Activity Diagrams (ADs) from natural language (NL) problem statements using Large Language Models (LLMs). The approach is built upon a Chain-of-Thought (CoT) prompting framework, where the model is guided to perform structured reasoning in multiple steps. Each stage of the pipeline mimics how a human analyst would incrementally analyze, extract, and construct modeling elements from textual requirements.

A. Steps of Activity Diagram Derivation

Inspired by the structured reasoning used in class diagram derivation [2], the process of activity diagram generation is broken down into the following key stages:

1) **Action Identification:** The pipeline begins by extracting atomic system behaviors or operations explicitly or implicitly described in the problem statement. These actions form the core activity nodes of the diagram.
2) **Define Activity Nodes:** Next, the model identifies specialized control elements that define the structure and flow of the activity:
   - Initial Node (start of the flow)
   - Final Node (termination of the flow)
   - Decision and Merge Nodes (for branching and converging conditions)
   - Fork and Join Nodes (to express parallelism)
3) **Establish Control Flow:** After nodes are identified, the LLM connects these nodes via directed transitions that represent the logical and procedural flow between actions and control elements. This step ensures adherence to UML activity diagram syntax and logic.
4) **PlantUML Script Generation**: The final step in this phase translates the identified nodes and flows into a PlantUML script, a textual representation that can be rendered into a visual UML diagram. The script ensures that the diagram is syntactically valid and easily viewable using standard UML tools.
5) **Generating Clarifying Questions:** Following the initial diagram generation, the quality of the output is evaluated through a second LLM that acts as an automated evaluator. Inspired by the ClarifyGPT framework, this LLM examines the completeness, correctness, and semantic alignment of the generated diagram with respect to the original NL requirements.
   If gaps, ambiguities, or inconsistencies are detected—such as missing parallel branches, incorrect transitions, or unmodeled scenarios—the evaluator generates a targeted set of clarifying questions. These questions are designed to elicit missing information and clarify vague or under-specified parts of the input requirements. The questions may be answered by a user or simulated programmatically.
6) **Regenerating the Activity Diagram:** Based on the answers to the clarifying questions, the first LLM regenerates an improved version of the activity diagram. This regeneration phase repeats the step-by-step CoT process, now informed by the clarified requirements. As

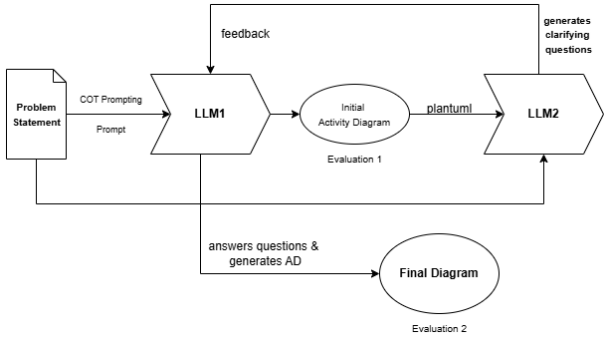

Fig. 1. Methodology

a result, the final diagram is more semantically faithful to the original intent, more structurally complete, and more compliant with UML conventions.

This iterative feedback loop—from generation to clarification to regeneration—enhances the quality, accuracy, and reliability of the generated activity diagrams, making the system more robust for real-world applications in requirements engineering and model-driven development.

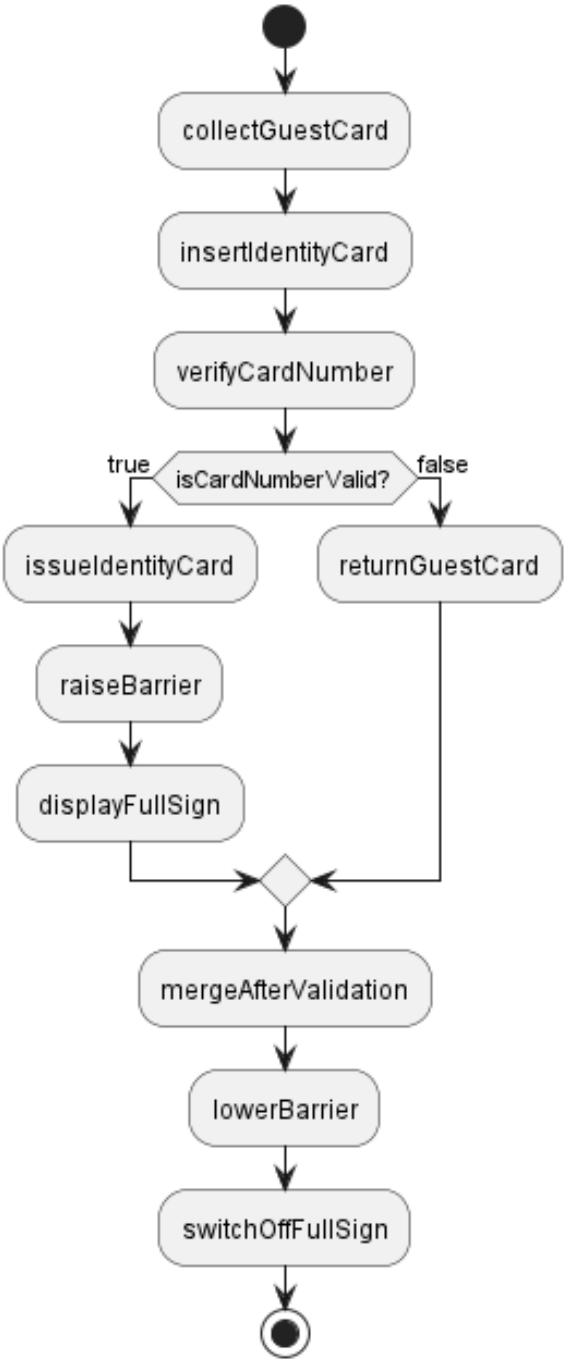

Fig. 2. Car Parking Scenario Before applying ClarifyGPT

B. Implementation

This project is built on a Chain-of-Thought (CoT) prompting framework tailored for generating and refining UML Activity Diagrams from natural language inputs. Unlike traditional rule-based methods, our approach leverages the reasoning capabilities of Large Language Models (LLMs) to sequentially identify actions, control nodes, and control flows.

The CoT framework is LLM-agnostic, allowing flexibility across models. For implementation, we primarily use gemini-1.5-flash via the GEMINI API, selected for its strong balance between performance and cost-effectiveness. Additionally, we employ local models—LLaMA 3.2 and DeepSeek-R1:1.5B—specifically for generating clarification questions in the iterative refinement loop.

IV. Experiment

The overarching goal of this study is to investigate the capability of Large Language Models (LLMs) to automatically generate UML Activity Diagrams from natural language (NL) problem statements. While activity diagrams can serve various roles—such as facilitating system design, process analysis, and workflow visualization—our study focuses on their role in complementing textual requirements to improve communication between stakeholders and support early-stage modeling in the requirements engineering (RE) process.

A. Research Questions

To guide our investigation, we formulate the following research questions:

RQ1: What is the quality of the activity diagrams generated from NL requirements by LLMs? RQ1 aims to provide a quantitative evaluation of the generated activity diagrams in terms of completeness, correctness, and adherence to UML standards. The goal is to assess the practical applicability of using LLMs (via CoT prompting) in real-world modeling scenarios.

RQ2: What issues emerge when using LLMs to generate activity diagrams from NL requirements? RQ2 explores the qualitative aspects of the generation process. We aim to identify a catalog of typical problems—such as missing nodes (e.g., fork/join), incorrect transitions, structural incoherence, or low understandability—that arise from inherent limitations in the LLM or ambiguities in the input requirements.

B. Experiment Design

This section outlines the experimental setup devised to evaluate the performance of our LLM-based pipeline to generate UML activity diagrams from statements from natural language problems. The evaluation framework is aligned with our research questions and is structured to assess both the quality of generated diagrams and the issues that arise during the generation process.

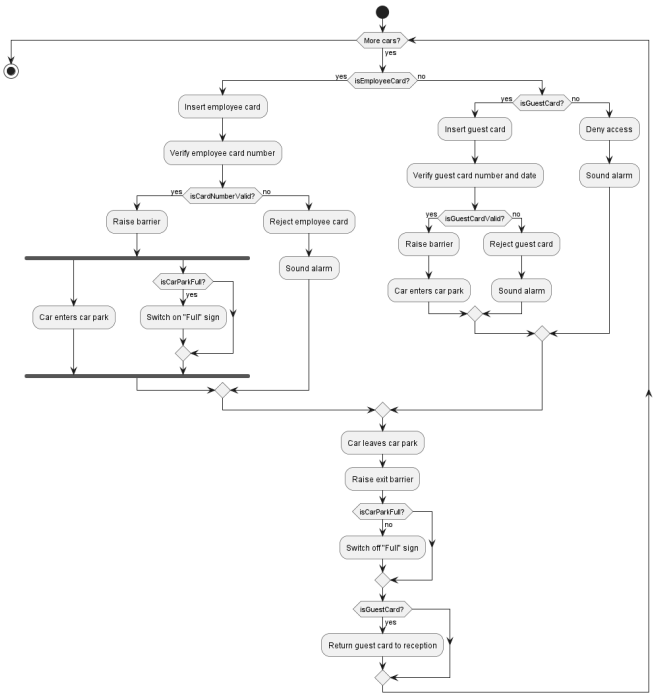

Fig. 3. Car Parking Scenario After applying ClarifyGPT

*1) **Objective**:* The experiment is designed to:

- Quantitatively measure the quality of the generated activity diagrams in terms of correctness, completeness, and syntactic validity.
- Qualitatively analyze typical problems encountered during diagram generation and identify areas where clarification-based refinement improves results.

*2) **Dataset**:* We selected a set of 3 Problem statements written in Natural Language from diverse application domains, covering a range of scenarios (e.g., e-commerce, healthcare and banking). These statements vary in complexity and style to simulate real-world variability in requirements. The dataset is curated to include both structured and unstructured textual inputs, mimicking the nature of requirements encountered in practical settings.

*3) **LLM Configuration**:* The experiments use gemini-1.5-flash as the default LLM for both initial diagram generation and clarification-based regeneration. Two more local LLMs are also used for generating clarification questions (e.g., Llama 3.2 and Deepseek-r1:1.5b). The LLM is accessed through the Gemini API. All prompts are constructed using the Chain-of-Thought (CoT) strategy and provide few-shot examples to guide each step in the generation process.

- Initial LLM (Generator): Extracts activity nodes, control nodes, and control flows, and generates the PlantUML script.
- Secondary LLM (evaluator): Reviews the generated diagram and produces clarification questions based on missing or ambiguous elements.

*4) **Evaluation Criteria**:* To assess the quality of the generated UML Activity Diagrams, we adopted a manual evaluation process conducted independently by me. Each generated diagram—both before and after the clarification-based refinement—was critically analyzed and assessed based on a set of well-defined quality criteria. The evaluation process was documented through detailed textual log files for traceability and qualitative insights.

The following five quality dimensions were used to assess each diagram:

- Completeness: The extent to which the diagram captures all relevant behavioral information from the natural language requirements. This includes coverage of actions, decision points, and control flow paths (external completeness [14]). A complete diagram should support effective communication with stakeholders.
- Correctness: The degree to which the behavior represented in the diagram is coherent, logical, and aligned with the functional intent of the original requirements. This includes validation of action sequencing, control logic, and node usage.
- Adherence to the Standard: The syntactic and semantic validity of the diagram, particularly whether it complies with UML syntax and whether the generated PlantUML script can be parsed without error. Semantic adherence refers to correct usage of constructs such as decision nodes, forks, and joins.
- Understandability: The clarity of the diagram in terms of layout, flow, and structural organization. A highly understandable diagram avoids unnecessary complexity and redundancies while remaining faithful to the requirements.
- Terminological Alignment: The consistency of terminology used in the diagram compared to that in the original problem statement. This ensures that stakeholders can map elements in the model directly to phrases or terms from the source requirements.

Each criterion was evaluated using a five-point ordinal scale, where each integer represents the extent to which the criterion was fulfilled: 1 = “Not fulfilled at all”; 2 = “Fulfilled to a minimal extent”; 3 = “Partially fulfilled”; 4 = “Mainly fulfilled”; 5 = “Completely fulfilled”. The resulting scores were analyzed to address RQ1.

In addition to the numerical ratings, a textual justification was provided for each score, reflecting on the specific issues observed in the generated diagrams. These qualitative insights were subsequently analyzed to answer RQ2.

These criteria were established through consensus among the project team, drawing from preliminary experiments and existing standards—particularly ISO/IEC/IEEE 29148:2018(E) [**?**], which outlines quality attributes for software requirements and their representations.

V. Execution and Results

A. RQ1: Quality of the Diagrams

Figures 4, 5, and 6 present bar charts comparing the evaluation results before and after applying improvements across different criteria for each problem statement, assuming well-formed requirements as input. Tables I–III report the statistical summaries and test results, while Tables IV and V display the clarification questions generated for Problem 3 using the ClarifyGPT approach.

The analysis reveals that completeness, understandability, adherence to standards, and terminological alignment show statistically significant improvements, with values exceeding the mean (p-value ≤ 0.05). This indicates a satisfactory level of model quality for these criteria.

However, correctness scores do not show a significant difference from the mean (p-value > 0.05), suggesting that this criterion is, on average, not adequately addressed. Furthermore, the violin plot for completeness, while skewed toward higher values, still reflects some suboptimal scores (e.g., several 3s and 4s).

These findings suggest that a requirements analyst aiming to generate diagrams using LLMs must carefully verify both the accuracy and completeness of the output. Special attention should be given to the quality issues discussed in the following section.

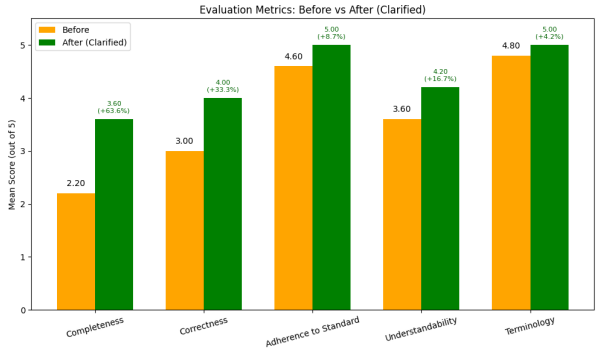

Fig. 4. Bar chart Before vs After

B. RQ2: Emerging Issues

In this section, we present the High-Level (HL) and Mid-Level (ML) codes that emerged from the thematic analysis. These codes are illustrated through representative excerpts from the requirements and corresponding sections of the generated diagrams. Each example highlights a specific portion of a requirements set along with a focused view ("zoom") of the associated model, helping to demonstrate the identified issues in context.

1) ***Completeness.***
Over-Summarization: Generated content often tends to be overly condensed, resulting in the omission of critical elements such as full requirements or their parts, system components, function calls, logical conditions, and message exchanges.
Effect of Poor Requirements Quality: Low-quality or ambiguous requirements often contribute to missing information in the generated models. When the input is unclear or difficult to interpret, important details may be unintentionally abstracted away, leading to incomplete or vague diagram representations.
Impact of Inconsistencies: Inconsistencies among requirements may also result in omissions. Rather than exposing conflicts, the LLM may smooth over them, which further contributes to missing or incorrect content in the output models.
Imprecision in Timing and Numerical Information: LLMs, with their emphasis on natural language, often struggle with reasoning about timing, numerical constraints, and mathematical logic. This limitation becomes particularly apparent when processing requirements that involve precise quantities or temporal conditions.

TABLE I
Statistical Comparison of Metrics Before and After Clarification

Metric	Before Mean	After Mean	% Improvement
Completeness	2.2	3.6	63.34
Correctness	3.0	4.0	33.33
Adherence to Standard	4.6	5.0	8.70
Understandability	3.6	4.2	16.67
Terminology	4.8	5.0	4.17

TABLE II
Statistical Comparison of Metrics Before and After Clarification

Metric	T-Statistic	P-Value	Significance (p-value ≤ 0.05)
Completeness	5.7155	0.0046	Significant
Correctness	∞	0.0000	Significant
Adherence to Standard	1.6330	0.1778	Not Significant
Understandability	2.4495	0.0705	Not Significant
Terminology	1.0000	0.3739	Not Significant

2) ***Correctness.***
Incorrect Interactions: The models may depict function calls or message exchanges in an incorrect sequence, or present an action flow that does not align with the intended logic of the requirements.
Misidentified Components or Actors: Components or actors are sometimes mistakenly used in place of functions or messages, reflecting a misunderstanding of their roles.
Structural Errors: LLMs occasionally fail to capture abstract conceptual structures such as states or control loops that are essential to a well-formed model. Additionally, related function calls may appear dispersed across the model rather than being logically grouped.

3) ***Adherence to Standard.***
Syntactic Errors: In some cases, the generated output contains syntax issues that prevent tools like PlantText from correctly interpreting the model. Even when the syntax error is identified, the LLM may not be able to resolve it, requiring manual intervention by the user.

Semantic Misuse of Constructs: Certain modeling elements are used incorrectly in a semantic sense, which can lead to misinterpretation of the model's intent or function.

4) ***Terminology.***
Introduction of Unfamiliar Terms: LLMs may introduce new terms that are not present in the original requirements, which can break consistency and reduce traceability between the model and the source material.
Omission of Key Terms: Terms identified as important during requirements analysis are sometimes omitted from the resulting model, reducing fidelity.
Terminological Inconsistency: The generated models may include terms that differ from those in the requirements while conveying a similar meaning, resulting in inconsistencies that can affect clarity and understanding.
5) ***Understandability.***
Lack of Traceability: Evaluating completeness and correctness becomes challenging when there is no clear mapping between elements in the requirements and those in the model. Although trace notes are sometimes included, they can be partial or inaccurate.
Manifestations of Incoherence: When faced with ambiguous, inconsistent, or incomplete requirements, LLMs still attempt to generate models—often resulting in diagrams that are difficult to interpret or verify.
Redundancy: Superfluous or repetitive elements may be introduced, which, while not incorrect, can clutter the model and diminish its readability.
Excessive Detail: At times, the diagrams are overly detailed—featuring deeply nested conditions or loops—making them harder to navigate. A more abstract representation would be preferable in such cases.

VI. Accuracy Calculation

$$\text{Percentage Improvement} = \left(\frac{\text{After} - \text{Before}}{\text{Before}}\right) \times 100$$

$$\text{Percentage Improvement}_{\text{Gemini}} = \left(\frac{23 - 20}{20}\right) \times 100 = 15\%$$

Similarly, We got the Percentage improvement for other models too for each problem statement.

VII. Conclusion

The experimental results demonstrate a significant improvement in the quality of generated Activity Diagrams after applying the proposed ClarifyGPT-based refinement approach using LLMs. Across three different problem statements, all evaluated models—Gemini, LLaMA-3, and Deepseek—showed consistent performance gains in terms of completeness, correctness, and overall alignment with UML standards.

For **Problem 1**, the percentage improvement ranged from 10% to 15%, indicating a modest but consistent enhancement over the baseline.

In **Problem 2**, all models achieved a 23.53% improvement, reflecting better clarification and integration of missing behavioral aspects into the diagrams.

Problem 3 demonstrated the most substantial gains, with all models showing a 37.5% improvement, confirming the effectiveness of iterative clarification in handling incomplete and semantically weak initial diagrams.

These results confirm that incorporating clarification questions, step-by-step reasoning (CoT), and iterative regeneration leads to higher-quality and semantically richer Activity Diagrams, irrespective of the LLM used. Gemini showed a slightly higher edge in Problem 1, while LLaMA-3 and Deepseek performed comparably well across all three tasks.

The ClarifyGPT-inspired pipeline not only improves diagram quality but also introduces a scalable and model-agnostic method to reduce ambiguity and ensure diagram correctness and completeness in automated model generation workflows.

References

[1] A. Ferrari, S. Abualhaija, and C. Arora, "Model generation with LLMs: From requirements to UML sequence diagrams," in *Proc. of the 32nd IEEE Int. Requirements Engineering Conference (RE)*, 2024.

[2] Y. Li, J. Keung, X. Ma, C. Y. Chong, and J. Zhang, "LLM-based class diagram derivation from user stories with chain-of-thought promptings," in *Proc. of the 48th IEEE Annual Computers, Software and Applications Conference (COMPSAC)*, 2024.

[3] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, and C. X. Wang, "ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification," *Proc. ACM Softw. Eng.*, 2024.

[4] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Comput. Surv.*, vol. 56, no. 2, pp. 1–40, 2023.

[5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," arXiv preprint arXiv:2310.03533, 2023.

[6] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," arXiv preprint arXiv:2308.10620, 2023.

[7] C. Jain, P. R. Anish, A. Singh, and S. Ghaisas, "A transformer-based approach for abstractive summarization of requirements from obligations in software engineering contracts," in *Proc. IEEE Int. Requirements Engineering Conference (RE)*, 2023, pp. 169–179.

[8] A. D. Rodriguez, K. R. Dearstyne, and J. Cleland-Huang, "Prompts matter: Insights and strategies for prompt engineering in automated software traceability," in *Proc. IEEE Int. Requirements Engineering Workshop (REW)*, 2023, pp. 455–464.

[9] B. Chen, K. Chen, S. Hassani, Y. Yang, D. Amyot, L. Lessard, G. Mussbacher, M. Sabetzadeh, and D. Varró, "On the use of GPT-4 for creating goal models: An exploratory study," in *Proc. IEEE Int. Requirements Engineering Workshop (REW)*, 2023, pp. 262–271.

[10] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, "On the assessment of generative AI in modeling tasks: An experience report with ChatGPT and UML," *Software and Systems Modeling (SoSyM)*, pp. 1–13, 2023.

[11] K. Chen, Y. Yang, B. Chen, J. A. H. López, G. Mussbacher, and D. Varró, "Automated domain modeling with large language models: A comparative study," in *Proc. ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, 2023, pp. 162–172.

[12] K. Ronanki, B. Cabrero-Daniel, J. Horkoff, and C. Berger, "Requirements engineering using generative AI: Prompts and prompting patterns," arXiv preprint arXiv:2311.03832, 2023.

[13] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," arXiv preprint arXiv:2303.07839, 2023.

[14] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," IST, vol. 45, no. 14, pp. 993–1009, 2003.