

Towards Reliable AI-Assisted Behavioral Modeling: Evaluating Prompting, Retrieval, and Repair Strategies for UML State Transition Diagram Generation

Mussammat Maimuna Faria ✉ 

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Mashiat Amin Farin ✉ 

Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

Yasin Sazid ✉ 

Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

Ahmedul Kabir ✉ 

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

Background. Generating behavioral models like state transition diagrams from natural language requirements presents challenges in requirements analysis and design. Traditional NLP and ML methods struggle to maintain the correct execution flow and structural consistency. Recent advances in LLMs offer new opportunities, though their effectiveness in generating behavioral models, particularly concerning prompting, retrieval, and repair strategies, remains largely unexamined.

Aims. This paper evaluates how LLM-based generation strategies (zero-shot, one-shot, few-shot, RAG) affect state transition diagram quality (correctness, completeness, understandability, terminological alignment) and whether iterative repair improves validity.

Method. We built a dataset of 80 requirement–diagram pairs from software engineering textbooks. We generated PlantUML code for each requirement using four prompting strategies across four open-source LLMs. To isolate the effect of different knowledge sources, we conducted a RAG analysis study comparing four retrieval corpus configurations (full, examples-only, rules-only, theory-only). We also assessed an iterative repair workflow using automated evaluation. We evaluated syntactic validity via PlantUML validation and structural validity via rule-based analysis, supplementing with human evaluation of quality dimensions. We produced 1,169 PlantUML outputs, of which 120 validated outputs were manually evaluated.

Results. Few-shot prompting improved syntactic validity from 28.7% to 93.5% on average over zero-shot. Iterative repair significantly improved structural validity, especially for DeepSeek (from 22.2% to 88.9%), with most repairs succeeding within two iterations. Human evaluation showed understandability was rated higher than correctness, indicating a quality-perception gap.

Conclusions. We recommend few-shot prompting with iterative repair for AI-assisted behavioral modeling. Our findings reveal critical trade-offs: retrieval improves syntax but risks hallucination, model architecture matters more than strategy, and automated validation cannot substitute for human correctness assessment. This is the first empirical study systematically evaluating prompting, retrieval, and repair in LLM-based UML state transition diagram generation from natural language.

2012 ACM Subject Classification Software and its engineering → Software design engineering

Keywords and phrases State Transition Diagrams, Behavioral Software Modeling

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.19

Category Technical Track Paper

Supplementary Material

Software: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow [13]
archived at `swb:1:dir:eb2ad1f0b722352d509a2d8d84762ee769daaa89`



© Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 19; pp. 19:1–19:21



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding *Mussammat Maimuna Faria*: Supported by a fellowship from the ICT Division, Government of Bangladesh (Memo No. 56.00.0000.000.052.33.0001.25-178, dated 9 June 2026).

1 Introduction

Requirements engineering bridges stakeholder aspirations and technical software implementation by relying primarily on natural language requirements, which remain widely used due to their flexibility and accessibility for non-technical stakeholders [47]. However, natural language often introduces lexical, syntactic, and semantic ambiguities [43], necessitating rigorous requirements analysis and modeling. Graphical modeling languages like the Unified Modeling Language (UML) [38] provide structured representations of system behavior [32]. UML diagrams are categorized into structural models and behavioral models [1, 45]. Among behavioral models, state transition diagrams are crucial for modeling discrete system states and event-driven transitions. These diagrams are essential in safety-critical applications, such as healthcare, financial processing, embedded systems, and robotics [39], where state logic errors can lead to significant failures. Yet, constructing these diagrams manually is labor-intensive and error-prone, as analysts must infer states, transitions, and guards implicitly embedded in text – a task more challenging than automating structural models [10]. Previous automation attempts, using Natural Language Processing (NLP) or machine learning (ML), failed to achieve the necessary contextual reasoning for global model consistency [61].

Recent advances in large language models (LLMs) enable automation in behavioral software modeling through their ability to process long inputs and capture semantic relationships across requirements [4]. Studies show LLMs effectively support software engineering tasks like code generation and requirements analysis [31, 50, 53, 52, 15, 11, 6, 46, 57]. However, most research has focused on structural UML models, leaving the generation of behavioral models, particularly UML state transition diagrams, underexplored [26, 60].

Behavioral diagrams must preserve global execution semantics [9], including state reachability, transition consistency, branching behavior, and lifecycle correctness [25] across the entire model. Consequently, the challenge lies in not just generating valid diagrams but ensuring they are structurally consistent, behaviorally correct, and comprehensible to practitioners. In practice, generated diagrams often require verification and refinement, making the reliability of AI-assisted modeling contingent on the LLM and the surrounding workflow, which includes prompting strategies, retrieval support, automated validation, and iterative repair. Our study examines how different AI-assisted generation workflows affect the reliability of synthesizing UML state transition diagrams from natural language requirements.

In this paper, we present a dataset and an empirical framework to systematically evaluate LLM-based generation of UML state transition diagrams from natural language requirements. The dataset consists of curated requirement–diagram pairs derived from software engineering textbook sources, including structured requirements and corresponding PlantUML [40] representations. We evaluate multiple open-source LLMs (Meta-Llama-3.1-8B-Instruct [22], DeepSeek-R1-Distill-Qwen-14B [23], Mistral-7B-Instruct [30], and Qwen2.5-7B-Instruct [58]), chosen for their use in recent literature [12, 56, 62], across generation strategies, including structured prompting [55], Retrieval-Augmented Generation (RAG), and an iterative repair workflow for iterative refinement. We introduce diagram-specific metrics such as syntax validity (PlantUML validation), structural validity (rule-based analysis of the diagram detecting orphan states, unreachable states, invalid transitions, etc.), and quality evaluation through human evaluation (correctness, completeness, understandability, and terminological alignment), which enables assessment of the diagrams and workflow. This study investigates the behavior of LLMs in software modeling tasks. While LLMs can generate diagrams from requirements, the impact of different prompting and retrieval strategies on structural

correctness and reliability is unclear. Generating valid state transition diagrams requires identifying states and transitions while ensuring global properties like reachability and consistency. To investigate the effect of generation strategies on structural correctness, we first ask:

RQ1: How do LLM-based generation strategies (zero-shot, one-shot, few-shot, RAG) impact the validity (syntactic and structural) of state transition diagrams?

Recent studies suggest that RAG may improve LLM performance by incorporating relevant contextual information; however, its effectiveness in structured diagram generation remains underexplored [54, 18]. To evaluate its impact, we analyze how retrieval augmentation influences syntactic validity and structural validity across different retrieval knowledge-base configurations, including requirement-diagram examples, PlantUML syntax documentation, and theoretical resources on UML state transition diagrams:

RQ2: How do different retrieval knowledge base configurations in RAG affect the validity of state transition diagrams?

To explore how repair mechanisms can improve state transition diagrams, we ask,

RQ3: How effective are iterative repair workflows in improving the syntactic and structural validity of state transition diagrams?

Beyond structural and syntactic validity, it is important to assess whether generated diagrams accurately represent the intended system behavior and remain understandable to human practitioners. Since the practical usefulness of behavioral models ultimately depends on human interpretation and evaluation, we assess the behavioral quality of generated diagrams through human evaluation across multiple quality dimensions:

RQ4: To what extent do UML state transition diagrams generated using different LLM-based strategies satisfy quality dimensions (correctness, completeness, understandability, and terminological alignment)?

The contributions of this work are: (1) a benchmark dataset of 80 requirement–state transition diagram pairs; (2) novel automated metrics for structural validity via rule-based graph analysis; (3) a systematic comparison of prompting, retrieval, and repair strategies for enabling LLMs to function as software modeling assistants; and (4) a human evaluation framework assessing quality dimensions.

2 Background and Related Work

2.1 Rule-Based and Symbolic NLP Approaches

Early automated software architectural model derivations utilized symbolic NLP techniques like part-of-speech tagging and syntax parsing [59]. ArchE [3] was an early system using predefined architectural reasoning rules for software design. Similarly, frameworks like aToucan [59] applied handcrafted linguistic patterns to extract entities and actions from use case models, mapping them to UML elements. Jahan et al. [28] enhanced rule-based approaches with heuristic mappings for generating sequence diagrams from textual requirements. While effective for structured requirements, these approaches were difficult to generalize across domains. Rule-based systems struggled to capture implicit behavioral dependencies and global execution semantics needed for modeling behaviors in UML state transition diagrams [59, 28, 29]. Consequently, symbolic approaches remained limited to requirement extraction tasks and structural UML models.

2.2 Machine Learning for Software Architectural Model Extraction

Later work reframed software model extraction as an ML problem using classification and sequence-labeling techniques [29], which utilized lexical, syntactic, and semantic features to identify software entities and relationships from natural language requirements. However, traditional ML approaches were constrained by the limited availability of large annotated datasets in software engineering [29]. Also, classical ML models often lacked the contextual reasoning capabilities required to preserve behavioral consistency across an entire state transition diagram [29, 37]. Consequently, most prior work focused on extracting structural relationships rather than generating behaviorally consistent UML models.

2.3 LLMs in Requirements Engineering and UML Modeling

LLMs have transformed requirements engineering by enabling complex reasoning without extensive manual feature engineering [16, 51]. Studies show LLMs assist in ambiguity detection, requirement refinement, and information extraction from Software Requirements Specifications (SRS) [34]. Frameworks like REQINONE [62] and ReqBrain [24] demonstrate LLMs can convert natural language requirements into structured representations, enhancing clarity and completeness through task-specific tuning. Research also focuses on UML model generation, particularly for structural artifacts like class diagrams [16, 51, 8]. LLMs generally produce higher-quality diagrams when relationships are clearly described [28, 29, 16, 56]. Various approaches aim to enhance AI-assisted software modeling workflows. Cervantes et al. [7] integrate the Attribute-Driven Design (ADD) into LLM interactions for better alignment with expert architectural solutions. Multi-agent systems like NOMAD [19] decompose UML class diagram generation into specialized subtasks for improved modularity. Other studies propose hybrid workflows combining LLM generation with static analysis and verification tools for increased reliability [27]. Despite these advancements, limitations persist. Most studies focus on structural models, leaving behavioral model generation underexplored [16, 51, 8, 12]. Generated diagrams are often evaluated through subjective inspection or syntax correctness, with limited attention to behavioral consistency [51, 37, 27]. Many approaches still rely heavily on manual refinement and expert intervention [51, 34, 7]. Lastly, prior studies lack standardized empirical evaluation frameworks for systematically comparing prompting, retrieval, and repair workflows [29, 16, 51, 12].

2.4 Motivation

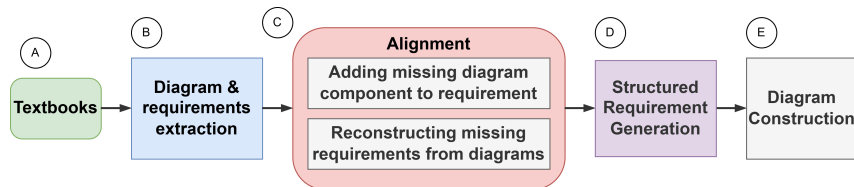
Existing research shows the potential of LLMs for requirements engineering and structural model generation [16, 51, 62, 24]. However, important gaps remain for behavioral modeling tasks. Prior work focuses on structural UML artifacts like class diagrams [16, 51, 8, 12], whereas behavioral models demand global execution semantics and behavioral consistency across the entire diagram [37, 27]. Moreover, existing approaches rely on manual refinement [51, 34, 7], offer limited automated validation [51, 49, 27], and lack standardized empirical evaluation frameworks for AI-assisted modeling workflows [29, 16, 51, 12]. Additionally, past evaluations treat diagram generation in isolation, rather than as part of a comprehensive software engineering workflow [27, 12], which limits understanding of how workflow components affect the AI-assisted behavioral modeling validity. Motivated by these limitations, this study systematically evaluates prompting strategies, retrieval augmentation, validation-aware repair workflows, and automated and quality evaluation for UML state transition diagram generation from natural language requirements.

3 Methodology

We conducted an empirical study to evaluate LLM capability in generating state transition diagrams from natural language requirements. First, we created a ground-truth dataset of requirement-diagram pairs from the literature. Then, we generated diagrams from the requirements using three strategies: prompting, retrieval, and repair. Finally, we evaluated the generated diagrams using automated and human evaluation metrics.

3.1 Dataset Preparation

To the best of our knowledge, no existing dataset with requirement-diagram pairs was available as ground truth for state transition diagram generation. Therefore, we constructed a benchmark dataset from 12 software engineering textbooks and UML instructional materials [48, 44, 41, 33, 47, 35, 38, 2, 21, 20, 17, 5]. Textbooks were chosen because they: (A) provide curated examples of system behavior reflecting standard modeling practices; (B) feature clear behavioral logic, essential for reliable ground truth; and (C) are publicly accessible and reproducible, aligning with empirical software engineering standards.



■ **Figure 1** Behavioral Modeling Dataset Construction Workflow.

For each data point, we extracted the state transition diagram in PNG format and its corresponding natural language requirements in txt format (Figs. 1-A, 1-B). However, many natural language requirements were incomplete or absent, with only diagrams provided in the books. We addressed this inconsistency by updating the requirements to align with the diagrams (Fig. 1-C). In case of absent requirements, we manually reconstructed the requirements from the diagram. In case a diagram introduced states or transitions not present in the requirements, we added the missing requirements. To mitigate bias during manual reconstruction, we enforced two measures: (1) independent dual review – a second researcher independently verified each reconstructed requirement–diagram pair; and (2) consensus resolution – disagreements were resolved through discussion. These steps ensured that reconstructions were driven by diagram evidence rather than subjective assumptions.

After aligning the requirements with the diagrams, we standardized the requirements text of each diagram into a *structured requirements* format (Fig. 1-D). We represented each requirements text as a list of individual requirements so that it is easier and clearer for the LLM to understand when used in prompts. Finally, we generated a corresponding diagram file in PlantUML [40] format (Fig. 1-E). We used PlantUML because its text-based format supports automated syntactic checks, reproducible rendering, and use in software engineering workflows [36]. Each diagram was verified for syntax through PlantUML code compilation and assessed for correctness against the requirements.

Each final data point consists of five artifacts: *raw requirements text* (unstructured natural language), *aligned requirements text* (used when requirements were missing or incomplete), *structured requirements text* (requirements in a standardized list format), *book diagram* (diagram in PNG format from the textbook), and *PlantUML code* (ground truth state transition

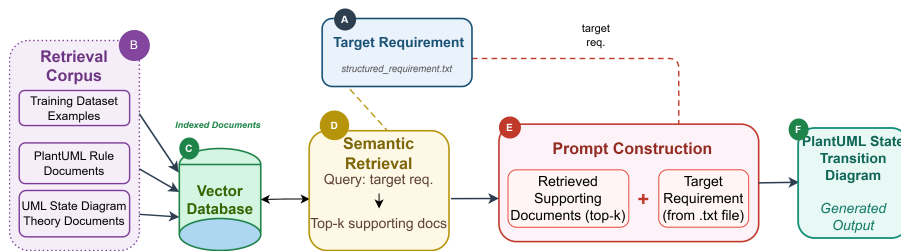
diagram in textual UML format). The final dataset contains 80 requirement-diagram pairs. All five artifacts for each case are provided in the companion repository [14]. By addressing data incompleteness, enforcing semantic alignment, and standardizing requirements and diagram representations, our dataset construction process supports reproducible and consistent evaluation of LLM-based state transition diagram generation, following empirical principles of software engineering [42]. We maintained multiple artifact representations to ensure traceability and flexibility for future research.

To map diagrams with varying complexity, we stratified data points by the number of states. Based on the empirical distribution (25th percentile = 6 states, 75th percentile = 14 states), we established three categories: **Simple** (≤ 6 states), **Medium** (7–14 states), and **Complex** (> 14 states). This stratification informed few-shot example selection and facilitated performance analysis. The corpus, spanning diverse domains like ATM/banking, healthcare, and business workflows, includes complex behavioral constructs. Across 80 cases, diagrams frequently exhibit choice nodes, self-transitions, negative/exception guards, temporal triggers, composite states, and entry/do/exit actions.

3.2 Diagram Generation Using Different Prompting Strategies

For our empirical exploration of state transition diagram generation, we used four open-source LLMs: Meta-Llama-3.1-8B-Instruct [22], DeepSeek-R1-Distill-Qwen-14B [23], Mistral-7B-Instruct [30], and Qwen2.5-7B-Instruct [58]. These models were chosen for their recent use in software engineering literature [12, 56, 62] and their availability for reproducible research. All generations used a temperature of 0.2, top-p of 0.9, and a maximum of 1,024 output tokens per call, with a 300-second timeout. Models were hosted locally via Ollama. All models used Q4_K_M quantization. From the dataset, we randomly selected 27 data points as the test dataset, using the remaining 53 for prompting and retrieval. For each requirement in the test dataset, we generated PlantUML code using three prompting strategies:

- **Zero-shot:** Only structured requirements text and task instructions were provided.
- **One-shot:** Structured requirements text, task instructions, and one requirement-diagram example (outside the test dataset) were provided.
- **Few-shot:** Structured requirements text, task instructions, and three examples (one from each complexity category: simple, medium, and complex) were provided.



■ **Figure 2** RAG Construction Workflow: retrieval corpus construction, vector database indexing, semantic retrieval, and prompt augmentation.

Diagram Generation Using RAG

In addition to different prompting strategies, we explored RAG’s capability for state transition diagram generation. Our RAG pipeline, illustrated in Figure 2, enhances prompts with dynamically retrieved contextual information from a retrieval corpus (Fig. 2-B). This

corpus includes: (1) 53 requirement-diagram pairs outside the test dataset, (2) PlantUML documentation, and (3) theoretical resources on UML state transition diagrams. We used ChromaDB with its default all-MiniLM-L6-v2 embedding model and squared L2 distance. Retrieval was source-aware, using up to two rule documents, two theory documents, and three dataset examples. All documents were indexed in a vector database using semantic embeddings (Fig. 2-C). The system performs semantic vector retrieval over these indexed documents (Fig. 2-D), returning the top-ranked supporting documents.

The prompt is constructed using the retrieved supporting documents and the target structured requirements text (Fig. 2-E). Finally, the LLM generates the corresponding PlantUML code for the state transition diagram (Fig. 2-F). The construction details of the retrieval corpus, including document sources and document-level indexing, are documented in the artifact repository [14]. To isolate the contribution of each knowledge source in the retrieval corpus, we conducted an analysis with four RAG configurations: (1) full corpus, (2) examples only, (3) rules only, and (4) theory only. This allowed us to assess how different retrieval knowledge sources affect diagram generation performance.

3.3 Diagram Evaluation

Evaluating the quality of generated UML state transition diagrams is inherently multi-faceted. Some aspects can be assessed automatically, such as *syntactic validity*, which checks whether the PlantUML code follows the language grammar, and *structural validity*, which checks whether the diagram follows UML state transition diagram rules. However, other aspects require human judgment, such as whether the diagram correctly captures the intended behavior, is complete with respect to the requirements, is understandable to a human, and uses terminology consistently with the requirements. Therefore, we used a two-stage evaluation process. First, we performed automated evaluation of syntactic and structural validity for every generated diagram. Then, we conducted human evaluation on a subset of diagrams that passed both automated stages. For human evaluation, we used quality dimensions used by other researchers for UML diagram evaluation [56, 36].

3.3.1 Automated Evaluation

Automated evaluation consists of two stages: **syntactic validity** and **structural validity**.

Both stages were applied to every generated PlantUML diagram before human assessment. Syntactic validity was assessed using the PlantUML command-line tool with the `plantuml-checkonly` flag. If the code is correct, it completes without errors; if there are violations, the tool reports them. A diagram is syntactically valid only if no errors are reported.

Diagrams that pass this check are evaluated for structural validity using our rule-based graph analyzer. First, the PlantUML code is normalized by removing markdown fences and extracting the `@startuml ... @enduml` block to retain only the diagram code. This normalized code is parsed into a graph representation, capturing states, transitions, starting points, ending states, and special elements. The analyzer stores each state and records transitions, along with source states, destination states, and transition labels or conditions. It also tracks the initial state, final states, and special elements like choice, fork, join, and history states.

The rule-based graph analyzer checks the graph against the structural rules in Table 1, each implemented as a deterministic check. A diagram passes structural validity only if all rules are satisfied; otherwise, the specific violation types are recorded.

The row (`parse_warning`) captures syntactically valid PlantUML code with ambiguities or non-standard constructs that hinder reliable graph extraction. The rule-based graph analyzer, detailed in the artifact repository [14], implements rules in Table 1. These rules operationalize

key well-formedness constraints from the UML state-machine specification [38], including requirements for a single initial state, state reachability, and correct pseudostate usage (choice, fork, join). Checks for missing final states and orphan states are also included as practical quality heuristics to catch common LLM omissions and ensure minimal completeness.

■ **Table 1** Structural validity rules, implementation checks, and violation types.

Rule	Implementation check	Violation type(s)
Exactly 1 initial state	initial_targets has length exactly 1.	missing initial state; multiple initial states
At least 1 final state	final_states is not empty.	missing final state
No duplicate transitions	len(transitions) equals len(set(transitions)).	duplicate transitions
No unreachable states	DFS from initial state; all states visited.	unreachable states
No orphan states	Every declared state has incoming or outgoing transitions.	orphan states detected
Choice node has outgoing transitions	If $\langle\langle\text{choice}\rangle\rangle$, at least one outgoing transition exists.	choice node without outgoing transitions
Choice node outgoing transitions are guarded	Each outgoing transition from a choice node has a label starting with [.	choice node without guarded outgoing transitions
Fork node has ≥ 2 outgoing branches	If $\langle\langle\text{fork}\rangle\rangle$, outgoing_count ≥ 2 .	fork without multiple outgoing branches
Join node has ≥ 2 incoming branches	If $\langle\langle\text{join}\rangle\rangle$, incoming_count ≥ 2 .	join without multiple incoming branches
History state used with composite state	If $[H]$ or $[H*]$ appears, a composite state must exist.	history state used without composite state
Valid graph extraction	Graph extraction completes without parser-level warnings.	parse warning

3.3.2 Human Evaluation

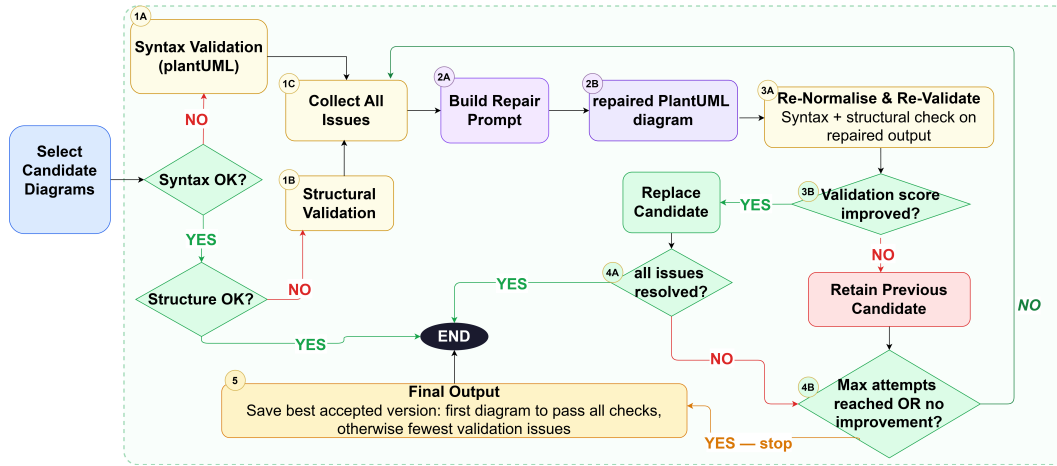
Diagrams that passed both syntactic and structural validation were further assessed through human evaluation. Each selected diagram was independently evaluated by two evaluators with prior academic or industry experience in UML state transition diagrams and software requirements modeling. The evaluators assessed the diagrams using four quality dimensions, following established practices for evaluating generated UML models [56, 36]: **Completeness** (all required states, transitions, events, guards, and initial/final states described in the requirements are present), **Correctness** (diagram’s behavior aligns with the intended system behavior in the requirements, without contradictions or invalid flows), **Understandability** (diagram is clear, readable, and free from unnecessary complexity or redundant elements), and **Terminological Alignment** (consistency of state names, transition labels, and behavioral terminology with requirements text is a key indicator for hallucination [36]).

Both evaluators independently rated each criterion using a 5-point Likert scale (1 = very poor, 5 = excellent). Inter-rater agreement was measured using Cohen’s Kappa coefficient (κ). Unlike purely syntactic or structural evaluations, state transition diagram modeling is a subjective modeling task. Multiple valid models can satisfy the same requirements because of differences in abstraction level, state granularity, error handling strategies, or modeling patterns. Therefore, a κ value above 0.41, indicating moderate agreement, was

considered acceptable. For correctness, moderate agreement is appropriate as evaluators may legitimately disagree on whether an omission is an error or a deliberate choice. In cases of lower agreement ($\kappa \leq 0.41$) or a Likert scale difference of two or more points, evaluators discussed the diagram to reach a consensus score. The evaluation form and scoring guidelines are provided as “Evaluation Form” in the companion repository [14].

3.4 Iterative Repair Workflow

To improve the syntactic and structural validity of generated diagrams, we explored an iterative repair workflow. This workflow was applied using the strongest initial strategy for each LLM, i.e., the strategy that produced the highest number of syntactically and structurally valid diagrams during the initial automated evaluation (see Section 4 for the per-LLM ranking). For each selected strategy, only diagrams from the test dataset that failed automated evaluation entered the iterative repair workflow. The repair process follows the looping mechanism illustrated in Figure 3:



■ Figure 3 Validation-aware iterative repair workflow.

Evaluation and Issue Collection. The candidate PlantUML diagram is checked using automated evaluation. This includes syntactic validity checking with `plantuml -checkonly` and structural validity checking using the rule-based graph analyzer described in Section 3.3.1. All detected issues, including syntactic error messages and structural violation types from Table 1, are collected into a list.

LLM Repair. The same LLM used for generation receives a repair prompt with instructions and identified issues, then generates new PlantUML code.

Re-evaluation and Acceptance. The repaired output is normalized and re-evaluated for syntactic and structural validity. If it reduces the recorded validation issues, it replaces the current candidate. Otherwise, the previous version is retained.

Iteration. The repair loop is repeated while validity issues remain or repair attempts are still available. The loop stops when the diagram passes all checks or the maximum repair attempt limit is reached.

Final Output. After the repair loop ends, the best accepted version is saved as the final diagram. This is either the first version that passes all checks or the version with the fewest validity violations.

The repair instruction is deliberately conservative: modify only necessary lines, avoid redesigning the diagram, and refrain from adding unsupported states or transitions. This prevents the LLM from hallucinating large behavioral changes while still allowing targeted corrections. The effectiveness of this workflow is evaluated by comparing the automated and human evaluation scores of the repaired diagrams against their initial counterparts. Results are presented in Section 4. The full repair workflow configuration, including prompt, iteration logging, and acceptance criteria, is available in the artifact repository [14].

4 Results & Discussion

4.1 RQ1: Impact of Prompting Strategies on Validity

For RQ1, we have evaluated the validity of diagrams generated using zero-shot, one-shot, few-shot, and RAG. Table 2 reports syntactic and structural validity across these strategies.

Finding 1: PlantUML examples provide essential guidance for syntactic validity. Zero-shot prompting yields low syntactic validity (lowest–Mistral 3.7%, highest–DeepSeek 59.26 %). Adding an example (one-shot) drastically (58% on average) improves syntactic validity for most models (lowest - Qwen 74.07%, highest - Llama 96.3 %), and few-shot further consolidates this gain by 6.5% on average (lowest - DeepSeek 88.9%, highest - Qwen 96.3%). This shows that LLMs learn the basic syntax of PlantUML (keywords, arrows, brackets) effectively even from a few examples (Table 2, Figure 4a).

Table 2 Automatic and Human Evaluation Results Across Prompting Strategies and Models (Llama as LM, Mistral as MT, DeepSeek as DS, Qwen as Qwn).

Evaluation Type	Metric	Zero-shot				One-shot				Few-shot				RAG			
		LM	MT	DS	Qwn	LM	MT	DS	Qwn	LM	MT	DS	Qwn	LM	MT	DS	Qwn
Automatic	Syntactic	29.6%	3.7%	59.3%	22.2%	96.3%	85.2%	92.6%	74.1%	92.6%	96.3%	88.9%	96.3%	100%	74.1%	85.2%	100%
	Structural	0%	0%	0%	0%	18.5%	3.7%	11.1%	3.7%	29.6%	7.4%	22.2%	11.1%	22.2%	14.8%	14.8%	14.8%
Human	Complete	–	–	–	–	4	5	4	3.5	3	4	4	4	3	4.5	4.5	3
	Correct	–	–	–	–	4	4.5	3.5	2.5	3	4	4	4	3	4	4.5	3.5
	Understandability	–	–	–	–	5	4.5	3.5	2.5	5	4.5	5	5	4.5	5	5	4.5
	Term. alignment	–	–	–	–	5	3.5	4	3	5	4.5	5	5	4.5	4	5	3.5

Finding 2: Structural validity remains low across all prompting strategies. Even with few-shot or RAG, structural validity never exceeds 29.6% (Llama few-shot). The dominant violation types are missing final states, missing initial states, multiple initial states, and unreachable states (see Table 3). This means the failures are not random; they are systematic omissions of required elements. Consequently, passing syntactic checks gives false confidence; a diagram can be valid PlantUML yet still be structurally invalid.

Finding 3: Few violation types repeated across many invalid diagrams. Across all models, the mean number of structural violations per diagram is low. Yet the same violation types (e.g., missing final state) appear repeatedly, suggesting that LLMs nearly understand the rules but consistently miss a few critical elements. Figure 4b shows the average violation counts per model and strategy.

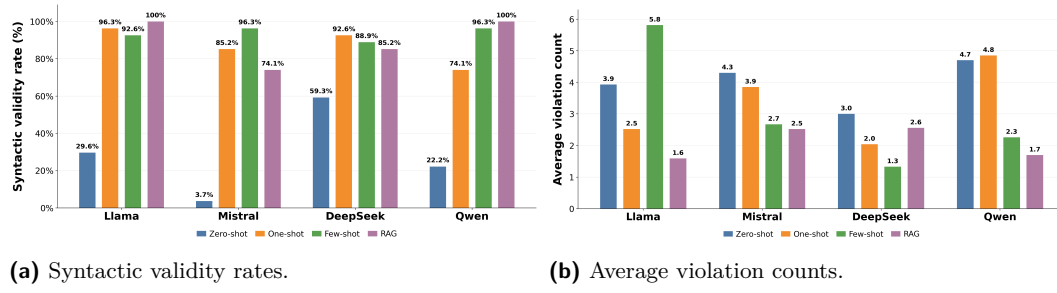


Figure 4 Syntactic validity rates across prompting strategies (Figure 4a), and average structural violation counts per diagram (Figure 4b).

Table 3 Counts of structural violations by model and prompting strategy (Z=zero-shot, O=one-shot, F=few-shot, R=RAG). Models: L=Llama, M=Mistral, D=DeepSeek, Q=Qwen.

Violation	Llama				Mistral				DeepSeek				Qwen			
	Z	O	F	R	Z	O	F	R	Z	O	F	R	Z	O	F	R
missing final state	25	10	12	9	27	16	16	11	27	21	16	17	27	18	13	10
orphan states	30	8	6	0	36	22	18	16	16	4	2	14	52	22	6	4
unreachable states	14	26	20	12	0	28	24	6	0	6	2	6	0	40	34	10
invalid choice node	3	105	3	–	0	16	2	5	0	0	3	3	0	9	1	1
missing initial state	8	2	2	0	27	2	3	4	27	13	8	7	27	1	0	0
syntactic error	19	1	2	0	26	4	1	7	11	2	3	4	21	7	1	0
invalid choice guards	0	9	7	10	0	12	4	7	0	1	3	10	0	27	1	1
multiple initial states	6	3	2	6	0	4	3	10	0	5	0	7	0	4	2	13
duplicate transitions	6	1	3	3	0	1	2	2	0	1	2	1	0	3	3	5

RQ1 Takeaway: Examples help syntax but not structure. Structural validity stagnates at around 30%, with missing initial/final states as the dominant, recurring errors. As a result, syntax check alone is insufficient for reliability.

4.2 RQ2: Retrieval Corpus Analysis – Which Retrieval Source Helps?

To understand how different document sources in the retrieval corpus affect diagram validity, we evaluated RAG using three separate settings: examples only, rules only, and theory only. Table 4 shows structural validity for the RAG configurations; the full-corpus results are shown in Table 2.

Finding 4: RAG is not uniformly reliable; different sources work best for different models.

For instruction-tuned models (Llama, Qwen), rules-only RAG is most effective (Llama 42.9%, Qwen 50.0%). For Mistral, full RAG (examples + rules + theory) gives the best result (14.8%), while examples-only (4.2%) and rules-only (5.6%) perform poorly. Theory-only RAG is uniformly ineffective (0–5% structural validity). Thus, retrieval corpus design must be model-aware. No single configuration works best for all LLMs.

We identified a failure case demonstrating the unreliability of examples-only RAG. A human evaluator noted that “requirements 3–6 are missing but adding new things.” The retrieved example was textually similar but behaviorally different from the target requirement. The generated diagram omitted required states and transitions from the target case and included added ones from the retrieved example, indicating a retrieval-induced hallucination where the model borrowed unsupported behavior from a relevant but different example.

RQ2 Takeaway: Reliability of retrieval document source depends on the model. Rules-only helps instruction-tuned models, whereas theory-only performs very poorly. Practitioners must tailor retrieval to the target LLM.

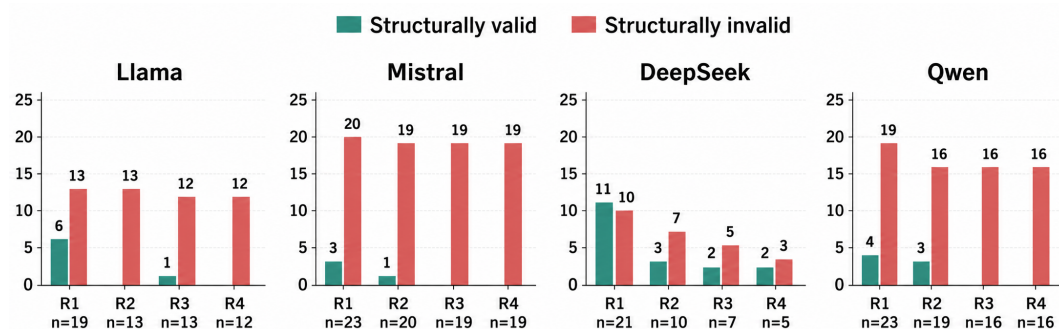
■ **Table 4** Evaluation Results for RAG Corpus Analysis Settings (Llama as LM, Mistral as MT, DeepSeek as DS, Qwen as Qwn).

Evaluation Type	Metric	RAG (Only Rules)				RAG (Only Examples)				RAG (Only Theory)			
		LM	MT	DS	Qwn	LM	MT	DS	Qwn	LM	MT	DS	Qwn
Automatic	Syntactic	77.8%	66.7%	81.5%	81.5%	96.3%	88.9%	74.1%	92.6%	88.9%	25.9%	63.0%	81.5%
	Structural	42.9%	5.6%	18.2%	50.0%	15.4%	4.2%	10.0%	12.0%	4.2%	0%	0%	4.6%
Human	Complete	5	2	4.5	4	3.5	5	5	5	5	—	—	5
	Correct	4	2	4.5	4	3	3	5	4	3	—	—	5
	Understandability	5	3	5	5	4.5	4	4.5	5	5	—	—	5
	Term. alignment	5	4	5	5	5	5	5	5	5	—	—	5

4.3 RQ3: Effectiveness of Iterative Repair

To improve invalid diagrams, we explored how iterative repair may increase their structural and syntactic validity. Table 6 presents structural validity outcomes after up to five repair iterations. DeepSeek performed best among all models, achieving 88.89% structural validity after repair, compared to its few-shot baseline of 22.2%. Llama, Qwen, and Mistral also improved, but to a lesser extent (55.6%, 40.7%, and 29.6%, respectively).

Finding 5: More iterations do not always improve repair. Most successful repairs occurred within the first two iterations. As shown in Figure 5, DeepSeek made 11 diagrams valid after the first iteration, but only 2 after the fourth iteration. No additional diagram became valid after the fifth iteration. Across all models, the number of newly repaired diagrams drops sharply after the first two iterations. This demonstrates that repair yields limited improvement beyond 2–3 iterations, and excessive repair attempts are not productive.



■ **Figure 5** Repair success per iteration. Most valid diagrams are obtained within the first two iterations; later iterations contribute few or no improvements.

Finding 6: Repair is effective for diagrams with fewer issues but struggles with many issues. Diagrams with multiple faults often require more iterations and rarely validate after three attempts. We noted that when violations exceeded two, DeepSeek aggressively attempted fixes, sometimes deleting states and transitions, which removed essential behavioral elements. This indicates that while repairs can manage moderate complexity, overly aggressive corrections may compromise completeness for structural validity.

Finding 7: Repair success varies by violation type and model. Table 5 shows that DeepSeek successfully fixes 100% of missing final states, missing initial states, duplicate transitions, orphan states, and invalid choice guards, but fails to address unreachable states (0%). Llama also achieves 100% for missing final states but struggles with other violations, particularly invalid choice guards (0%) and duplicate transitions (0%). Mistral excels at fixing duplicate transitions (100%) but has low success rates for missing final states (27.3%), missing initial states (25.0%), and multiple initial states (20.0%). Qwen is strong on missing final states (90.0%) but performs poorly on multiple initial states (15.4%) and cannot fix invalid choice nodes or guards. These results indicate that some models, like DeepSeek, offer broader repair capabilities, while others are more specialized or limited.

■ **Table 5** Repair success rates per violation type (percentage of instances fixed).

Violation Type	DeepSeek	Llama	Mistral	Qwen
missing_final_state	100% (16/16)	100% (12/12)	27.3% (3/11)	90.0% (9/10)
missing_initial_state	100% (8/8)	50.0% (1/2)	25.0% (1/4)	–
duplicate_transitions	100% (2/2)	0% (0/1)	100% (2/2)	60.0% (3/5)
orphan_states	100% (1/1)	66.7% (2/3)	25.0% (2/8)	50.0% (1/2)
unreachable_states	0% (0/1)	60.0% (6/10)	0% (0/3)	40.0% (2/5)
multiple_initial_states	–	50.0% (1/2)	20.0% (2/10)	15.4% (2/13)
invalid_choice_node	–	40.0% (2/5)	75.0% (3/4)	0% (0/1)
invalid_choice_guards	100% (1/1)	0% (0/1)	33.3% (1/3)	0% (0/1)

RQ3 Takeaway: Repair could not improve diagrams with a high number of issues, even after many iterations. This suggests that repair should be used as a targeted tool for small corrections, not as a brute-force method to rewrite entire diagrams. DeepSeek’s reasoning capability made it the strongest repair performer. However, even DeepSeek struggled when the initial diagram contained a high number of issues.

■ **Table 6** Evaluation Results for Iterative Repair.

Evaluation Type	Metric	Repair			
		Llama	Mistral	DeepSeek	Qwen
Automatic	Syntactic	96.30%, n=26	74.07%, n=20	92.59%, n=25	92.59%, n=25
	Structural	55.56%, n=15	29.63%, n=8	88.89%, n=24	40.74%, n=11
Human	Completeness	4	4	4	4
	Correctness	4	5	4	4
	Understandability	5	5	5	3
	Terminological alignment	4	5	5	4

4.4 RQ4: Human Evaluation of Quality Dimensions

To evaluate the quality of valid generated diagrams, we collected human evaluation scores for correctness, completeness, understandability, and terminological alignment.

Finding 8: Even incorrect diagrams can be understandable – a concerning disconnect. For few-shot prompting, median understandability scores are 5 (excellent) for Llama, DeepSeek, and Qwen, while correctness scores are only 3–4. LLMs produce visually clear, readable diagrams (logical state names, clean layout) that *look* correct on the surface, but often miss

critical behavior. Practitioners may be misled by these aesthetically pleasing but incorrect diagrams. Consider the following real critique from our human evaluation of a microwave oven state diagram:

“Some confusion regarding transitions... after “CheckExpiry”, we should notify the user, but the diagram lacks this. The core user choice requirement is completely ignored ...; instead it makes up “get amount”. No transition for when cooking finishes naturally ...”

This example shows that a diagram can be clear but still miss behavioral flows, overlook key requirements, and include incorrect transitions. Such cases highlight a gap between perceived and actual quality in LLM-generated state machines.

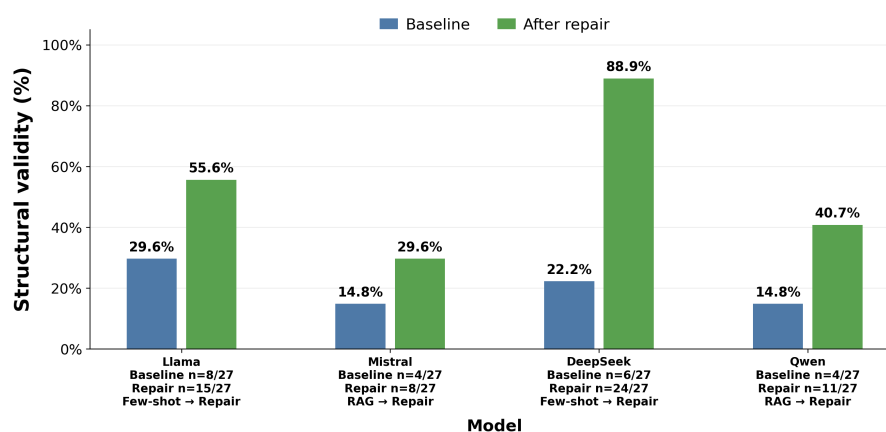
Finding 9: Few-shot prompting yields better behavioral quality than RAG. Despite RAG achieving higher syntactic validity, human-rated correctness and completeness are consistently higher for few-shot than for RAG. Fixed examples provide more reliable behavioral guidance than retrieved examples, which can introduce irrelevant or misleading content.

RQ4 Takeaway: Few-shot prompting is more reliable than retrieval for diagram quality. However, visually understandable diagrams may still be incorrect. This indicates that LLM-based diagram generation requires careful human oversight.

4.5 Cross-RQ Findings

Finding 10: Syntactic validity is necessary but insufficient for behavioral modeling. A diagram can be perfectly valid PlantUML (e.g., Llama RAG 100% syntax) yet fail all structural rules (only 22% structural validity) (as shown in Table 2). Syntax checks give a false sense of security.

Finding 11: Structural validity does not guarantee human-perceived correctness. We observed cases where diagrams passed all structural rules, such as having a single initial state and no unreachable states, but still received low human correctness scores (2–3) because they missed critical transitions. This shows that automated validation alone is not sufficient and must be supported by human oversight.



■ **Figure 6** Baseline structural validity (best strategy) vs. final validity after repair. DeepSeek shows the largest gain (22.2% → 88.9%); all models benefit substantially from iterative repair.

Finding 12: Repair is the best technique – improves both automated and human evaluation. DeepSeek achieved the highest repair success rate for structural validity (88.9%), followed by Llama (55.6%), Qwen (40.7%), and Mistral (29.6%) (Figure 6). This success is mainly due to targeted feedback: the LLM receives precise structural violation messages (e.g., `missing_final_state`) and is instructed to modify only necessary lines. Importantly, diagrams that passed structural validation after repair were also highly rated by human evaluators across all quality dimensions, a trend consistent in both few-shot and RAG-based repair settings. Fixing structural issues consequently improved the behavioral quality of the diagrams. This suggests that automated repair success is a strong indicator of human-perceived quality. Therefore, repair is not only the most effective strategy for structural validity, but also the most trustworthy strategy for practical use.

Finding 13: Repair effectiveness is model-dependent. Models that struggle with initial structural validity (Mistral, Qwen) also see lower repair success (29.6%, 40.7%) compared to DeepSeek (88.9%). The repair loop cannot compensate for fundamental limitations in a model’s ability to understand state transition diagram semantics.

4.6 Practical Recommendations for AI-Assisted Behavioral Modeling

Based on our findings, we provide three recommendations for practitioners using LLMs for state transition diagram generation. **(1) Prioritize few-shot prompting.** Use 3–4 diverse examples (simple, medium, complex) that are structurally correct and domain-aligned. Few-shot prompting improves semantic fidelity and reduces retrieval-induced hallucinations. **(2) Integrate iterative repair.** Most diagrams (70–80%) will have structural violations. Implement an automated repair loop with 2–3 iterations; additional iterations show diminishing returns. This significantly improves structural validity (e.g., DeepSeek from 22.2% to 88.9%). **(3) Incorporate human review for final assessment.** Automated checks alone cannot verify behavioral semantics or completeness. Use automated validation as a filter, followed by domain expert review of a small subset to catch semantic mismatches. Human review is mandatory for safety-critical systems.

In ambiguous industrial settings, adapt the workflow: **(1) Pre-generation clarification:** Prompt the LLM to list open behavioral questions (e.g., missing error handlers) for analyst resolution before generation. **(2) Project-specific retrieval:** Replace textbook examples with the organization’s approved state machines and domain glossaries for consistency. **(3) Failure-as-feedback:** If structural repair fails, use the violation log as a structured audit checklist for stakeholders, rather than forcing the LLM to hallucinate missing behavior. This redefines the LLM as a requirements auditor, not an autonomous generator.

4.7 Limitations and Future Work

Despite the comprehensive evaluation, several limitations remain. The dataset, while systematically constructed, consists of textbook examples that may not capture industrial complexity (concurrency, timing constraints, hierarchical states). The repair loop was capped at 5 iterations based on pilot experiments showing no improvements beyond this point, but more iterations might benefit certain cases. Human evaluation, while rigorous, involved only two evaluators; larger-scale validation with diverse practitioners would strengthen generalization. Our findings are limited to the four open-source models evaluated (7B–14B parameters) and may not generalize to larger, newer, or commercial models (e.g., GPT-4, Claude) that possess greater reasoning capacity. Additionally, each configuration was executed once due

to the combinatorial scale of our design (27 test cases $\times$ 4 models $\times$ 4 prompting strategies $\times$ 4 RAG ablations + repair evaluations) and the cost of human assessment; LLM generation is inherently non-deterministic, and output variability across runs remains uncharacterized.

Future work should explore: (a) hierarchical state transition diagrams and orthogonal regions, which introduce additional validation complexity; (b) few-shot example selection optimization (currently random, but semantic similarity-based selection may improve performance); (c) using LLMs as judges to automate or augment human evaluation of diagram quality; (d) providing structural rules themselves as part of the RAG context to assess whether explicit rule injection improves structural validity; and (e) analyzing oscillation and regression during repair – specifically, tracking per-iteration errors, fixes, and fix patterns to understand why some models regress and how to design more stable repair prompts; and (f) systematic characterization of output variability across multiple runs to assess whether our findings hold stably and whether certain strategies yield more consistent outputs.

5 Threats to Validity

The research identifies several potential threats to the validity of the findings.

Construct validity: While PlantUML syntactic and structural validity offer a quantitative measure of diagram correctness, they may not capture a diagram’s practical usefulness in development workflows. Human evaluation metrics (correctness, completeness, understandability, terminology alignment) complement automated metrics for a more comprehensive quality assessment.

Internal validity: The main internal threat is variability in LLM responses based on prompt construction. A standardized prompt template was utilized for all experiments. The non-deterministic nature of autoregressive generation can produce outliers; we addressed this by running each generation once per configuration and focusing on systematic patterns across 27 test cases per condition.

External validity: External validity is limited due to the use of textbook case studies, which may not fully represent the complexities found in actual industrial documents. Textbook examples offer a controlled baseline, but real-world requirements are often messy and incomplete. Future work should evaluate workflows on industrial requirements.

Human evaluator bias: Human evaluation introduces subjectivity, especially in assessing behavioral correctness and understandability. To mitigate this, diagrams were independently evaluated by two UML-experienced evaluators using predefined criteria and scoring guidelines. Inter-rater agreement was measured with Cohen’s Kappa coefficient ($\kappa = 0.44$ – 0.68 across dimensions), with disagreements resolved through consensus.

Reliability of reconstructed requirements: Some requirements were partially reconstructed from textbook diagrams due to missing specifications. Despite manual verification, this reconstruction may introduce implicit assumptions.

Generality of validation heuristics: Some structural validation rules are based on practical modeling heuristics rather than strict UML standards. While these heuristics enhance automated diagram quality assessment, alternative interpretations may exist. The full rule set is available in the artifact repository [14] for replication and adaptation.

LLM training data contamination: A significant threat is that the evaluated LLMs may have been pre-trained on the same textbooks and UML resources as our dataset. If models “saw” the case study diagrams during training, their performance could be inflated due to memorization. While we cannot rule out contamination, we note: (a) we used structured requirement reformulations differing from original textbook wording; (b) we required

generation in PlantUML syntax, unlikely to appear verbatim in training corpora; and (c) the poor zero-shot performance across all models (0% structural validity) suggests that memorization alone is insufficient, as models still fail without explicit prompting. Future work should validate findings on newly created, non-public requirements.

6 Conclusion

This work contributes one of the first validation-aware empirical frameworks for studying AI-assisted behavioral model generation, combining automated structural verification, retrieval-based prompting, iterative repair, and human-centered evaluation. Using a novel dataset of 80 requirement–state transition diagram pairs derived from software engineering textbooks, we evaluated four open-source LLMs across zero-shot, one-shot, few-shot, RAG, and repair workflows. Our key findings are: (1) Few-shot prompting significantly outperforms zero-shot and one-shot for syntactic validity, but structural validity remains low (30%) across all models, with missing final and initial states as common errors. Passing a syntax check alone gives false confidence. (2) RAG’s effectiveness is model-dependent. Rules-only RAG works best for instruction-tuned models (Llama, Qwen), while theory-only RAG is uniformly ineffective. RAG can induce hallucination by copying irrelevant content from retrieved examples. (3) Iterative repair dramatically improves structural validity, especially for DeepSeek (from 22.2% to 88.9%), with most repairs succeeding within the first two iterations. Repair is most effective for small, localized errors; diagrams with many violations rarely become valid. Repair success also varies by violation type – DeepSeek fixes missing final states and missing initial states perfectly but cannot handle unreachable states. (4) Human evaluation shows a disconnect: diagrams are rated high for understandability (often 5/5) but low for correctness (3–4/5). Few-shot prompting aligns better with requirements than RAG, and diagrams passing automated validation after repair receive high human ratings, making repair the most reliable strategy. Taken together, these results demonstrate that reliable AI-assisted behavioral modeling is achievable not through larger prompts or more retrieval, but through targeted, validation-aware workflows that combine few-shot examples with iterative repair. Practitioners can adopt our validation rules and repair prompts to integrate LLMs into their modeling processes as collaborative assistants rather than autonomous generators.

Data Availability Statement

All artifacts generated and used in this study are publicly available at: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow

References

- 1 Sharif Ahmed, Arif Ahmed, and Nasir U Eisty. 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, 2022. doi:10.1109/SERA54885.2022.9806783.
- 2 Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, Boston, MA, 2nd edition, 2005.
- 3 Felix Bachmann, Len Bass, and Mark Klein. Preliminary design of arche: A software architecture design assistant. Technical Report CMU/SEI-2003-TR-021, Software Engineering Institute, Carnegie Mellon University, 2003. doi:10.1184/R1/6582149.v1.
- 4 Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: Prospects, challenges, and a case study. In *Bridging the Gap Between AI and Reality: First International Conference, AISoLA 2023, Crete, Greece, October 23–28, 2023, Proceedings*, pages 355–374, 2023. doi:10.1007/978-3-031-46002-9_23.

- 5 Simon Bennett, Steve McRobb, and Ray Farmer. *Object-oriented systems analysis and design using UML*. McGraw Hill Higher Education, 2006.
- 6 Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)*, 2025. arXiv preprint arXiv:2506.18824. doi:10.48550/arXiv.2506.18824.
- 7 Humberto Cervantes, Rick Kazman, and Yuanfang Cai. An llm-assisted approach to designing software architectures using add, 2025. doi:10.48550/arXiv.2506.22688.
- 8 Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Daniel Varró. Automated domain modeling with large language models: A comparative study. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 162–172, 2023. doi:10.1109/MODELS58315.2023.00037.
- 9 Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of uml models: a systematic review of research and practice. *Softw. Syst. Model.*, 18(3):2313–2360, June 2019. doi:10.1007/s10270-018-0675-4.
- 10 Maria Stella De Biase, Simona Bernardi, Stefano Marrone, José Merseguer, and Angelo Palladino. Completion of sysml state machines from given-when-then requirements. *Software and Systems Modeling*, 23(6):1455–1491, 2024. doi:10.1007/S10270-024-01228-3.
- 11 Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of llms and transformation by example. *Proceedings of the ACM on Software Engineering*, 1(FSE):631–653, 2024. doi:10.1145/3643755.
- 12 Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomas Cerny, Karthik Vaidhyanathan, Valentina Lenarduzzi, and Davide Taibi. Generative ai for software architecture. applications, challenges, and future directions, 2025. doi:10.48550/arXiv.2503.13310.
- 13 Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir. mashiat0808/State_Transition_Diagram_LLM_workflow. Software, swbId: swb:1:dir:eb2ad1f0b722352d509a2d8d84762ee769daaa89 (visited on 2026-09-21). URL: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow, doi:10.4230/artifacts.28071.
- 14 Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir. State transition diagram llm workflow, 2026. URL: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow.
- 15 Nick Feng, Lina Marsso, Sinem Getir Yaman, Isobel Standen, Yesugen Baatartogtokh, Reem Ayad, Victória Oldemburgo de Mello, Beverley Townsend, Hanne Bartels, Ana Cavalcanti, Radu Calinescu, and Marsha Chechik. Normative requirements operationalization with large language models. In *Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference (RE 2024)*, pages 129–141, 2024. doi:10.1109/re59067.2024.00022.
- 16 Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. Model generation with llms: From requirements to uml sequence diagrams, 2024. arXiv:2404.06371.
- 17 Jeff Garland and Richard Anthony. *Large-scale software architecture: a practical guide using UML*. John Wiley & Sons, 2003.
- 18 Soham Ghosh and Gaurav Mittal. Advancing engineering research through context-aware and knowledge graph-based retrieval-augmented generation. *Frontiers in Artificial Intelligence*, 8:1697169, 2025. doi:10.3389/FRAI.2025.1697169.
- 19 Polydoros Giannouris and Sophia Ananiadou. Nomad: A multi-agent llm system for uml class diagram generation from natural language requirements. In *Proceedings of the 14th International Conference on Model-Based Software and Systems Engineering*, pages 257–264, 2026. doi:10.5220/0014301900004058.
- 20 Hassan Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.
- 21 Patrick Grässle, Henriette Baumann, and Philippe Baumann. *UML 2.0 in Action: A Project-Based Tutorial*. Packt Publishing, 2005.

- 22 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. The llama 3 herd of models. *arXiv preprint*, 2024. [arXiv:2407.21783](#).
- 23 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint*, 2025. [arXiv:2501.12948](#).
- 24 Mohammad Kasra Habib, Daniel Graziotin, and Stefan Wagner. Reqbrain: Task-specific instruction tuning of llms for ai-assisted requirements generation, 2025. [doi:10.48550/arXiv.2505.17632](#).
- 25 M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996. [doi:10.1109/32.508311](#).
- 26 Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), December 2024. [doi:10.1145/3695988](#).
- 27 James Ivers and Ipek Ozkaya. Will generative ai fill the automation gap in software architecting? In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, pages 41–45, 2025. [doi:10.1109/ICSA-C65153.2025.00014](#).
- 28 Munima Jahan, Zahra Shakeri Hossein Abad, and Behrouz Far. Generating sequence diagram from natural language requirements. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 39–48. IEEE, 2021. [doi:10.1109/REW53955.2021.00012](#).
- 29 Munima Jahan, Mohammad Mahdi Hassan, Reza Golpayegani, Golshid Ranjbaran, Chanchal Roy, Banani Roy, and Kevin Schneider. Automated derivation of uml sequence diagrams from user stories: Unleashing the power of generative ai vs. a rule-based approach. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 138–148, 2024. [doi:10.1145/3640310.3674081](#).
- 30 Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b, 2023. [doi:10.48550/arXiv.2310.06825](#).
- 31 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. [doi:10.1145/3747588](#).
- 32 Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, S  bastien G  rard, and Michel RV Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical software engineering*, 25(6):4427–4471, 2020. [doi:10.1007/S10664-020-09835-6](#).
- 33 Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and iterative development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
- 34 Taslim Mahbub, Dana Dghaym, Aadith Shankarnarayanan, Taufiq Syed, Salsabeel Shapsough, and Imran Zualkernan. Can gpt-4 aid in detecting ambiguities, inconsistencies, and incompleteness in requirements analysis? a comprehensive case study. *IEEE Access*, 12:171972–171992, 2024. [doi:10.1109/ACCESS.2024.3464242](#).
- 35 Russ Miles and Kim Hamilton. *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc., 2006.
- 36 Jackson Nguyen, Rui En Koe, Fanyu Wang, Chetan Arora, and Alessio Ferrari. Class model generation from requirements using large language models. *arXiv preprint*, 2026. [doi:10.48550/arXiv.2603.09100](#).

- 37 Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, 2000. doi:10.1145/336512.336523.
- 38 Object Management Group. Unified modeling language (UML) version 2.5.1. Standard formal/17-12-05, Object Management Group, December 2017. URL: <https://www.omg.org/spec/UML/2.5.1>.
- 39 Leonid Ozirkovskyy, Bohdan Volochiy, Oleksandr Shkiliuk, Mykhailo Zmysnyi, and Pavlo Kazan. Functional safety analysis of safety-critical system using state transition diagram. *Radioelectronic and Computer Systems*, 0(2):145–158, 2022. doi:10.32620/reks.2022.2.12.
- 40 PlantUML Team. Plantuml language reference guide. URL: <https://plantuml.com/guide>.
- 41 Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
- 42 Paul Ralph, Nauman bin Ali, Sebastian Baltes, et al. Empirical standards for software engineering research, 2020. arXiv:2010.03525.
- 43 Aqsa Rasheed, Bushra Zafar, Tehmina Shehryar, Naila Aiman Aslam, Muhammad Sajid, Nouman Ali, Saadat Hanif Dar, and Samina Khalid. Requirement engineering challenges in agile software development. *Mathematical Problems in Engineering*, 2021(1):6696695, 2021.
- 44 Pascal Roques. *UML in practice: the art of modeling software systems demonstrated through worked examples and solutions*. John Wiley & Sons, 2006.
- 45 Nimra Shehzadi, Javed Ferzund, Rubia Fatima, and Adnan Riaz. Automatic complexity analysis of uml class diagrams using visual question answering (vqa) techniques. *Software*, 4(4):22, 2025. doi:10.3390/SOFTWARE4040022.
- 46 Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 1628–1639, 2025. doi:10.1109/ICSE55347.2025.00005.
- 47 Ian Sommerville. *Software Engineering*. Pearson, 9 edition, 2011.
- 48 Suriya Sundaramoorthy. *UML diagramming: a case study approach*. Auerbach Publications, 2022.
- 49 Brennan Swick, Sean Donegan, Andrew Gillman, and Michael Groeber. Human planning of robot actions through llm-guided state machine synthesis. In *2024 33rd IEEE International Conference on Robot and Human Interactive Communication (ROMAN)*, pages 430–437, 2024. doi:10.1109/RO-MAN60168.2024.10731325.
- 50 Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. The future of ai-driven software engineering. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. doi:10.1145/3715003.
- 51 Beian Wang, Chong Wang, Peng Liang, Bing Li, and Cheng Zeng. How llms aid in uml modeling: An exploratory study with novice analysts, 2024. doi:10.48550/arXiv.2404.17739.
- 52 Chengjie Wang, Jingzheng Wu, Xiang Ling, Tianyue Luo, and Chen Zhao. Correct code, vulnerable dependencies: A large scale measurement study of llm-specified library versions. In *Proceedings of the International Conference on Software Engineering (ICSE 2026)*, 2026. arXiv preprint arXiv:2605.06279. doi:10.48550/arXiv.2605.06279.
- 53 Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. In *Proceedings of the 48th International Conference on Software Engineering (ICSE 2026)*, 2026. arXiv preprint arXiv:2503.18666. doi:10.48550/arXiv.2503.18666.
- 54 Yuchen Wang, Shangxin Guo, and Chee Wei Tan. From code generation to software testing: Ai copilot with context-based rag. *IEEE Software*, 42(4):34–42, 2025. doi:10.1109/MS.2025.3549628.
- 55 Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023. doi:10.48550/arXiv.2302.11382.
- 56 Chi Xiao, Daniel Ståhl, and Jan Bosch. Uml sequence diagram generation: A multi-model, multi-domain evaluation. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 272–283, 2025. doi:10.1109/ICSE-SEIP66354.2025.00030.

- 57 Aashish Yadavally and Tien N. Nguyen. From seed to scope: Reasoning to identify change impact sets. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ICSE '26, page To Appear, 2026.
- 58 An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint*, 2024. [arXiv:2412.15115](#).
- 59 Tao Yue, Lionel C Briand, and Yvan Labiche. atoucan: an automated framework to derive uml analysis models from use case models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–52, 2015. [doi:10.1145/2699697](#).
- 60 Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, 2025. [doi:10.1007/S10664-024-10602-0](#).
- 61 Shaohong Zhong, Andrea Scarinci, and Alice Cicirello. Natural language processing for systems engineering: Automatic generation of systems modelling language diagrams. *Knowledge-Based Systems*, 259:110071, 2023. [doi:10.1016/J.KNOSYS.2022.110071](#).
- 62 Taohong Zhu, Lucas C. Cordeiro, and Youcheng Sun. Requinone: A large language model-based agent for software requirements specification generation, 2025. [doi:10.48550/arXiv.2508.09648](#).