

Article

Large Language Models for UML Class Diagram Modeling: A Preliminary Empirical Evaluation

Yong Cheng, You Huang , Shixin Yao , Yue Qian , Liang Zhang, Xueyin Fang and Tianjun Wu

National University of Defense Technology, Changsha 410073, China; ycheng@nudt.edu.cn (Y.C.); yaoshixin22@nudt.edu.cn (S.Y.); yueqian@nudt.edu.cn (Y.Q.); zhangliang@nudt.edu.cn (L.Z.); fangxueyin@nudt.edu.cn (X.F.); wutianjun08@nudt.edu.cn (T.W.)

* Correspondence: huangyou@nudt.edu.cn

Abstract

UML class diagram modeling is a fundamental task in software engineering, yet the application of large language models (LLMs) to this domain remains underexplored. Existing studies predominantly focus on single closed-source models with simple prompting strategies, lacking systematic comparisons across model types, prompt engineering techniques, and iterative refinement approaches. In this paper, we construct a difficulty-stratified dataset of 30 UML class diagram exercises and propose an automated weighted evaluation metric over generated PlantUML code—both of which are rarely constructed and systematically applied in existing LLM-driven UML modeling research. We present a preliminary empirical evaluation comparing open-source and closed-source LLMs across multiple scales and types, diverse prompting strategies, and varying requirement complexity levels. Beyond the single-round static paradigm of prior work, we further introduce and evaluate iterative prompting schemes that continuously improve model outputs through structured feedback. Our findings reveal that chain-of-thought prompting has different effects on improving the quality of different models, that relationship modeling is the persistent bottleneck under increasing complexity, and attribute extraction remains a largely unsolved technical challenge across all tested LLMs. Further, automated feedback-driven iterative refinement yields varied improvements: it brings notable performance gains for reasoning-oriented thinking models while delivering only marginal promotion for high-performance general chat models. These results provide actionable guidance for practitioners and researchers applying LLMs to UML modeling tasks.

Keywords: large language models; UML class diagram; prompt engineering; iterative refinement



Academic Editors: Douglas O'Shaughnessy, Affan Yasin, Javed Ali Khan and Lijie Wen

Received: 8 May 2026

Revised: 22 June 2026

Accepted: 26 June 2026

Published: 1 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The Unified Modeling Language (UML) serves as the de facto standard for visualizing and documenting software system designs [1,2]. As a cornerstone of model-driven engineering, UML encompasses a rich set of diagram types—including class diagrams [3], sequence diagrams [4], use case diagrams [5], and activity diagrams [6]—each capturing a distinct structural or behavioral aspect of a software system. Constructing these diagrams from natural language requirements is a cognitively demanding task that requires both domain understanding and modeling expertise, making it a compelling target for automation [7–10].

The rapid advancement of large language models (LLMs) [11,12], including GPT-4, Claude, Gemini, and open-source alternatives such as LLaMA and Qwen, has demonstrated remarkable capabilities across a wide range of software engineering tasks, including code generation [13], bug fixing [14], test generation, and documentation [15]. These capabilities have naturally prompted researchers to explore whether LLMs can automate the construction of UML diagrams from textual requirements. Early investigations have shown that LLMs can produce syntactically valid UML artifacts and capture high-level structural patterns from requirement descriptions, suggesting genuine potential for UML modeling automation. Studies have explored LLM-based generation of various UML diagram types, including use case diagrams, sequence diagrams, and class diagrams, reporting promising but inconsistent results across different models and task settings.

Among all UML diagram types, class diagrams occupy a particularly central role. They capture the static structure of a system through classes, attributes, methods, and inter-class relationships such as inheritance, association, aggregation, composition, and dependency, forming the structural backbone upon which other diagram types are built [16]. The complexity and expressiveness of class diagrams—requiring the model to simultaneously identify entities, type their relationships, and enumerate fine-grained attributes—make them both the most informative and the most challenging UML artifact to generate automatically [17,18]. For these reasons, UML class diagram generation represents the most critical and demanding test case for evaluating LLM-based UML modeling capabilities [19,20], and is the focus of this paper.

However, despite the growing interest in this area, the research landscape for LLM-based UML class diagram generation suffers from several critical gaps. First, most existing studies evaluate only a single, typically closed-source model, preventing meaningful cross-model comparisons. Second, prompt engineering—a factor known to substantially influence LLM output quality—has received little systematic attention in the UML context. Third, the effect of requirement complexity on model performance remains uncharacterized. Fourth, virtually all prior work adopts a single-round, static evaluation paradigm, ignoring the potential of iterative refinement to progressively improve model outputs. Finally, the field lacks a standardized, publicly available dataset and a principled automated evaluation metric for UML class diagram generation.

This paper addresses all of the above gaps through a structured empirical study. Our contributions are as follows:

- We construct a standardized dataset of 30 UML class diagram exercises with ground-truth PlantUML code, difficulty-stratified into easy, medium, and hard groups using a composite scoring scheme based on class count, relationship complexity, attribute count, and Flesch–Kincaid readability scores.
- We propose an automated weighted similarity metric that evaluates generated PlantUML code against ground truth across three structural dimensions: classes, relationships, and attributes.
- We conduct systematic experiments across multiple LLM types and scales, prompting strategies, difficulty levels, and iterative refinement schemes, organized around four well-defined research questions. We also derive empirically grounded findings and practical guidelines for applying LLMs to UML class diagram modeling, covering model selection, prompting strategy, and refinement approach for different task complexity levels.

The remainder of this paper is organized as follows. Section 2 reviews related work on LLMs in software engineering, UML automation, prompt engineering and iterative refinement. Section 3 describes dataset construction, weighted evaluation metric and complete experimental design for four research questions. Section 4 presents all experimental results

and multi-dimensional analysis. Section 5 synthesizes core findings, provides practical application guidelines and analyzes multiple threats to validity. Section 6 concludes the whole research and lists promising future research directions.

2. Related Work

2.1. LLMs for Software Engineering

Large language models have been extensively applied to software engineering tasks, demonstrating strong capabilities across code generation, bug fixing, test generation, and documentation. Chen et al. introduced Codex, a GPT-derived language model fine-tuned on a large corpus of public Python code from GitHub [21]. Evaluated on HumanEval—a manually constructed benchmark comprising 164 Python programming problems—Codex achieved a pass@1 rate of 28.8%, significantly outperforming baseline models including GPT-3 and GPT-J. This work established the foundation for LLM-based code generation research and demonstrated that large-scale pre-training on code corpora enables models to solve non-trivial programming tasks from natural language docstrings alone.

Nijkamp et al. proposed CodeGen, a family of autoregressive language models with parameters ranging from 350M to 16.1B [22]. These models were trained sequentially on three corpora: the natural language corpus The Pile, the multilingual programming language corpus BigQuery, and the Python-specific corpus BigPython. A core contribution of CodeGen is the proposal of the multi-turn program synthesis paradigm, which decomposes complex programming tasks into a sequence of sub-problems solved through progressive interaction. The authors further constructed the Multi-Turn Programming Benchmark (MTPB) and verified that this multi-turn paradigm significantly improves the performance of program synthesis compared with the single-turn paradigm.

Guo et al. presented DeepSeek-Coder, a series of code-focused large language models (LLMs) with parameter sizes ranging from 1.3B to 33B, which were trained from scratch on a carefully curated corpus of 2 trillion tokens covering 87 programming languages [23]. DeepSeek-Coder proposed a repository-level code pre-training strategy that integrates cross-file contextual dependencies, enabling the model to effectively tackle project-level code completion tasks. Equipped with a 16K context window and the Fill-in-the-Middle (FIM) objective, the model achieved state-of-the-art performance among open-source code models on benchmarks including HumanEval, MBPP, and DS-1000. Notably, its 33B instruction-tuned variant outperformed closed-source models such as GPT-3.5-Turbo on most code-related tasks.

Beyond code generation, Kang et al. explored the application of large language models (LLMs) in software testing, with a specific focus on automatic bug reproduction from bug reports [24]. Their proposed LIBRO framework leverages few-shot prompting to guide LLMs in generating executable test cases for bug reproduction, which was evaluated on 750 real-world bugs from the Defects4J benchmark. The study demonstrates that this fully automated approach can successfully reproduce 33.5% of the bugs, and post-processing strategies including test injection, dependency resolution, test selection and ranking significantly improve the practical utility of the method. Hou et al. performed a systematic literature review on large language models for software engineering (LLM4SE), analyzing 395 primary studies published from January 2017 to January 2024 [25]. The review categorized LLM architectures, software engineering datasets, optimization/evaluation techniques, and 85 specific software engineering tasks, identifying code generation and program repair as the most extensively studied tasks. Crucially for our work, the review pointed out that model-driven engineering artifacts (including UML diagrams) remain one of the least explored application scenarios, with extremely few studies focusing on UML

generation in the entire corpus. This gap directly motivates our systematic research on this domain.

2.2. UML Generation and Automation

Early approaches to automated UML generation relied on learning-based natural language processing pipelines that extracted structural elements from requirement texts through syntactic and semantic analysis. Pudlitz et al. proposed a learning-based named entity recognition (NER) approach to extract system states from natural language requirements, with a focus on automotive domain requirements [26]. They developed a NER model based on bidirectional LSTMs and CNNs, combined with bootstrapping for training data generation, and designed two workflows: an almost fully automated approach with minimal manual effort and an iterative semi-automated approach with incremental user feedback. This work demonstrated that machine learning-based state extraction can drastically reduce manual labor compared with fully manual extraction, while maintaining practical performance for requirements analysis and simulation validation.

Elallaoui et al. applied NLP techniques to realize the automatic transformation from user stories into UML use case diagrams [27]. Their approach adopted the TreeTagger parser for part-of-speech (POS) tagging to process user stories that follow the template of “As a(n) [role], I want to/I can/I am able to [action], so that [benefit]”, so as to extract actors, use cases and their association relationships. The method was evaluated on a dataset of 90 user stories from the WebCompany case study, achieving precision and recall of 98% for actor extraction, and 87% precision and 85% recall for both use case and relationship extraction. The study shows that the structured form of user stories offers favorable syntactic basis for NLP-based extraction, while the main limitations lie in the fact that complex sentences, containment relations, and multi-complex nouns cannot be processed by the proposed plugin.

More recently, Zakaria et al. proposed a novel approach for automatically extracting UML class diagrams from textual software specifications by leveraging large language models (LLMs) [28]. They first reviewed two mainstream traditional methods: linguistic approaches based on rules and grammars, and statistical approaches relying on machine learning. To overcome the limitations of conventional methods, the authors designed an LLM-centered framework that includes data pre-processing, encoding, transformer-based processing, decoding, and class diagram generation. This approach uses pre-trained models such as BERT and GPT to identify classes, attributes, methods and inter-class relationships, and is currently in the stage of training dataset collection for subsequent model validation. Nguyen et al. proposed an approach to automatically generate UML class diagrams from natural language requirements using state-of-the-art large language models [29]. Their method adopts chain-of-thought prompting to guide the model to extract domain entities, attributes, and associations step by step, and evaluates the generated diagrams through a comprehensive dual-validation framework that combines LLM-as-a-Judge pairwise comparison and human-in-the-loop expert assessment. The study demonstrates that advanced large language models can produce structurally coherent and semantically meaningful class diagrams, while facing challenges in handling domain-specific complexity and subjective evaluation criteria such as comprehensibility in specialized scenarios.

Beyond UML generation, Camburn et al. explored machine learning-based automated evaluation for natural language design concepts in early-stage engineering design, proposing a framework that extracts ontological data via machine learning and quantifies design concepts with two core metrics: Span (for novelty) and Detail (for level of detail) [30]. Although their work does not focus on UML diagrams, it offers methodological references for automated assessment of design artifacts—especially the use of ontology-driven quanti-

tative metrics to measure creativity-related attributes—which guides the construction of our automated evaluation metric. Their experimental results show that the human inter-rater agreement for detail is high (0.73–0.91), and the machine–human correlation for the Detail metric reaches 0.69, which supports the validity of automated quantitative evaluation for design artifacts and provides a basis for our PlantUML-based evaluation scheme.

2.3. Prompt Engineering

Prompt engineering has emerged as a critical factor in eliciting high-quality outputs from LLMs, with a growing body of work systematically characterizing the effects of different prompting strategies across task types. Wei et al. introduced chain-of-thought (CoT) prompting, demonstrating that augmenting few-shot prompts with intermediate reasoning steps—rather than providing only input–output pairs—substantially improves LLM performance on arithmetic, commonsense, and symbolic reasoning tasks [31]. Their experiments across PaLM, GPT-3, and LaMDA showed that CoT gains scale with model size, emerging most strongly in models with over 100B parameters. This work established the theoretical foundation for our P4 prompting strategy, in which we decompose the class diagram generation task into explicit reasoning steps.

Brown et al. established the few-shot in-context learning paradigm through their seminal work on GPT-3, a 175B parameter autoregressive language model [32]. By demonstrating that providing a small number of input–output examples directly in the prompt context enables strong task performance without any gradient updates or fine-tuning, this work fundamentally changed how practitioners interact with large language models (LLMs). Their evaluation across 42 accuracy-denominated benchmarks showed that few-shot performance improves steadily with both model size and the number of in-context examples, revealing the effectiveness of in-context learning for prompting-based LLM adaptation. Liu et al. presented a systematic survey of prompt-based learning in natural language processing, proposing a unified formal framework and typology to organize prompting methods across dimensions including prompt template engineering, prompt answer engineering, multi-prompt learning, and prompt-aware training strategies [33]. They also elucidated the paradigm shift from traditional supervised learning to the “pre-train, prompt, and predict” paradigm, and summarized the applications, challenges, and correlations of prompt learning with other research fields. This survey provides a solid theoretical basis for our systematic comparison of prompting strategies.

White et al. proposed a domain-independent catalog of 16 prompt patterns for conversational large language models, which are classified into six functional categories: input semantics, output customization, error identification, prompt improvement, interaction, and context control [34]. Among these patterns, the Persona Pattern (role-based prompting) and Output Automater Pattern (generating executable scripts to automate recommended steps) are highly relevant to structured and domain-specific output tasks. They theoretically elaborated that the combination of multiple prompt patterns can enhance the performance of LLM outputs, which provides a theoretical basis for our study on combined prompting strategies. Kojima et al. demonstrated that zero-shot chain-of-thought reasoning can be elicited from LLMs simply by appending the phrase “Let’s think step by step” to a prompt, without any task-specific examples or manually crafted reasoning chains [35]. Evaluated across arithmetic, symbolic, and commonsense reasoning benchmarks, zero-shot CoT significantly outperformed standard zero-shot prompting while underperforming few-shot CoT, suggesting that the reasoning capability is latent in the model and can be activated through minimal prompting. This finding is particularly relevant to our work as it suggests that structured decomposition prompts—even without domain-specific examples—can substantially improve class diagram generation quality.

2.4. Iterative LLM Refinement

Iterative refinement has emerged as a powerful paradigm for improving LLM output quality beyond single-round generation, with several distinct approaches demonstrating consistent gains across diverse tasks. Madaan et al. proposed Self-Refine, a general framework for iteratively improving LLM outputs that operates without supervised training data or reinforcement learning [36]. In Self-Refine, a single LLM serves three roles: generator, feedback provider, and refiner. The model first generates an initial output, then critiques it by identifying specific weaknesses, and finally produces an improved version based on the critique. Evaluated across seven diverse tasks including code optimization, dialogue response generation, and mathematical reasoning, Self-Refine achieved an average of 20% absolute performance improvement over single-round generation. Critically, the authors found that the quality of self-generated feedback is the primary determinant of refinement effectiveness, which motivates our investigation of externally grounded feedback signals in IT2.

Shinn et al. introduced Reflexion, a framework that equips LLM-based agents with verbal reinforcement learning by converting task feedback signals into natural language reflections stored in an episodic memory buffer [37]. Compared with Self-Refine, the core difference of Reflexion is that it adds an episodic memory module to store reflective experiences, and can flexibly integrate both external task feedback (e.g., test case results, environment observations) and internal self-generated feedback (e.g., self-written unit tests, LLM self-evaluation) to achieve more targeted iterative revisions with grounded signals. Evaluated on AlfWorld sequential decision-making, HotpotQA question answering, and HumanEval coding tasks, Reflexion achieved state-of-the-art results, with a 91% pass@1 single-sample accuracy on HumanEval. The idea of Reflexion to realize iterative optimization through verbal feedback and episodic memory directly inspired our IT2 automated feedback refinement scheme.

Chen et al. proposed SELF-DEBUGGING, a framework that enables large language models (LLMs) to iteratively self-debug generated code via few-shot prompting [38]. Inspired by rubber duck debugging, the core of their approach is to ask the model to explain the generated code line-by-line in natural language for error localization; in scenarios with unit tests, it can further combine code execution results and test outputs as feedback, without human guidance or additional model training. Their work demonstrates that the combination of code explanation (static self-critique) and execution feedback yields the best performance, effectively identifying and correcting semantic errors (e.g., flawed SQL logic, code translation mismatches). Pure static code explanation also plays a critical role in tasks without test cases. On the Spider text-to-SQL benchmark (no unit tests), self-debugging improves single-round generation by 2–3% and by up to 9% on the hardest problems; on the TransCoder code translation benchmark (with unit tests), it achieves an improvement of up to 12%.

Olausson et al. conducted a rigorous empirical study on the self-repair capability of LLM-generated code, systematically evaluating the performance of self-repair supported by model self-generated feedback and feedback from stronger models/humans across multiple models and benchmarks [39]. Their key finding is that self-repair is not a universal solution to improve code generation performance, and the performance gains are often modest when computational costs are taken into account; the efficacy of self-repair is bottlenecked by the quality of feedback that the model itself provides on faulty code, and using feedback from stronger models or human experts can significantly improve repair success rates. Peng et al. pioneered the use of GPT-4 to generate instruction-following data for large language model instruction tuning, and also collected GPT-4-generated comparison and scoring feedback to evaluate model outputs and train reward models [40]. Their work

validated that instruction-following data produced by GPT-4 significantly enhances the zero-shot performance of instruction-tuned models, and demonstrated the feasibility of using GPT-4's automatic feedback for model evaluation and alignment research. This informed our use of structured quantitative scoring and targeted element assessment in the feedback design, drawing on the value of GPT-4's evaluative feedback for model alignment.

3. Methodology

3.1. Dataset Construction

3.1.1. Exercise Collection and Curation

We collected 30 UML class diagram exercises from software engineering textbooks, academic course materials, and publicly available modeling repositories. Each exercise consists of: (1) a natural language requirement description, (2) a ground-truth UML class diagram image, and (3) the corresponding ground-truth PlantUML code. Exercises were selected to cover a diverse range of application domains, including library management systems, e-commerce platforms, hospital information systems, and banking applications, among others.

3.1.2. Difficulty Stratification

To enable analysis of how requirement difficulty affects model performance, we stratified the 30 exercises into three difficulty groups—easy (10), medium (10), and hard (10)—using a composite difficulty score computed from four dimensions:

Class Count (C). The number of distinct classes in the ground-truth diagram. Higher class counts indicate greater structural complexity.

Relationship Complexity (R). The number of different relationships in the ground-truth diagram. Larger number of relationships denotes greater semantic complexity. Different relationship types contain dependency, association, aggregation, composition, and inheritance.

Attribute Count (A). The total number of attributes across all classes in the ground-truth diagram.

Flesch–Kincaid Score (FK). Applied to the natural language requirement text to quantify linguistic complexity. The Flesch–Kincaid (FK) readability metric ranges from 0 to 100 numerically. Higher FK values correspond to easier plain text, whereas lower scores stand for compact and complicated technical descriptions. Its standard calculation formula is defined as $FK = 206.835 - 1.015 \times ASL - 84.6 \times ASW$, where ASL means average sentence length and ASW denotes average syllables per single word. Its core formula was first introduced in the 1974 monograph *The Art of Readable Writing* [41], and the complete derivation and practical classification criteria were released in a 1975 U.S. technical report titled *Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease)* [42]. Following the original classic standard, we classify FK scores as follows: values greater than 60 indicate easy prose, scores between 30 and 60 represent moderately complex technical text, and values below 30 stand for highly condensed and difficult technical content. The FK-related calculations can be found at the following website: <https://goodcalculators.com/flesch-kincaid-calculator> (accessed on 1 August 2025).

Therefore, the composite difficulty score D for exercise i is computed as

$$D_i = w_c \cdot C_i + w_r \cdot R_i + w_a \cdot A_i + w_{fk} \cdot FK_i \quad (1)$$

where the weights $(w_c, w_r, w_a, w_{fk}) = (0.35, 0.4, 0.23, 0.02)$ were determined through expert consultation. The four-dimensional composite difficulty weights are finalized via consensus discussion among two experienced senior software engineering researchers who

possess over ten years of UML modeling teaching and engineering practice experience. The tiny weight assigned to FK originates from domain consensus that structural indicators including class quantity and inter-class relationship dominate modeling difficulty, while text readability exerts limited influence on technical requirement comprehension [17]. We retain original raw-data weighting without normalization, as primitive numerical features carry inherent domain semantics for UML difficulty evaluation. Exercises are then ranked by D_i and partitioned into three equal-sized groups.

It is worth noting that, even with the small coefficient of FK (0.02), we preserve it in composite difficulty rather than dropping the readability dimension entirely. The composite score is designed to integrate structural and textual complexity simultaneously, so FK supplements unique linguistic information unavailable from class, relation and attribute metrics. For standard UML modeling scenarios, structural features are the primary source of task difficulty, while linguistic characteristics act as auxiliary attributes. Retaining FK enables our scoring scheme to differentiate tasks with identical structural indicators but distinct expression styles, capturing fine-grained discrepancies and refining the accuracy of difficulty stratification.

Table 1 summarizes the data characteristics of each exercise, and Table 2 summarizes the statistical characteristics of each difficulty group (average level). The medium difficulty group obtains the highest average FK value among three tiers. This phenomenon stems from writing styles of requirement texts: easy requirements are usually described in verbose and plain language (high FK), hard requirements adopt extremely concise professional expressions (low FK), while medium requirements balance plain narration and technical wording naturally, thus forming a non-monotonic distribution of FK scores. According to mean values and standard deviations, most indicators show completely separated value ranges across three difficulty groups. Only the FK metric presents partial overlap between easy and medium groups, which is caused by natural differences in text expression and does not affect the rationality of our difficulty stratification. The detailed dataset can be viewed on the following GitHub link: <https://github.com/FlyingDwarfViolaMelon> (accessed on 1 May 2026).

3.2. Evaluation Metric

Given a model-generated PlantUML code snippet, we apply a rule-based parser to extract three structural sets: the set of classes $\mathcal{C}$, the set of relationships $\mathcal{R}$ (each represented as a triple $\langle \text{source}, \text{type}, \text{target} \rangle$), and the set of attributes $\mathcal{A}$ (each represented as a pair $\langle \text{class}, \text{attribute_name} \rangle$). The same parser is applied to the ground-truth PlantUML code to obtain reference sets $\mathcal{C}^*$, $\mathcal{R}^*$, and $\mathcal{A}^*$.

For each structural dimension, we compute a similarity matching score. For classes:

$$S_C = \frac{|\mathcal{C} \cap \mathcal{C}^*|}{|\mathcal{C}^*|} \quad (2)$$

Class matching is performed using normalized string similarity (lowercased, stemmed) with a threshold of 0.85 to account for minor naming variations. Relationship matching requires both endpoint classes and relationship type to match. Attribute matching requires both the owning class and attribute name to match. Analogous similarity matching scores S_R and S_A are computed for relationships and attributes respectively.

Therefore, the final score S for a generated diagram is

$$S = w_1 \cdot S_C + w_2 \cdot S_R + w_3 \cdot S_A \quad (3)$$

where $(w_1, w_2, w_3) = (0.4, 0.4, 0.2)$. The higher weight assigned to classes and relationships reflects their greater structural significance in class diagram modeling. Attribute completeness, while important, is considered secondary to structural correctness.

Table 1. The data characteristics of UML class diagram exercise.

Exercise ID	Class	Relationship	Attribute	Flesch–Kincaid	Difficulty
exercise 1	5	5	14	50.4	easy
exercise 2	6	6	28	38.5	hard
exercise 3	6	7	10	45.9	easy
exercise 4	6	5	12	55.2	easy
exercise 5	7	8	12	67.5	easy
exercise 6	6	7	31	39.9	hard
exercise 7	8	11	14	56.9	medium
exercise 8	9	11	12	66.4	medium
exercise 9	6	5	7	58.6	easy
exercise 10	9	9	15	66.8	medium
exercise 11	6	8	16	54.0	medium
exercise 12	4	3	9	47.2	easy
exercise 13	11	12	12	54.0	hard
exercise 14	8	10	13	42.4	medium
exercise 15	11	13	16	45.1	hard
exercise 16	10	16	17	28.2	hard
exercise 17	10	15	20	41.6	hard
exercise 18	6	6	17	68.8	easy
exercise 19	7	10	11	48.6	medium
exercise 20	8	7	2	62.2	easy
exercise 21	13	16	38	39.4	hard
exercise 22	8	11	26	38.3	hard
exercise 23	6	9	11	55.4	medium
exercise 24	7	7	12	53.8	easy
exercise 25	9	9	20	66.4	medium
exercise 26	5	5	12	59.6	easy
exercise 27	8	8	0	74.5	medium
exercise 28	11	14	13	59.3	hard
exercise 29	9	10	13	54.7	medium
exercise 30	10	10	37	60.8	hard

Table 2. The dataset statistics by difficulty group. Values outside parentheses represent average values; values inside parentheses denote standard deviation.

Difficulty	Avg. Class	Avg. Relationship	Avg. Attribute	Avg. FK
easy	6.0 (1.1)	5.8 (1.4)	10.7 (3.9)	56.92 (7.50)
medium	7.9 (1.1)	9.5 (1.0)	12.5 (4.9)	58.61 (9.22)
hard	9.6 (2.2)	12.0 (3.3)	23.8 (9.1)	44.51 (9.84)

3.3. Research Questions and Experimental Design

We organize our empirical study around four research questions, each targeting a distinct dimension of LLM-based UML class diagram modeling.

RQ1: How do model type and parameter scale affect UML class diagram modeling performance?

The LLM landscape encompasses models that differ substantially in architecture, training paradigm, reasoning capability, and parameter count. In particular, the emergence of reasoning-oriented models (e.g., DeepSeek-R1 distillation series) alongside general-purpose chat models raises the question of whether enhanced reasoning capability translates into

better UML structural modeling. Understanding which model characteristics correlate with class diagram modeling quality is essential for practitioners selecting models for this task.

Experimental design for RQ1: We evaluate the following six models using the standardized baseline prompt strategy **S0** (described in detail under RQ2):

- General-purpose chat models: DeepSeek-V3.2-Chat, Qwen1.5-32B-Chat, Qwen1.5-72B-Chat.
- Reasoning-distilled models: DeepSeek-R1-Distill-Qwen-14B, DeepSeek-R1-Distill-Qwen-32B.
- Large-scale thinking model: Qwen3-235B-A22B-Thinking.

Six representative LLMs are selected covering three mainstream categories: general-purpose chat models, reasoning-distilled variants and ultra-large thinking model. Other mainstream commercial LLMs like GPT-4 and Claude are excluded owing to permanent API access restrictions and our core research target on open-source reproducible model evaluation. Each model is evaluated on all 30 exercises. We report mean weighted scores S overall and per difficulty group, as well as the three sub-scores S_C , S_R , S_A separately, to characterize where different model types succeed or fail. We analyze performance differences across two dimensions: (1) parameter scale, by comparing models within the same family at different sizes (Qwen1.5-32B vs. Qwen1.5-72B; DeepSeek-R1-Distill-Qwen-14B vs. DeepSeek-R1-Distill-Qwen-32B); and (2) model type, by comparing general chat models against reasoning-distilled models at comparable parameter scales (Qwen1.5-32B-Chat vs. DeepSeek-R1-Distill-Qwen-32B), and both against the large-scale chat model (DeepSeek-V3.2-Chat) and the thinking model (Qwen3-235B-A22B-Thinking). Statistical significance of pairwise performance differences is assessed using the Wilcoxon signed-rank test for multiple comparisons. Considering the problem of multiple comparisons, we have applied the Bonferroni method to adjust all original p -values to control Type I error. The corrected significance level is set to ($\alpha' = 0.05/15 = 0.0033$).

RQ2: How do different prompting strategies influence class diagram modeling quality?

Prompt engineering is known to substantially affect LLM output quality, yet no prior work has systematically compared prompting strategies for UML class diagram generation across multiple models. We design four prompting strategies with progressively richer guidance (Appendix A), enabling us to isolate the contribution of each added component: domain knowledge anchoring, structured step decomposition, and self-verification.

Experimental design for RQ2: All four strategies share a common base template consisting of a role definition, a user requirement placeholder, a core guiding prompt (the unique differentiator across strategies), and fixed output requirements specifying PlantUML format. The four strategies are defined as follows:

- **S0**—Baseline (No Guidance): The core guiding prompt instructs the model to generate a UML class diagram from the provided requirements with no additional constraints. This strategy maximizes proximity to the model's native generation capability and serves as the reference baseline against which all gains from subsequent strategies are attributed.
- **S1**—UML Standard Guidance: Building on **S0**, **S1** adds explicit UML 2.5 standard anchoring. The model is instructed to accurately identify all classes and distinguish their attributes and methods, use only the five standard UML relationship types (association, inheritance, aggregation, composition, dependency), and mark inter-class multiplicities consistent with business logic. This strategy isolates the effect of domain knowledge embedding, testing whether anchoring the model to professional UML norms reduces low-level errors such as non-standard relationship expressions and missing multiplicities.

- **S2—Structured Step Guidance:** Building on **S1**, **S2** adds a four-step structured modeling process: (1) extract all classes, eliminate redundant concepts, and define attributes and methods; (2) analyze business relationships between each pair of classes, determine relationship types, and mark multiplicities; (3) verify compatibility of classes and relationships, checking for missing classes, incorrect relationships, and business logic consistency; (4) integrate all elements into a complete class diagram. This strategy tests whether explicit step-by-step decomposition improves inter-class reasoning and business logic consistency beyond domain knowledge anchoring alone.
- **S3—Self-Verification Guidance:** Building on **S2**, **S3** extends the four-step process with a fifth self-check and error correction step. After generating a preliminary diagram, the model is instructed to verify: (1) whether core classes, relationships, attributes, and methods are missing; (2) whether relationship types are incorrectly marked; (3) whether multiplicities contradict business logic; (4) whether the output conforms to PlantUML format specifications; (5) whether redundant classes or relationships are present. Only after correcting all identified errors is the final diagram output. This strategy tests whether post-generation self-verification further reduces low-level error rates and improves completeness.

All four strategies are applied to all six models across all 30 exercises. We report mean S , S_C , S_R , and S_A for each model-strategy combination, and compute the incremental gain of each strategy over **S0** (ΔS) to quantify the contribution of each added component. We further analyze whether prompting gains are consistent across model types and difficulty levels, and whether the relative ordering of strategies is preserved across models.

RQ3: How does requirement difficulty affect LLM class diagram modeling performance?

Real-world software requirements vary substantially in complexity. Understanding how difficulty modulates model performance—and which structural dimensions (classes, relationships, attributes) are most sensitive to increasing complexity—informs deployment decisions and highlights areas requiring targeted improvement.

Experimental design for RQ3: Design. Using the best-performing prompt strategy identified in RQ2 for each model, we conduct a fine-grained analysis of performance across the three difficulty groups (easy, medium, hard). For each model and each difficulty group, we report S , S_C , S_R , and S_A separately, enabling a multi-dimensional characterization of how difficulty impacts different structural aspects of class diagram generation. We compute the performance drop from easy to hard for each model and each sub-score dimension to quantify difficulty sensitivity. We further analyze the correlation between individual difficulty dimensions ($\mathcal{C}$: class count, $\mathcal{R}$: relationship complexity, $\mathcal{A}$: attribute count, FK : Flesch–Kincaid score) and model score degradation using Spearman rank correlation, identifying which difficulty factors are the strongest predictors of performance decline. Additionally, we compare difficulty sensitivity across model types—general chat models vs. reasoning-distilled models vs. the large-scale thinking model—to determine whether enhanced reasoning capability provides greater robustness to requirement complexity.

RQ4: How can iterative prompting strategies continuously improve LLM class diagram modeling capability?

Single-round generation represents a fundamental limitation of prior work. The prompting strategies **S1**, **S2**, and **S3** defined in RQ2 introduce progressively richer guidance elements; a natural question is whether these elements can be applied iteratively—building on an existing generated result rather than generating from scratch—to achieve further improvements. We design a structured iterative refinement experiment that tests whether sequential application of prompting guidance elements can continuously improve model outputs, and characterizes the contribution of each refinement step.

Experimental design for RQ4: Iterative refinement experiments are conducted on two representative models: DeepSeek-V3.2-Chat (best-performing general chat model) and Qwen3-235B-A22B-Thinking (large-scale thinking model), enabling comparison of iterative refinement effectiveness across model types. The iterative refinement procedure is built on a common mechanism: given an existing PlantUML code output, the model is prompted to check and revise it according to the guidance of the target strategy, with the instruction “Check and revise the generated PlantUML code as per the requirements.” appended to the strategy prompt. The initial PlantUML code fed into all iterative chains is the output generated by **S0** for each exercise, ensuring a consistent starting point across all refinement paths.

We evaluate five iterative refinement paths of increasing depth:

- **S0 → S1**: The **S0**-generated output is revised once using **S1** guidance (UML standard anchoring).
- **S0 → S2**: The **S0**-generated output is revised once using **S2** guidance (structured step decomposition).
- **S0 → S3**: The **S0**-generated output is revised once using **S3** guidance (self-verification).
- **S0 → S1 → S2**: The **S0**-generated output is first revised using **S1**, then the **S1**-revised output is further revised using **S2**.
- **S0 → S1 → S2 → S3**: The **S0 → S1 → S2** output is further revised using **S3**, forming the full three-step iterative chain.

This design allows us to answer several specific sub-questions: (1) Does iterative application of individual guidance elements (**S0 → S1**, **S0 → S2**, **S0 → S3**) outperform their single-round counterparts? (2) Does chaining multiple refinement steps (**S0 → S1 → S2 → S3**) yield cumulative gains beyond any single refinement step? (3) Do marginal gains diminish as the refinement chain lengthens? (4) Is the effectiveness of iterative refinement consistent across model types?

For each refinement path and each model, we report S , S_C , S_R , and S_A averaged across all 30 exercises, as well as per-difficulty-group breakdowns. Score trajectories along the refinement chain (**S0 → S0 + S1 → S0 + S1 + S2 → S0 + S1 + S2 + S3**) are plotted to visualize the cumulative effect of each refinement step. Statistical significance of improvements at each step is assessed using the Wilcoxon signed-rank test.

3.4. Experimental Setup

For model deployment, DeepSeek-V3.2-Chat and Qwen series are invoked via official open API service; DeepSeek-R1 distilled models are fully deployed locally on hardware equipped with A100 80 GB GPUs. No RAG retrieval enhancement or extra external tool is embedded during all prompt input procedures. All requirement texts are directly embedded into prompt template as plain text. Each group of experiments is repeated three times independently and final reported scores adopt average value across multiple runs.

4. Experimental Results

4.1. RQ1: Impact of Model Type and Parameter Scale

Tables 3 and 4 present the mean weighted scores of all six evaluated models on the full dataset under the **S0** baseline prompt strategy, broken down by overall performance and per difficulty group. $\Delta_{E \rightarrow H}$ indicates the performance drop from easy to hard.

Based on the above results, we can make the following discoveries in RQ1.

Finding 1: DeepSeek-V3.2-Chat achieves the highest overall score (Avg. $S = 44.01\%$) among all evaluated models, substantially outperforming the second-ranked model, Qwen3-235B-A22B-Thinking (Avg. $S = 36.03\%$), by a margin of 7.98 percentage points. The overall scores across all models are notably low, with the best-performing model reach-

ing only 44.01% and the worst-performing model (Qwen1.5-32B-Chat) reaching 26.54%. This indicates that UML class diagram generation from natural language requirements under zero-guidance conditions remains a genuinely difficult task for all evaluated models, with substantial room for improvement through prompting strategies and iterative refinement. In addition, The highest overall weighted score 44.01% obtained by DeepSeek-V3.2-Chat indicates moderate overall modeling capacity, which means this model correctly identifies nearly 44% of total valid UML structural elements including classes, relationships and attributes. Such limited score intuitively proves generating standard UML class diagram from natural language remains a tough challenge for existing mainstream large language models.

Finding 2: Across all models, S_C (class matching score) is consistently the highest sub-score, ranging from 56.41% (Qwen1.5-32B-Chat) to 68.41% (DeepSeek-V3.2-Chat). In stark contrast, S_R (relationship matching score) is dramatically lower, ranging from 8.12% (Qwen1.5-32B-Chat) to 39.84% (DeepSeek-V3.2-Chat), and S_A (attribute matching score) is the lowest across all models, with all six models scoring below 4.5%. This pronounced sub-score disparity reveals a clear capability hierarchy: identifying classes is the most tractable sub-task, while relationship modeling and attribute extraction represent severe bottlenecks under baseline conditions. The near-zero S_A scores suggest that attribute-level generation is largely unreliable without explicit prompting guidance.

Finding 3: Parameter scale does not monotonically determine performance across model families. Within the Qwen1.5 general chat family, the 72B model (Avg. $S = 29.93\%$) outperforms the 32B model (Avg. $S = 26.54\%$) by 3.39 percentage points, consistent with the expectation that larger scale yields better performance within the same training paradigm. However, within the DeepSeek-R1-Distill-Qwen reasoning family, the 32B model (Avg. $S = 35.20\%$) outperforms the 14B model (Avg. $S = 30.81\%$) by 4.39 percentage points, also consistent with scale expectations. Crucially, DeepSeek-V3.2-Chat (671B) substantially outperforms all other models, suggesting that at sufficiently large scale, general chat models can dominate both reasoning-distilled and thinking-oriented models on this task.

Finding 4: At comparable parameter scales (32B), the reasoning-distilled model (DeepSeek-R1-Distill-Qwen-32B, Avg. $S = 35.20\%$) substantially outperforms the general chat model (Qwen1.5-32B-Chat, Avg. $S = 26.54\%$) by 8.66 percentage points. This gap is particularly pronounced in S_R (25.73% vs. 8.12%, a difference of 17.61 percentage points), indicating that reasoning-oriented training provides a decisive advantage specifically in relationship modeling—the most challenging sub-task. This finding suggests that the structured multi-step inference characteristic of reasoning-distilled models is particularly well-suited to the relationship identification demands of UML class diagram generation.

Finding 5: The difficulty-group analysis in Table 3 reveals several noteworthy patterns. First, DeepSeek-V3.2-Chat demonstrates the most stable performance across difficulty levels, with scores of 46.62%, 43.39%, and 42.02% for easy, medium, and hard exercises respectively ($\Delta_{E \rightarrow H} = -4.60$), indicating strong robustness to requirement complexity. Second, DeepSeek-R1-Distill-Qwen-32B shows the largest performance drop from easy to hard ($\Delta_{E \rightarrow H} = -10.28$), despite its strong overall ranking, suggesting that its reasoning capability is more sensitive to structural complexity than its overall score implies. Third, and most strikingly, Qwen3-235B-A22B-Thinking is the only model that shows a positive $\Delta_{E \rightarrow H} = +3.82$, meaning its performance actually improves from easy to medium exercises (31.66% $\rightarrow$ 40.94%) before declining slightly on hard exercises (35.48%). This anomalous pattern suggests that the thinking model may underperform on simple requirements—possibly due to over-elaboration or excessive reasoning steps—while performing relatively better when requirements provide richer structural context to reason

about. Moreover, The non-monotonic changing trend across three difficulty tiers is reasonable. Easy-level requirements are usually described in ambiguous, over-simplified expressions lacking detailed structural constraints; hard tasks contain excessive overlapping structures beyond model processing capacity; medium requirements balance textual description and structural complexity, fitting the inherent reasoning boundary of current LLMs, hence achieving the optimal average performance.

Finding 6: Wilcoxon signed-rank tests with Bonferroni correction ($\alpha' = 0.0033$) confirm that DeepSeek-V3.2-Chat significantly outperforms Qwen1.5-32B-Chat ($p = 0.0003$) and DeepSeek-R1-Distill-Qwen-14B ($p = 0.0018$). Differences between DeepSeek-V3.2-Chat and the remaining models, while directionally consistent, do not reach statistical significance after correction, reflecting the limited sample size of 30 exercises. Notably, the difference between Qwen3-235B-A22B-Thinking and DeepSeek-R1-Distill-Qwen-14B is non-significant ($p = 0.586$), consistent with their near-identical overall scores (36.03% vs. 35.20%).

Table 3. Overall performance of all models under **S0** strategy (%).

Model	Type	Params	Avg. S	Avg. S_C	Avg. S_R	Avg. S_A
Qwen3-235B-A22B-Thinking	Thinking	235B	36.03	63.06	25.24	3.54
DeepSeek-V3.2-Chat	General	671B	44.01	68.41	39.84	3.54
Qwen1.5-32B-Chat	General	32B	26.54	56.41	8.12	3.66
Qwen1.5-72B-Chat	General	72B	29.93	61.49	11.68	3.33
DeepSeek-R1-Distill-Qwen-14B	Reasoning	14B	30.81	56.98	17.85	4.38
DeepSeek-R1-Distill-Qwen-32B	Reasoning	32B	35.20	60.53	25.73	3.46

Table 4. Average S score performance by difficulty group under **S0** strategy (%).

Model	Easy	Medium	Hard	$\Delta_{E \rightarrow H}$
Qwen3-235B-A22B-Thinking	31.66	40.94	35.48	3.82
DeepSeek-V3.2-Chat	46.62	43.39	42.02	−4.60
Qwen1.5-32B-Chat	28.48	25.45	25.70	−2.78
Qwen1.5-72B-Chat	31.75	33.68	24.36	−7.39
DeepSeek-R1-Distill-Qwen-14B	31.02	34.73	26.68	−4.34
DeepSeek-R1-Distill-Qwen-32B	37.51	40.85	27.23	−10.28

4.2. RQ2: Impact of Prompting Strategies

Table 5 reports the performance of all six models under each of the four prompting strategies. Table 6 presents the incremental gains between strategies. Table 7 summarizes sub-score improvements averaged across all models.

Table 5. Average S score performance under different prompting strategies (%).

Model	S0	S1	S2	S3
Qwen3-235B-A22B-Thinking	36.03	39.32	37.57	39.43
DeepSeek-V3.2-Chat	44.01	43.04	42.70	45.19
Qwen1.5-32B-Chat	26.54	27.10	29.74	26.58
Qwen1.5-72B-Chat	29.93	31.25	30.60	31.89
DeepSeek-R1-Distill-Qwen-14B	30.81	32.89	31.78	33.03
DeepSeek-R1-Distill-Qwen-32B	35.20	35.40	34.61	33.46

Table 6. Incremental gains ΔS (%).

Model	S1-S0	S2-S0	S3-S0	S2-S1	S3-S1	S3-S2
Qwen3-235B-A22B-Thinking	+3.29	+1.54	+3.40	−1.75	+0.11	+1.86
DeepSeek-V3.2-Chat	−0.97	−1.31	+1.18	−0.34	+2.15	+2.49
Qwen1.5-32B-Chat	+0.56	+3.20	+0.04	+2.64	−0.52	−3.16
Qwen1.5-72B-Chat	+1.32	+0.67	+1.96	−0.65	+0.64	+1.29
DeepSeek-R1-Distill-Qwen-14B	+2.08	+0.97	+2.22	−1.11	+0.14	+1.25
DeepSeek-R1-Distill-Qwen-32B	+0.20	−0.59	−1.74	−0.79	−1.94	−1.15

Table 7. Sub-score improvements from S0 to S3 (averaged across all models).

Sub-Score	S0	S1	S2	S3	Total Gain
S_C	61.15	62.46	62.12	62.40	1.25
S_R	21.41	22.79	22.20	22.99	1.58
S_A	3.65	3.67	3.85	3.88	0.23
S	33.75	34.83	34.50	34.93	1.18

Based on the above results, we can make the following discoveries in RQ2.

Finding 7: The overall gains from prompting strategies are modest across all models, with the average weighted score improving from 33.75% (S0) to only 34.93% (S3), a total gain of merely 1.18 percentage points averaged across all models. This limited improvement stands in sharp contrast to the expectations from the prompt engineering literature, and reveals that simply adding UML domain knowledge, structured steps, or self-verification instructions to a single-round prompt does not reliably unlock substantial performance gains for this task. The result motivates the iterative refinement approach investigated in RQ4.

Finding 8: No single prompting strategy consistently dominates across all models. The best-performing strategy varies by model: S3 achieves the highest score for Qwen3-235B-A22B-Thinking (39.43%), DeepSeek-V3.2-Chat (45.19%), Qwen1.5-72B-Chat (31.89%), and DeepSeek-R1-Distill-Qwen-14B (33.03%); S2 achieves the highest score for Qwen1.5-32B-Chat (29.74%); while for DeepSeek-R1-Distill-Qwen-32B, S0 (35.20%) actually outperforms all other strategies, with performance monotonically decreasing from S1 through S3. This heterogeneity indicates that the effect of prompting guidance is model-dependent, and that more structured prompts do not universally benefit all model types.

Finding 9: The incremental gain patterns in Table 6 reveal that the relationship between prompting complexity and performance improvement is non-linear and model-specific. For most models, S1 (UML standard anchoring) provides a positive first-step gain, with Qwen3-235B-A22B-Thinking (+3.29), DeepSeek-R1-Distill-Qwen-14B (+2.08), and Qwen1.5-72B-Chat (+1.32) all benefiting from domain knowledge embedding. However, the transition from S1 to S2 yields negative incremental gains for four out of six models (S2-S1 ranging from −0.34 to −1.75), suggesting that adding structured step decomposition on top of UML standard guidance introduces conflicting constraints for most models rather than providing complementary benefits. S3 partially recovers these losses for several models, with the S3-S2 gain being positive for four models, indicating that the self-verification mechanism in S3 is more compatible with model behavior than the rigid step decomposition in S2.

Finding 10: The sub-score analysis in Table 6 reveals that all three structural dimensions show only marginal improvements from S0 to S3 when averaged across models. S_R shows

the largest absolute gain (+1.58 percentage points), followed by S_C (+1.25) and S_A (+0.23). The near-zero gain in S_A (from 3.65% to 3.88%) confirms that attribute-level generation remains largely unresponsive to single-round prompting guidance across all strategies, reinforcing the conclusion that attribute extraction requires more targeted interventions such as iterative refinement. Notably, S_R and S_C both show a non-monotonic trajectory across strategies, mirroring the non-monotonic pattern observed at the model level and suggesting that the interaction between prompting strategy and model behavior is complex and not simply additive.

4.3. RQ3: Impact of Requirement Difficulty

Table 8 presents the performance breakdown by difficulty group for all models using their best-performing prompt strategy (S3), reporting all four score dimensions. Figures 1–3 are the overall radar charts of the models under different difficulty levels.

Table 8. Performance by difficulty group under S3 strategy (Avg.).

Dataset	Metrics	Easy	Medium	Hard	$\Delta_{E \rightarrow H}$
Qwen3-235B-A22B-Thinking	Avg. S	31.42	47.07	39.85	+8.43
	Avg. S_C	65.11	70.66	61.20	−3.91
	Avg. S_R	13.43	42.02	38.22	+24.79
	Avg. S_A	0	10	0.43	+0.43
DeepSeek-V3.2-Chat	Avg. S	45.85	47.63	42.06	−3.79
	Avg. S_C	73.77	69.68	63.98	−9.79
	Avg. S_R	40.86	44.09	41.18	0.32
	Avg. S_A	0	10.62	0	0
Qwen1.5-32B-Chat	Avg. S	27.83	27.76	24.16	−3.67
	Avg. S_C	58.17	59.50	50.57	−7.60
	Avg. S_R	11.11	4.91	9.64	−1.47
	Avg. S_A	0.59	10	0.39	−0.20
Qwen1.5-72B-Chat	Avg. S	31.86	35.25	28.58	−3.28
	Avg. S_C	62.76	62.46	56.64	−6.12
	Avg. S_R	16.89	20.66	14.81	−2.08
	Avg. S_A	0	10	0	0
DeepSeek-R1-Distill-Qwen-14B	Avg. S	37.98	36.83	24.28	−13.70
	Avg. S_C	68.68	64.82	55.02	−13.66
	Avg. S_R	25.97	19.82	5.48	−20.49
	Avg. S_A	0.59	14.89	0.39	−0.20
DeepSeek-R1-Distill-Qwen-32B	Avg. S	35.82	36.42	28.14	−7.68
	Avg. S_C	65.35	63.53	50.89	−14.46
	Avg. S_R	23.90	22.20	19.26	−4.64
	Avg. S_A	0.59	10.62	0.39	−0.20

Based on the above results, we can make the following discoveries in RQ3.

Finding 11: The impact of requirement difficulty on model performance is highly heterogeneous across models, and does not follow a simple monotonic degradation pattern as commonly assumed. Five out of six models show a non-monotonic trajectory across difficulty levels, with medium exercises generally yielding higher scores than both easy and hard exercises. Specifically, DeepSeek-V3.2-Chat scores 45.85% (easy), 47.63% (medium), and 42.06% (hard); Qwen1.5-72B-Chat scores 31.86%, 35.25%, and 28.58%; and DeepSeek-R1-Distill-Qwen-32B scores 35.82%, 36.42%, and 28.14% respectively. This pattern suggests that easy exercises may present deceptively difficult challenges for LLMs—possibly due to under-specified or ambiguous requirements that lack sufficient structural context—while medium exercises provide a level of detail that better aligns with model capabilities. Hard

exercises, by contrast, introduce complexity that exceeds model capacity, resulting in the lowest scores across most models.

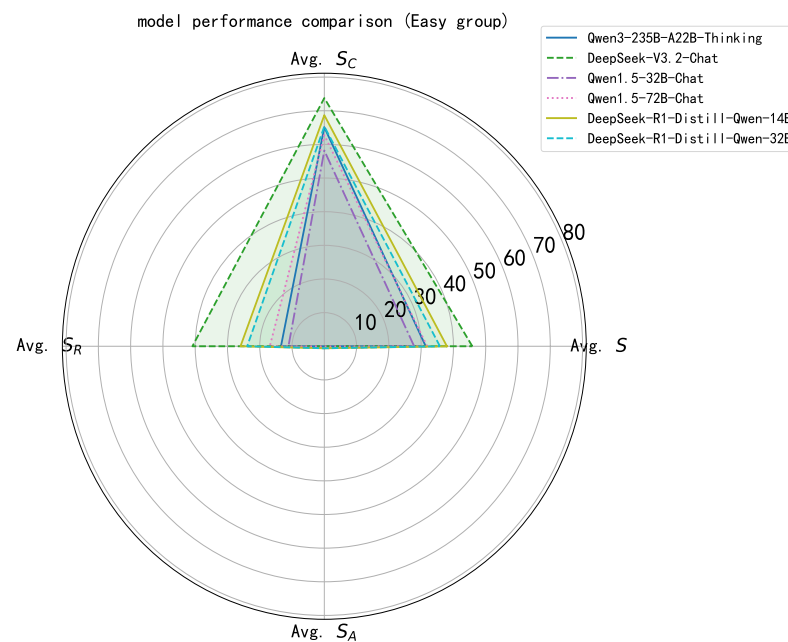


Figure 1. The overall model performance radar chart of the easy group under the S3 strategy.

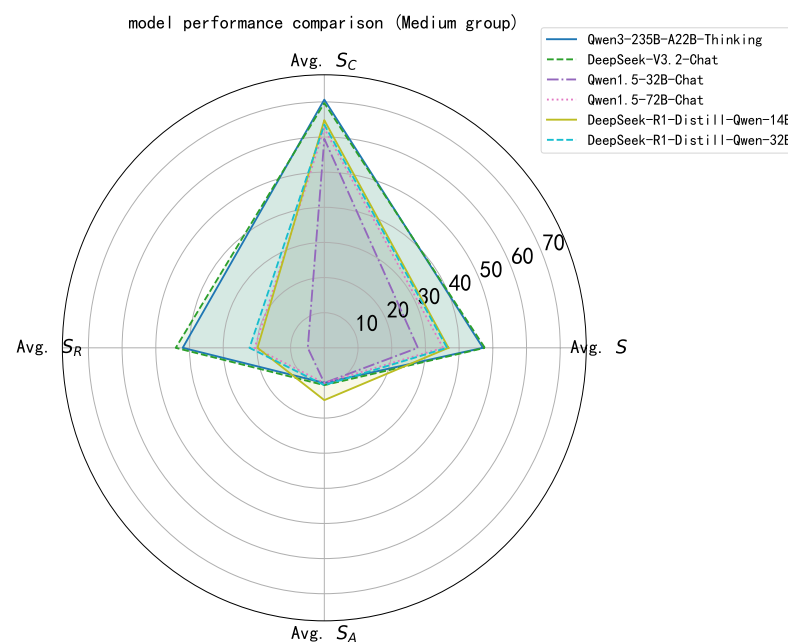


Figure 2. The overall model performance radar chart of the medium group under the S3 strategy.

Finding 12: S_A (attribute matching score) exhibits a striking and consistent pattern across all six models: scores are near zero for easy and hard exercises, but rise to approximately 10% for medium exercises. This pattern holds regardless of model type—DeepSeek-V3.2-Chat scores 0, 10.62, 0; Qwen1.5-72B-Chat scores 0, 10.00, 0; Qwen3-235B-A22B-Thinking scores 0, 10.00, 0.43—suggesting that attribute generation is not primarily driven by model capability differences but rather by characteristics specific to the medium-difficulty exercises in our dataset. A likely explanation is that medium-difficulty exercises contain more explicitly stated attributes in their requirement texts, making attribute extraction more tractable regardless of model type, while easy exercises

may describe systems at too high a level of abstraction and hard exercises may contain attributes distributed across complex, interleaved descriptions that resist extraction.

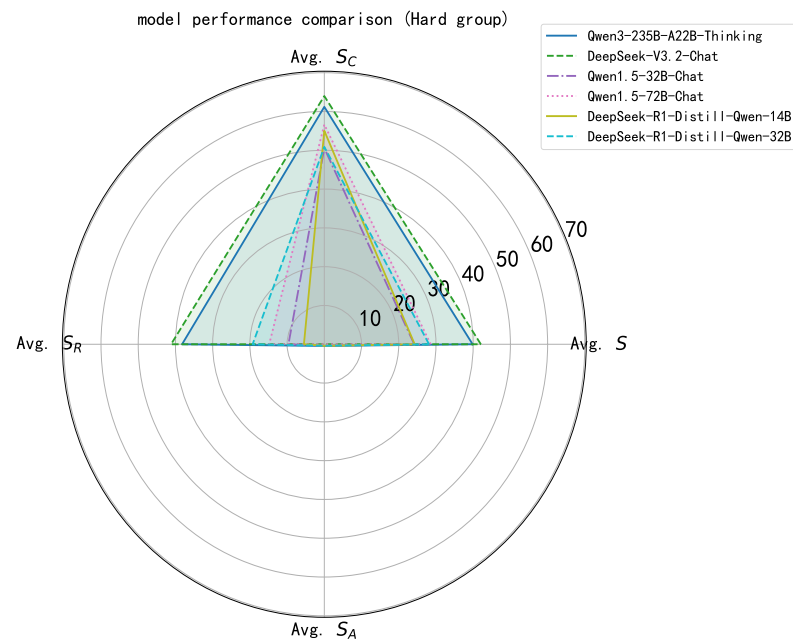


Figure 3. The overall model performance radar chart of the hard group under the S3 strategy.

Finding 13: The consistently near-zero S_A scores across easy and hard difficulty groups—regardless of model type, parameter scale, or prompting strategy—point to a fundamental limitation of current LLMs in attribute-level UML modeling (from Figure 1 to Figure 3). Even the strongest model, DeepSeek-V3.2-Chat, achieves 0% attribute matching on both easy and hard exercises under S3, despite achieving competitive scores on S_C and S_R . This suggests that attribute extraction requires a qualitatively different type of fine-grained reading comprehension that current prompting strategies, including self-verification (S3), are unable to reliably elicit. Targeted interventions specifically designed for attribute extraction—such as attribute-focused prompting or iterative attribute refinement—are likely necessary to meaningfully improve S_A scores.

Finding 14: Spearman rank correlation analysis between individual difficulty dimensions and model score degradation reveals that relationship complexity ($\mathcal{R}$) is the strongest predictor of performance decline ($\rho = -0.262$), followed by class count ($\mathcal{C}$, $\rho = -0.231$), attribute count ($\mathcal{A}$, $\rho = -0.056$), and Flesch–Kincaid readability score (FK , $\rho = -0.102$). The dominance of structural complexity factors over linguistic complexity confirms that the primary challenge in hard exercises is structural rather than linguistic in nature, and that improvements to LLM-based UML class diagram generation should prioritize structural reasoning capabilities—particularly relationship identification and typing—over natural language understanding. All calculated Spearman correlation coefficients have absolute values below 0.3, which are defined as weak correlation in statistical norms. Among four difficulty dimensions, relationship complexity owns the maximum negative correlation coefficient $\rho = -0.262$, verifying structural complexity dominates performance attenuation instead of linguistic FK index. As exploratory analysis with limited samples, adjusted p-values and confidence intervals are not supplemented in current manuscript.

4.4. RQ4: Impact of Iterative Prompting Strategies

Tables 9 and 10 report the score trajectories along all five iterative refinement paths for DeepSeek-V3.2-Chat and Qwen3-235B-A22B-Thinking respectively. Table 11 presents per-difficulty-group breakdowns for the full iterative chain.

Table 9. Iterative refinement score trajectories—DeepSeek-V3.2-Chat (%).

Path	Avg. S	Avg. S_C	Avg. S_R	Avg. S_A	Gain vs. S_0
S_0 (baseline)	43.45	68.34	38.52	3.54	–
$S_0 \rightarrow S_1$	43.66	68.34	39.05	3.54	+0.21
$S_0 \rightarrow S_2$	42.44	68.34	35.99	3.54	–1.01
$S_0 \rightarrow S_3$	42.77	68.34	36.82	3.54	–0.68
$S_0 \rightarrow S_1 \rightarrow S_2$	44.22	68.89	39.89	0.00	+0.77
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$	44.00	68.89	39.33	3.54	+0.55

Table 10. Iterative refinement score trajectories—Qwen3-235B-A22B-Thinking (%).

Path	Avg. S	Avg. S_C	Avg. S_R	Avg. S_A	Gain vs. S_0
S_0 (baseline)	34.74	62.83	22.26	3.54	–
$S_0 \rightarrow S_1$	38.36	65.52	28.61	3.54	+3.62
$S_0 \rightarrow S_2$	38.01	64.68	28.58	3.54	+3.27
$S_0 \rightarrow S_3$	36.61	62.91	26.84	3.54	+1.87
$S_0 \rightarrow S_1 \rightarrow S_2$	38.34	64.36	29.72	3.54	+3.60
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$	38.98	63.75	31.93	3.54	+4.24

Table 11. Full iterative chain ($S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$) by difficulty group.

Model	Metric	Easy	Medium	Hard	Overall
DeepSeek-V3.2-Chat	Avg. S	43.22	44.98	43.80	44.00
	Gain vs. S_0	+1.23	+0.10	+0.30	+0.55
Qwen3-235B-A22B-Thinking	Avg. S	32.90	46.82	37.23	38.98
	Gain vs. S_0	+3.61	+4.22	+4.89	+4.24

Based on the above results, we can make the following discoveries in RQ4.

Finding 15: The two models exhibit fundamentally different responses to iterative refinement, revealing that the effectiveness of iterative prompting is strongly model-dependent. Qwen3-235B-A22B-Thinking benefits consistently and substantially from iterative refinement, with all five refinement paths yielding positive gains over S_0 (ranging from +1.87 to +4.24 percentage points), and the full chain $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$ achieving the highest overall score of 38.98% (+4.24 over baseline). In contrast, DeepSeek-V3.2-Chat shows a mixed and largely marginal response: three out of five refinement paths yield negative or near-zero gains ($S_0 \rightarrow S_2$: –1.01, $S_0 \rightarrow S_3$: –0.68, $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$: +0.55), and the best iterative path ($S_0 \rightarrow S_1 \rightarrow S_2$: +0.77) provides only a modest improvement over baseline. This divergence suggests that iterative refinement is most beneficial for models whose single-round generation is constrained by insufficient guidance activation, while models that already perform near their ceiling under S_0 conditions derive limited additional benefit from iterative instruction layering.

Finding 16: The sub-score trajectories reveal an important structural pattern: S_C and S_A are remarkably stable across all iterative paths for both models, while S_R is the primary driver of performance variation. For DeepSeek-V3.2-Chat, S_C remains fixed at 68.34% across all single-step paths and increases marginally to 68.89% for multi-step paths, while S_A is uniformly 3.54% except in $S_0 \rightarrow S_1 \rightarrow S_2$ where it drops to 0%. The S_R variation (35.99% to 39.89%) accounts for virtually all of the observed performance differences across iterative paths. For Qwen3-235B-A22B-Thinking, S_R similarly drives the gains, rising from 22.26%

(S0) to 31.93% (S0 → S1 → S2 → S3), a gain of +9.67 percentage points, while S_C shows only modest variation (62.83% to 65.52%) and S_A remains constant at 3.54% across all paths. This confirms that iterative refinement primarily operates through improving relationship modeling, and has negligible effect on class identification and attribute extraction.

Finding 17: For Qwen3-235B-A22B-Thinking, the single-step iterative paths S0 → S1 (+3.62) and S0 → S2 (+3.27) both yield substantially larger gains than their single-round counterparts in RQ2 (S1-S0: +3.29, S2-S0: +1.54). This asymmetry is particularly pronounced for S2, where the iterative gain (+3.27) more than doubles the single-round gain (+1.54). This finding reveals that for this thinking model, applying structured step decomposition as a revision operation on an existing output is substantially more effective than using it to guide generation from scratch, likely because the existing S0 output provides a concrete structural scaffold that the model can systematically refine rather than construct anew. This suggests a fundamental difference in how thinking models process revision instructions versus generation instructions.

Finding 18: The per-difficulty analysis in Table 11 reveals contrasting patterns between the two models. For DeepSeek-V3.2-Chat, iterative gains are concentrated almost entirely on easy exercises (+1.23), with negligible gains on medium (+0.10) and hard (+0.30) exercises, suggesting that iterative refinement helps this model primarily by correcting minor errors in simple requirements where it already performs well, while providing little additional benefit for more complex requirements. For Qwen3-235B-A22B-Thinking, the pattern is reversed: gains increase monotonically with difficulty, from +3.61 (easy) to +4.22 (medium) to +4.89 (hard). This monotonically increasing gain pattern indicates that iterative refinement is most valuable for this model precisely on the exercises where single-round generation is most inadequate, and that the thinking model's reasoning capability is better leveraged through iterative revision than through single-round generation, particularly under conditions of high structural complexity.

Finding 19: Comparing the overall iterative refinement gains against the single-round prompting gains from RQ2, iterative refinement provides substantially larger improvements for Qwen3-235B-A22B-Thinking: the best iterative gain (+4.24 from S0 → S1 → S2 → S3) exceeds the best single-round gain (+3.40 from S3 in RQ2) by 0.84 percentage points. For DeepSeek-V3.2-Chat, the comparison favors single-round strategies: the best single-round gain (+1.18 from S3 in RQ2) exceeds the best iterative gain (+0.77 from S0 → S1 → S2). This cross-model comparison leads to a practically important conclusion: for thinking-oriented models, iterative refinement is the preferred improvement strategy; for high-performing general chat models, carefully selected single-round prompting strategies may be more effective and cost-efficient than iterative revision chains.

5. Discussion

5.1. Synthesis of Findings

Our experiments reveal key findings that collectively characterize the current state of LLM-based UML class diagram modeling. Specifically, our results expose a set of non-trivial interactions among model type, prompting strategy, requirement difficulty, and iterative refinement that carry important implications for both research and practice.

Overall performance levels remains low across all conditions, indicating fundamental task difficulty. The best-performing model (DeepSeek-V3.2-Chat) achieves only 44.01% under the baseline condition, and the best single-round prompting strategy raises this to 45.19%. These scores confirm that generating structurally accurate UML class diagrams from natural language requirements is a genuinely difficult task for current LLMs, particularly at the relationship and attribute levels, with substantial room for improvement remaining across all evaluated approaches. The persistently near-zero S_A scores across all

models, strategies, and difficulty levels underscore that attribute-level generation remains an essentially unsolved problem under current prompting paradigms.

Model type determines both performance level and guidance responsiveness. The finding that DeepSeek-V3.2-Chat (general chat, 671B) substantially outperforms Qwen3-235B-A22B-Thinking (thinking, 235B) under baseline conditions, while the thinking model responds more strongly to both prompting guidance and iterative refinement, reveals a fundamental trade-off between native generation capability and guidance responsiveness. High-performing general chat models appear to operate near their single-round ceiling, leaving little room for prompting or iterative interventions to add value. Thinking-oriented models, by contrast, appear to have latent structural reasoning capabilities that are not fully activated under minimal prompting but can be progressively unlocked through structured guidance. This suggests that the evaluation of LLMs for UML modeling tasks should not be based solely on single-round baseline performance, as this metric systematically underestimates the potential of thinking-oriented models.

Prompting strategy effectiveness is model-dependent, not universal. The finding that no single prompting strategy dominates across all models—and that DeepSeek-R1-Distill-Qwen-32B actually performs best under the zero-guidance baseline S_0 —challenges the common assumption that more structured prompts are universally beneficial. The counterproductive effect of structured step decomposition and self-verification on certain models (particularly reasoning-distilled models) suggests that these models have already internalized the reasoning structures that external prompts attempt to impose, making additional guidance redundant or disruptive. This has a practical implication: prompt engineering for UML modeling cannot be treated as a model-agnostic intervention, and practitioners must calibrate prompting complexity to model type.

Relationship modeling is the universal and persistent bottleneck. Across all experiments—model comparisons, prompting strategies, difficulty levels, and iterative refinement— S_R consistently lags behind S_C and is the primary dimension along which models fail under increasing complexity. The near-collapse of S_R for smaller reasoning-distilled models on hard exercises (DeepSeek-R1-Distill-Qwen-14B: $S_R = 5.48\%$ on hard exercises) and the dominance of S_R variation in driving iterative refinement gains confirm that relationship identification and typing is the core unsolved challenge in LLM-based UML class diagram generation. Future work should prioritize targeted interventions for relationship modeling, such as relationship-focused prompting templates or specialized fine-tuning on relationship-rich datasets.

Attribute extraction is an unsolved problem. S_A scores are near zero across all models, strategies, and difficulty levels. No evaluated approach reliably elicits attribute-level generation, indicating that attribute extraction requires fundamentally different mechanisms beyond current prompting paradigms.

Iterative refinement unlocks gains beyond single-round prompting, but only for certain models. The finding that iterative refinement provides substantially larger benefits for Qwen3-235B-A22B-Thinking than for DeepSeek-V3.2-Chat, and that the best iterative chain for the thinking model (+4.24 percentage points) exceeds the best single-round strategy for the same model (+3.40 percentage points), confirms that iterative revision is a qualitatively different optimization mechanism from single-round prompting. However, the limited and sometimes negative iterative gains for DeepSeek-V3.2-Chat caution against treating iterative refinement as a universally applicable improvement strategy. The cost of multiple inference rounds must be weighed against the expected benefit, which our results show to be model-specific.

5.2. Practical Guidelines

Based on our findings, we offer the following guidelines for researchers and practitioners:

- (1) Match prompting strategy to model type. Apply **S3** prompting strategy for general chat and thinking models; use **S0** or **S1** strategy for reasoning-distilled models, as structured guidance hurts rather than helps these models.
- (2) Use iterative refinement for thinking-oriented models on complex requirements. The full **S0** → **S1** → **S2** → **S3** chain provides the largest gains for Qwen3-235B-A22B-Thinking, with gains increasing monotonically with difficulty. For high-performing general chat models, single-round **S3** prompting is more cost-efficient than iterative revision.
- (3) Treat **S1** prompting strategy as the minimum prompting baseline. UML standard anchoring (**S1**) provides consistent positive gains across all models with no observed negative effects, and should replace zero-guidance prompting as the default starting point.
- (4) Do not rely on automated generation for attribute accuracy. Given near-zero S_A scores across all conditions, applications requiring attribute-level precision must incorporate post-generation human verification or domain-specific extraction pipelines.
- (5) Evaluate models on medium-difficulty requirements. The non-monotonic difficulty response indicates that easy exercises underestimate model capability. Medium-difficulty requirements better reflect realistic deployment conditions and should be prioritized in benchmark evaluation.

5.3. Threats to Validity

Internal validity. Our automated metric measures structural similarity rather than semantic correctness. Errors in relationship direction, multiplicity, or method signatures may not affect matching scores, potentially overestimating performance on specific exercises. In addition, our proposed weighted structural similarity metric mainly focuses on element matching rate and is naturally inclined to recall indicator without extra penalty items for redundant or spurious generated content. We fully acknowledge this inherent limitation and suggest future research construct multi-dimensional evaluation metrics balancing precision and recall simultaneously.

Construct validity. The difficulty stratification scheme relies on four proxy metrics (C, R, A, FK) that capture structural and linguistic complexity but may not fully reflect the cognitive difficulty of requirement interpretation. In particular, the Flesch–Kincaid score measures surface-level readability and may not capture domain-specific conceptual complexity that affects model performance independently of structural factors. Moreover, prompt content requires LLMs to generate methods, multiplicity parameters and directional relationships, whereas our automatic evaluation concentrates on core class, relationship and attribute items without fine-grained check for methods, multiplicity and relation direction. This mismatch is an acknowledged limitation of current evaluation scheme, and detailed fine-grained assessment will be explored in subsequent research.

External validity. Our dataset of 30 exercises, while carefully difficulty-stratified and domain-diverse, is limited in size. The non-monotonic difficulty effects and model-specific prompting responses observed in our experiments may not generalize to larger or more diverse requirement collections. Replication on larger benchmarks is necessary to confirm the robustness of our findings.

Model version validity. All experiments were conducted on specific model versions available at the time of the study. Given the rapid pace of LLM development, the absolute performance figures and relative model rankings reported here may not hold for future model releases. However, the qualitative findings regarding model-type differences,

prompting sensitivity, and iterative refinement dynamics are likely to remain relevant as general principles.

6. Conclusions and Future Work

6.1. Conclusions

This paper presented a preliminary empirical evaluation of large language model capabilities for UML class diagram modeling, addressing four critical gaps in the existing literature: the lack of systematic multi-model comparisons, the absence of principled prompt engineering analysis, the uncharacterized effect of requirement complexity, and the unexplored potential of iterative refinement.

We constructed a difficulty-stratified dataset of 30 UML class diagram exercises and proposed an automated weighted evaluation metric operating across three structural dimensions—classes (S_C), relationships (S_R), and attributes (S_A). We evaluated six models spanning general chat, reasoning-distilled, and thinking paradigms, across four prompting strategies (**S0–S3**) of increasing guidance richness, and five iterative refinement paths of increasing depth.

Our key findings are the following: (1) overall performance remains low across all conditions, confirming the genuine difficulty of the task; (2) model type determines both performance level and guidance responsiveness—general chat models lead in single-round performance but operate near their ceiling, while thinking-oriented models have latent capabilities better unlocked through structured guidance and iterative refinement; (3) prompting strategy effectiveness is model-dependent: reasoning-distilled models perform best under minimal guidance (**S0**), as externally imposed structured prompts interfere with their internalized reasoning processes, while other model types generally benefit from richer guidance; (4) relationship modeling is the universal and persistent bottleneck— S_R consistently lags behind S_C across all models, strategies, and difficulty levels, with near-collapse observed for smaller reasoning-distilled models on hard exercises; (5) S_A remains near zero under all conditions, identifying attribute extraction as a fundamentally unsolved problem; (6) iterative refinement benefits thinking-oriented models substantially but yields limited or negative returns for general chat models, confirming that its value is model-specific.

These research results collectively provide a detailed empirical basis for understanding the capabilities of LLM in UML models, and offer operational guidelines for researchers and practitioners who deploy these systems in actual software engineering processes.

6.2. Future Work

Our study opens several directions for future investigation:

Larger and more diverse datasets. Expanding the benchmark to hundreds of exercises spanning more application domains, requirement styles, and UML complexity patterns would enable finer-grained analysis of difficulty effects and strengthen the generalizability of our findings. In particular, the non-monotonic difficulty response observed in our 30-exercise dataset warrants investigation at larger scale.

Targeted relationship modeling interventions. Given that S_R is the universal bottleneck across all models and conditions, future work should develop and evaluate prompting strategies, fine-tuning approaches, and evaluation metrics specifically targeting relationship identification and typing. Relationship-focused chain-of-thought templates, contrastive examples highlighting relationship type distinctions, and relationship-specialized fine-tuning datasets represent promising directions.

Attribute extraction mechanisms. The near-zero S_A scores observed across all experimental conditions indicate that attribute extraction requires fundamentally different mech-

anisms from class and relationship modeling. Future work should investigate attribute-focused prompting, multi-stage generation pipelines that address attributes as a separate post-processing step, and specialized evaluation metrics for attribute completeness.

Fine-tuning for UML modeling. Investigating whether fine-tuning open-source models on UML-specific datasets can close the performance gap with general chat models and address the specific weaknesses identified in our study—particularly relationship modeling and attribute extraction—represents a high-value research direction with practical deployment implications.

Multi-diagram consistency and other UML types. Extending the study to other UML diagram types (sequence diagrams, use case diagrams, activity diagrams) and investigating cross-diagram consistency would address a broader range of software modeling tasks and enable more comprehensive assessment of LLM-based model-driven engineering capabilities.

Author Contributions: Conceptualization, Y.C., Y.H. and S.Y.; methodology, S.Y., Y.Q. and T.W.; software, L.Z. and X.F.; validation, Y.C., Y.H. and S.Y.; formal analysis, Y.Q. and L.Z.; investigation, X.F. and T.W.; resources, Y.C. and Y.H.; data curation, Y.H., S.Y. and Y.Q.; writing—original draft preparation, Y.H. and S.Y.; writing—review and editing, Y.C., Y.H., S.Y., Y.Q. and T.W.; visualization, L.Z. and X.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The experimental data of this paper can be found at the GitHub link <https://github.com/FlyingDwarfViolaMelon> (accessed on 1 May 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Prompting Strategy

Appendix A.1. S0 Prompting Strategy

You are a UML class diagram teacher with extensive experience in teaching PlantUML code generation. Please complete the UML class diagram generation task. The specific requirements are as follows:

User requirement: (user_requirement)

Core guiding prompt: Your task is to generate the corresponding UML class diagram based on the provided natural language requirements. There are no other additional requirements. Just complete it according to the output requirements.

Output Requirements:

1. Generate a UML class diagram that complies with engineering standards, clearly marking all classes, relationships between classes, and the attributes and methods of each class.
2. Output in PlantUML code format (The PlantUML code without markdown formatting).

Appendix A.2. S1 Prompting Strategy

You are a UML class diagram teacher with extensive experience in teaching PlantUML code generation. Please complete the UML class diagram generation task. The specific requirements are as follows:

User requirement: (user_requirement)

Core guiding prompt: Your task is to generate a UML class diagram that complies with the ****UML 2.5 standard****, strictly following the professional norms of UML class diagrams:

1. Accurately identify all the classes in the user requirements, and distinguish the attributes and methods of each class.

2. Precisely determine the types of relationships between classes, using only the UML standard relationships: association, inheritance, aggregation, composition, and dependency. Non-standard relationship expressions are prohibited.

3. Mark the degree of inter-class relationships according to the UML specification, and ensure that the degree is consistent with the business logic.

Output Requirements:

1. Generate a UML class diagram that complies with engineering standards, clearly marking all classes, relationships between classes, and the attributes and methods of each class.

2. Output in PlantUML code format (The PlantUML code without markdown formatting).

Appendix A.3. S2 Prompting Strategy

You are a UML class diagram teacher with extensive experience in teaching PlantUML code generation. Please complete the UML class diagram generation task. The specific requirements are as follows:

User requirement: (user_requirement)

Core guiding prompt: Your task is to generate a UML class diagram that complies with the **UML 2.5 standard** based on the provided natural language requirements. You must strictly follow the professional norms of UML class diagrams and follow the **fixed steps for modeling** to ensure a logical and rigorous process:

Step 1: Extract all the classes from the user requirements, eliminate redundant concepts, and clearly define the attributes and methods of each class;

Step 2: Analyze the business relationships between each pair of classes, accurately determine the relationship type (association/inheritance/aggregation/composition/dependency), and mark the corresponding multiplicity;

Step 3: Verify the compatibility of classes and relationships, ensuring that there are no missing classes, no incorrect relationships, and that the multiplicity and business logic are consistent.

Step 4: Integrate all classes and relationships according to UML standards to create a complete class diagram.

Output Requirements:

1. Generate a UML class diagram that complies with engineering standards, clearly marking all classes, relationships between classes, and the attributes and methods of each class.

2. Output in PlantUML code format (The PlantUML code without markdown formatting).

Appendix A.4. S3 Prompting Strategy

You are a UML class diagram teacher with extensive experience in teaching PlantUML code generation. Please complete the UML class diagram generation task. The specific requirements are as follows:

User requirement: (user_requirement)

Core guiding prompt: Your task is to create a UML class diagram that adheres to the **UML 2.5 standard** based on the provided natural language requirements. You must strictly follow the professional norms of UML class diagrams, break down the modeling process in a fixed sequence, and complete the self-check and error correction after generation to ensure that the result has no low-level errors:

Step 1: Extract all the core classes from the requirements, eliminate redundant concepts, and clearly define the core attributes and methods of each class;

Step 2: Analyze the business relationships between each pair of classes, accurately determine the relationship type (association/inheritance/aggregation/composition/dependency), and mark the corresponding multiplicity;

Step 3: Verify the compatibility of classes and relationships, ensuring that there are no missing classes, no incorrect relationships, and that the multiplicity and business logic are consistent.

Step 4: Integrate all classes and relationships according to UML standards to create a preliminary class diagram;

Step 5: Self-check and correct errors. The key points to check are: (1) Whether the core classes/relationship/attributes/methods are missing; (2) Whether the relationship types are marked incorrectly; (3) Whether the multiplicity is in contradiction with the business logic; (4) Whether it conforms to the PlantUML format specification; (5) Whether there are redundant classes/relationships. After correcting all the identified errors, please output the final class diagram.

Output Requirements:

1. Generate a UML class diagram that complies with engineering standards, clearly marking all classes, relationships between classes, and the attributes and methods of each class.
2. Output in PlantUML code format (The PlantUML code without markdown formatting).

References

1. Medvidovic, N.; Rosenblum, D.S.; Redmiles, D.F.; Robbins, J.E. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol. TOSEM* **2002**, *11*, 2–57. [[CrossRef](#)]
2. Ahmed, S.; Ahmed, A.; Eisty, N.U. Automatic transformation of natural to unified modeling language: A systematic review. In *IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA)*; IEEE: Piscataway, NJ, USA, 2022; pp. 112–119.
3. Babaalla, Z.; Jakimi, A.; Oualla, M. LLM-Driven MDA Pipeline for Generating UML Class Diagrams and code. *IEEE Access* **2025**, *13*, 171266–171286. [[CrossRef](#)]
4. Ferrari, A.; Abualhaija, S.; Arora, C. Model generation with LLMs: From requirements to UML sequence diagrams. In *IEEE 32nd International Requirements Engineering Conference Workshops (REW)*; IEEE: Piscataway, NJ, USA, 2024; pp. 291–300.
5. Liu, X.; Liu, Y.; Zhuang, Y.; Hou, W. UCD-LLM: A use case diagram requirement modeling multi-agent framework with large language model. *Inf. Softw. Technol.* **2025**, *190*, 107955.
6. Khamsepour, P.; Cole, M.; Ashraf, I.; Tan, D.; Puri, S.; Sabetzadeh, M.; Nejati, S. The Impact of Critique on LLM-Based Model Generation from Natural Language: The Case of Activity Diagrams. *arXiv* **2025**, arXiv:2509.03463.
7. Sharma, R.; Srivastava, P.K.; Biswas, K.K. From natural language requirements to UML class diagrams. In *IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*; IEEE: Piscataway, NJ, USA, 2015; pp. 1–8.
8. Dawood, O.S. From requirements engineering to uml using natural language processing—survey study. *Eur. J. Eng. Technol. Res.* **2017**, *2*, 44–50.
9. Abdelnabi, E.A.; Maatuk, A.M.; Hagal, M. Generating uml class diagram from natural language requirements: A survey of approaches and techniques. In *International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*; IEEE: Piscataway, NJ, USA, 2021; pp. 288–293.
10. Almazroi, A.A.; Abualigah, L.; Alqarni, M.A.; Houssein, E.H.; AlHamad, A.Q.M.; Elaziz, M.A. Class diagram generation from text requirements: An application of natural language processing. In *Deep Learning Approaches for Spoken and Natural Language Processing*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 55–79.
11. Yao, Y.; Duan, J.; Xu, K.; Cai, Y.; Sun, Z.; Zhang, Y. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confid. Comput.* **2024**, *4*, 100211. [[CrossRef](#)]
12. Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; Chen, H. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv* **2024**, arXiv:2408.02479.
13. Xia, C.S.; Deng, Y.; Dunn, S.; Zhang, L. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.* **2025**, *2*, 801–824. [[CrossRef](#)]
14. He, J.; Shi, J.; Zhuo, T.Y.; Treude, C.; Sun, J.; Xing, Z.; Du, X.; Lo, D. Llm-as-a-judge for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.* **2025**, 1–28. *accepted*.

15. Wang, R.; Guo, J.; Gao, C.; Fan, G.; Chong, C.Y.; Xia, X. Can llms replace human evaluators? An empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.* **2025**, *2*, 1955–1977.
16. Li, Y.; Keung, J.; Ma, X.; Chong, C.Y.; Zhang, J.; Liao, Y. Llm-based class diagram derivation from user stories with chain-of-thought promptings. In *48th Annual Computers, Software, and Applications Conference (COMPSAC)*; IEEE: Piscataway, NJ, USA, 2024; pp. 45–50.
17. De Bari, D.; Garaccione, G.; Coppola, R.; Torchiano, M.; Ardito, L. Evaluating large language models in exercises of uml class diagram modeling. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*; IEEE: Piscataway, NJ, USA, 2024; pp. 393–399.
18. Giannouris, P.; Ananiadou, S. NOMAD: A Multi-Agent LLM System for UML Class Diagram Generation from Natural Language Requirements. *arXiv* **2025**, arXiv:2511.22409.
19. Calamo, M.; Mecella, M.; Snoeck, M. Assessing the suitability of large language models in generating uml class diagrams as conceptual models. In *International Conference on Business Process Modeling, Development and Support*; Springer: Berlin/Heidelberg, Germany, 2025; pp. 211–226.
20. Garaccione, G.; Calabrese, D.M.; Coppola, R.; Ardito, L. A comparison of different Large Language Models for the generation of UML class diagrams. In *28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*; IEEE: Piscataway, NJ, USA, 2025; pp. 541–545.
21. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
22. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv* **2022**, arXiv:2203.13474.
23. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. *arXiv* **2024**, arXiv:2401.14196.
24. Kang, S.; Yoon, J.; Yoo, S. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 2312–2323.
25. Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–79. [[CrossRef](#)]
26. Pudlitz, F.; Brokhausen, F.; Vogelsang, A. Extraction of system states from natural language requirements. In *International Requirements Engineering Conference (RE)*; IEEE: Piscataway, NJ, USA, 2019; pp. 211–222.
27. Elalloui, M.; Nafil, K.; Touahni, R. Automatic transformation of user stories into UML use case diagrams using NLP techniques. *Procedia Comput. Sci.* **2018**, *130*, 42–49. [[CrossRef](#)]
28. Zakaria, B.; Bouziane, E.M.; Jakimi, A.; Oualla, N.; Saadane, R. Towards an approach for extracting uml class diagrams using advanced language models. In *International Conference on Smart City Applications*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 83–92.
29. Nguyen, J.; Koe, R.E.; Wang, F.; Arora, C.; Ferrari, A. Class Model Generation from Requirements using Large Language Models. *arXiv* **2026**, arXiv:2603.09100.
30. Camburn, B.; He, Y.; Raviselvam, S.; Luo, J.; Wood, K. Machine learning-based design concept evaluation. *J. Mech. Des.* **2020**, *142*, 031113. [[CrossRef](#)]
31. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 24824–24837. [[CrossRef](#)]
32. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
33. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **2023**, *55*, 1–35. [[CrossRef](#)]
34. White, J.; Fu, Q.; Hays, S.; Sandborn, M.; Olea, C.; Gilbert, H.; Elnashar, A.; Spencer-Smith, J.; Schmidt, D.C. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. In *Proceedings of the 30th Conference on Pattern Languages of Programs*, Monticello, IL, USA, 22–25 October 2023; pp. 1–31.
35. Kojima, T.; Gu, S.S.; Reid, M.; Matsuo, Y.; Iwasawa, Y. Large language models are zero-shot reasoners. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 22199–22213. [[CrossRef](#)]
36. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegrefe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. Self-refine: Iterative refinement with self-feedback. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 46534–46594. [[CrossRef](#)]
37. Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 8634–8652. [[CrossRef](#)]
38. Chen, X.; Lin, M.; Schärli, N.; Zhou, D. Teaching Large Language Models to Self-Debug. In *Twelfth International Conference on Learning Representations*; ICLR: Appleton, WI, USA, 2023; pp. 1–18.

39. Olausson, T.X.; Inala, J.P.; Wang, C.; Gao, J.; Solar-Lezama, A. Is Self-Repair a Silver Bullet for Code Generation? In *Twelfth International Conference on Learning Representations*; ICLR: Appleton, WI, USA, 2023; pp. 1–12.
40. Peng, B.; Li, C.; He, P.; Galley, M.; Gao, J. Instruction tuning with gpt-4. *arXiv* **2023**, arXiv:2304.03277.
41. Flesch, R.F. *The Art of Readable Writing: With the Flesch Readability Formula*; Harper & Brothers: New York, NY, USA, 1974.
42. Chissom, B.S. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. In *Scientific Studies of Reading*; Naval Technical Training Command of the United States Navy: Meridian, MS, USA, 1975.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.