A4Design: Generating and Correcting Design Diagrams with LLMs

Milo Piccioli, Carola Bonamico, Gabriele Agosta
Riccardo Mozzicato, Domenico Scalera
Polytechnic University of Turin
{s359009, s339129, s336818, s347492, s333304}@studenti.polito.it

Abstract

Generating formal software design artifacts from natural language requirements is a complex task that demands strict adherence to structural and logical constraints. While Large Language Models (LLMs) have shown promise in code synthesis, they frequently struggle to produce consistent and syntactically valid UML diagrams when operating in a single-shot manner. In this paper, we present **A4Design**, an automated framework that addresses these limitations through a cyclic multi-agent architecture. Unlike standard baselines, our approach decomposes the generation process into specialized phases of extraction, synthesis, and iterative refinement, leveraging feedback loops to self-correct syntax and logical inconsistencies. We evaluate the system using a robust semantic framework designed to measure domain fidelity beyond simple textual overlap. Our comparative analysis demonstrates that the multi-agent workflow enhances relationship mapping and syntactic validity, effectively eliminating common hallucinations found in direct generation approaches.

1 Introduction

Software design artifacts, particularly Unified Modeling Language (UML) diagrams, play a crucial role in documenting system architecture and facilitating communication among developers. However, keeping these diagrams synchronized with evolving software requirements is a labor-intensive and error-prone process. As a result, documentation often becomes missing or obsolete.

The recent advent of Large Language Models (LLMs) has shown promise in automating software engineering tasks (LLM4SE). While LLMs excel at generating code snippets, generating formal design diagrams presents unique challenges. This task requires not only syntactic correctness in description languages like PlantUML (Roques, 2009) but also strict logical consistency and adherence to structural constraints. Direct generation often leads to hallucinations, such as undefined relationships or invalid syntax.

In this work, we introduce **A4Design**[1], a framework designed to bridge the gap between textual requirements and formal design artifacts. Building upon the methodology proposed by De Bari et al. (2024), we investigate the capabilities of LLMs in the domain of software modeling by comparing a standard single-agent approach against a sophisticated multi-agent pipeline.

Specifically, this study addresses the following Research Questions (RQs):

- **RQ1:** *How accurate are LLMs at generating formal design diagrams from natural language?*
- **RQ2:** *Can multi-agent pipelines improve diagram consistency compared to single-agent baselines?*
- **RQ3:** *How effective are LLMs at detecting and correcting structural errors in diagrams?*

To answer these questions, we implemented a system capable of parsing natural language requirements to produce UML Class diagrams. We selected a set of 3 diverse software components (ranging from small management systems to workflow engines) to serve as our testbed. Our implementation leverages a multi-agent workflow that decomposes the task into retrieval, extraction, generation, and validation phases. We evaluate the system using a rigorous methodology that compares generated artifacts against a human-created gold standard, utilizing both structural metrics and semantic similarity embeddings.

[1]The source code is available at: https://github.com/LLM-29/A4Design

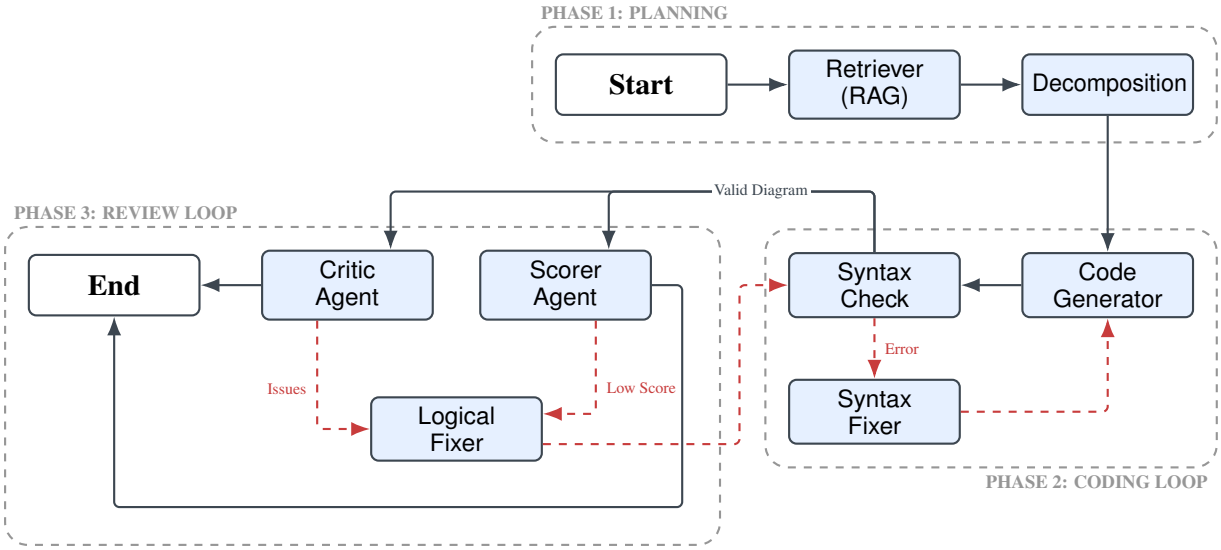


Figure 1: **System Architecture.** The workflow operates in three phases. After planning (Phase 1), the system enters a Coding Loop (Phase 2). Valid diagrams are then passed to the Review Loop (Phase 3), utilizing either a **Critic** or **Scorer** agent for semantic refinement.

2 Method

To address the research questions, we developed **A4Design**, a Python-based framework that automates the generation of UML Class diagrams. The core implementation relies on **LangChain** (Chase, 2022) for LLM abstraction and **LangGraph** (AI, 2024) for stateful agent orchestration. The overall system workflow is illustrated in Figure 1. The system relies on three shared components:

- **Model Manager:** Dynamically configures LLMs (via OpenRouter (OpenRouter, 2023)) with task-specific temperatures ($T = 0.15$ for extraction, $T = 0.0$ for generation).
- **Few-Shot RAG:** Uses **FAISS** (Johnson et al., 2019) to retrieve semantically similar examples, enriching the context.
- **PlantUML Tool:** A wrapper around the public **PlantUML API** (Roques, 2009) for stateless rendering and syntax validation, eliminating the need for local dependencies.

2.1 Pipeline 1: Single-Agent Baseline

The single-agent baseline serves as the control group to evaluate the intrinsic capabilities of the LLM without architectural scaffolding. We implemented this pipeline using `LangGraph` as a linear state machine with a single functional node.

Architecture. The workflow is defined as a directed acyclic graph $G_{single} = (V, E)$, where $V = \{\text{START}, \text{GENERATE}, \text{END}\}$ and $E = \{(\text{START}, \text{GENERATE}), (\text{GENERATE}, \text{END})\}$. The agent operates in a "one-shot" manner: it receives the requirement text $\mathcal{R}$ and attempts to produce the final diagram $\mathcal{D}_P$ in a single inference pass.

Implementation Details. The generation node utilizes the `ChatOpenAI` wrapper. To mitigate parsing errors common in raw string outputs, we leverage the **Structured Output** paradigm offered by modern LLM APIs (e.g., OpenAI Function Calling (OpenAI, 2023)). The model is constrained to return a JSON object conforming to a strict schema defined by the `SingleAgentOutput` Pydantic class:

$$\mathcal{D}_P = \text{LLM}(\theta_{gen}, \mathcal{P}_{sys} \oplus \mathcal{R}) \quad (1)$$

where θ_{gen} represents the model parameters for the generation task, $\mathcal{P}_{sys}$ is the system prompt containing instructions and PlantUML syntax rules, and $\oplus$ denotes concatenation. Unlike the multi-agent pipeline, this baseline lacks a feedback loop (‘try-catch‘ blocks are only used for logging system failures, not for model self-correction). Consequently, any hallucinated syntax or logical inconsistency in $\mathcal{D}_P$ persists in the final output.

2.2 Pipeline 2: Multi-Agent Workflow

To overcome the limitations of the single-shot baseline, we implemented a cyclic multi-agent architecture. We formalize this workflow as a state machine

modeled by a directed graph $\mathcal{G} = (V, E)$, where nodes V represent specialized agents and edges E represent state transitions. The state space $\mathcal{S}$ encompasses the cumulative context, including the requirements $\mathcal{R}$, the current diagram candidate $\mathcal{D}_t$ at iteration t, and the validation report $\mathcal{V}_t$.

2.2.1 Phase 1: Retrieval and Decomposition

The process begins with Retrieval-Augmented Generation (RAG). Given requirements $\mathcal{R}$, the *Retriever* agent queries a vector database (FAISS) to find a set of k similar examples $E_{few} = \{e_1, ..., e_k\}$ based on cosine similarity of embeddings:

$$E_{few} = \underset{e \in \text{DB}}{\text{top-k}} \left(\frac{\mathbf{v}_{\mathcal{R}} \cdot \mathbf{v}_e}{\|\mathbf{v}_{\mathcal{R}}\| \|\mathbf{v}_e\|} \right) \tag{2}$$

Subsequently, we employ a *Divide and Conquer* strategy. Instead of generating the diagram directly, two sequential agents decompose the problem into structured components:

1. **Class Extractor:** Identifies the set of classes C and their attributes.
2. **Relationship Extractor:** Conditioned on C, identifies the set of edges E_{rel} (associations, inheritance).

Both agents utilize **Structured Outputs** (JSON schemas) to guarantee strict adherence to the data model, eliminating parsing errors common in unstructured text generation.

2.2.2 Phase 2: Generation and Syntax Refinement

The decomposed plan is synthesized into an initial PlantUML code $\mathcal{D}_0$. This candidate enters the first feedback loop, governed by the *Syntax Checker* function f_{syn}:

$$f_{syn}(\mathcal{D}_t) = \begin{cases} \text{Phase 3} & \text{if } \mathcal{D}_t \text{ is valid} \\ \text{SynFix}(\mathcal{D}_t, \epsilon) & \text{otherwise} \end{cases} \tag{3}$$

The *Syntax Fixer* agent receives the compiler error message and the invalid code, iteratively refining $\mathcal{D}_t$ until it compiles or a maximum iteration limit T_{max} is reached.

2.2.3 Phase 3: Review and Refinement

Syntactically valid diagrams are subjected to a semantic review. We implemented two alternative feedback mechanisms to guide the refinement loop:

Strategy A: Critique and Convergence. In this configuration, a *Critic* agent generates a structured report $\mathcal{V}_{log}$ containing qualitative findings (e.g., missing methods). If critical errors are found, the *Logical Fixer* attempts to produce a corrected diagram $\mathcal{D}_{t+1}$. To prevent infinite oscillation, we employ a **Convergence Detection** mechanism based on the normalized Levenshtein similarity ratio (Levenshtein, 1966). We define the similarity $\mathcal{S}_{lev}$ between the current diagram $\mathcal{D}_t$ and the refined version $\mathcal{D}_{t+1}$ as:

$$\mathcal{S}_{lev}(\mathcal{D}_t, \mathcal{D}_{t+1}) = \frac{|\mathcal{D}_t| + |\mathcal{D}_{t+1}| - \text{dist}_{lev}(\mathcal{D}_t, \mathcal{D}_{t+1})}{|\mathcal{D}_t| + |\mathcal{D}_{t+1}|} \tag{4}$$

where $|\mathcal{D}|$ denotes the character length of the diagram and dist_{lev} is the standard Levenshtein edit distance. The loop terminates when stability is reached, i.e., $\mathcal{S}_{lev} \geq \tau_{conv}$ (e.g., 0.96).

Strategy B: Quality Scoring. Alternatively, a *Scorer* agent evaluates the diagram's compliance with requirements on a quantitative scale $S(\mathcal{D}_t) \in [1, 5]$. The refinement loop continues until the quality score satisfies a threshold condition:

$$S(\mathcal{D}_t) \geq \tau_{score} \tag{5}$$

where τ_{score} is an empirically determined cutoff (e.g., 3.14). If the score is insufficient, the feedback is passed to the Fixer for another iteration.

2.3 Formal Evaluation Framework

Evaluating generative models for formal languages requires rigorous definitions. Let a diagram be defined as a tuple $\mathcal{D} = (C, A, R)$, where C is the set of classes, A is the set of attributes, and R is the set of relationships. We denote the human-created Gold Standard as $\mathcal{D}_G$ and the LLM-generated diagram as $\mathcal{D}_P$. The evaluation task is to quantify the semantic overlap between $\mathcal{D}_G$ and $\mathcal{D}_P$.

Semantic Embedding Space. Since LLMs may generate synonymous terms (e.g., "Client" vs. "Customer") that refer to the same entity, exact string matching is insufficient. We map each extracted element e to a vector representation in $\mathbb{R}^d$ using a pre-trained BERT model (Devlin et al., 2019) ϕ:

$$\mathbf{v}_e = \phi(\text{label}(e)) \tag{6}$$

The similarity between a generated element $e_p \in \mathcal{D}_P$ and a gold element $e_g \in \mathcal{D}_G$ is computed using cosine similarity:

$$\text{sim}(e_p, e_g) = \frac{\mathbf{v}_{e_p} \cdot \mathbf{v}_{e_g}}{\|\mathbf{v}_{e_p}\| \|\mathbf{v}_{e_g}\|} \tag{7}$$

Table 1: **Detailed Performance by Scenario.** The top section shows breakdown by scenario. The **Overall Average** (bottom) highlights that the Multi-Agent Critic consistently outperforms the baseline, particularly in Relationship recovery (+23% average gain).

Scenario	Element	Single-Agent (Baseline)			Multi-Agent (Scorer)			Multi-Agent (Critic)		
		P	R	F1	P	R	F1	P	R	F1
1. Library System	Classes	1.00	0.67	0.78	1.00	0.83	0.78	1.00	0.83	**0.85**
	Attributes	0.50	0.40	**0.53**	0.60	0.30	0.49	0.60	0.30	0.51
	Relations	1.00	0.25	0.37	1.00	0.50	0.50	1.00	0.50	**0.56**
2. MyDocto Mgmt	Classes	1.00	0.60	0.62	1.00	0.70	**0.70**	1.00	0.50	0.66
	Attributes	0.13	0.13	0.08	0.20	0.13	0.02	0.08	0.07	**0.09**
	Relations	0.83	0.42	0.24	0.62	0.42	0.49	0.75	0.25	**0.55**
3. Online Shopping	Classes	0.65	0.65	0.63	0.75	0.75	0.74	0.75	0.75	**0.75**
	Attributes	0.00	0.00	0.07	0.33	0.17	**0.14**	0.60	0.12	0.11
	Relations	0.44	0.44	0.65	0.62	0.56	0.83	0.67	0.67	**0.84**
Overall Average	Classes	-	-	0.68	-	-	0.74	-	-	**0.75**
	Attributes	-	-	0.23	-	-	0.22	-	-	**0.24**
	Relations	-	-	0.42	-	-	0.61	-	-	**0.65**

Threshold Optimization. A naive evaluation would use a fixed threshold (e.g., 0.8) to decide if e_p matches e_g. However, the optimal threshold varies by domain. We define a dynamic matching function $\mathcal{M}_\tau$ where a match exists if $\text{sim}(e_p, e_g) \geq \tau$. To determine the optimal τ^*, we perform a grid search on a validation set. We maximize the F1-score over the threshold space $\tau \in [0, 1]$:

$$\tau^* = \arg\max_{\tau} \left(2 \cdot \frac{P(\tau) \cdot R(\tau)}{P(\tau) + R(\tau)} \right) \quad (8)$$

where $P(\tau)$ and $R(\tau)$ are the Precision and Recall calculated at threshold τ. This optimization ensures the system is robust against minor terminological variations while penalizing hallucinations.

3 Experiments

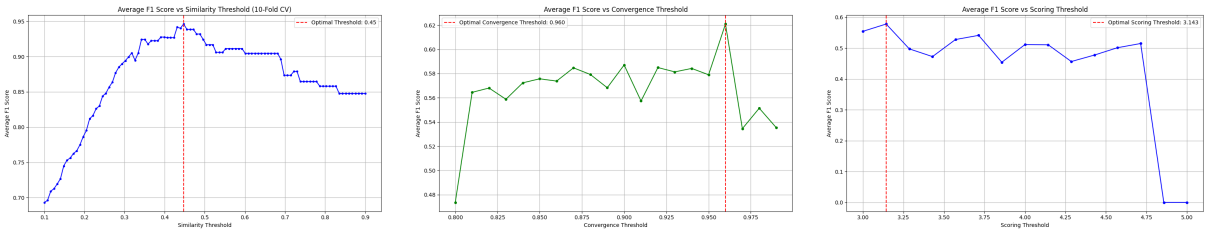
To validate the effectiveness of A4Design, we conducted a comparative analysis between the single-agent baseline and the proposed multi-agent workflows.

3.1 Experimental Setup

Following the methodology of benchmarks like **SWE-bench** (Jimenez et al., 2024), we framed our evaluation as a text-to-code generation task. The assessment relied on a curated dataset of 20 software design problems, specifically selected to represent a wide spectrum of complexity—ranging from low-coupling scenarios (e.g., a simple "To-Do List") to high-density domain models (e.g., "Hospital Management System"). To prevent data leakage, the dataset was strictly stratified: 3 exercises populated the FAISS vector index for RAG retrieval, 14 served as a validation set for hyperparameter tuning, and a held-out test set of 3 unseen complex scenarios ("Library Management", "MyDocto", "Online Shopping") was used for the final comparative analysis. Ground-truth alignment was established through a rigorous labeling process, where domain experts manually mapped natural language requirements to UML elements to create a reliable Gold Standard.

Model Configuration. Experiments were conducted using `nvidia/nemotron-3-nano-30b` (NVIDIA, 2023) as the core LLM, accessed via OpenRouter. We selected this model specifically for its favorable trade-off between reasoning capabilities and inference cost. Unlike standard instruction-tuned models, Nemotron generates an internal reasoning trace prior to the final response. This feature proved critical for our multi-agent workflow, as it allows the *Decomposition* and *Critic* agents to articulate intermediate logical steps before synthesizing to the strict JSON schemas required by our structured outputs. For the auxiliary components, we employed:

- **Embeddings:** `BAAI/bge-large-en-v1.5`, chosen for its superior performance in retrieval tasks.



(a) Semantic Threshold (10-Fold) (b) Convergence Threshold (c) Scoring Threshold

Figure 2: **Hyperparameter Optimization Landscape.** (a) The 10-fold cross-validation on the semantic matcher identifies the optimal cutoff for entity mapping. (b) The convergence metric peaks at $\tau_{conv} = 0.96$, filtering out unstable iterations. (c) The scoring threshold optimization identifies $\tau_{score} = 3.14$ as the optimal balance point for the Critic agent's quality assessment.

- **Evaluator:** `sentence-transformers/all-mpnet-base-v2` (Reimers and Gurevych, 2019) for computing semantic similarity during the scoring phase.
- **Compiler:** The public **PlantUML API** (Roques, 2009) for standardizing syntax validation across all runs.

Hyperparameters. We adopted a dual-temperature strategy to balance creativity and determinism: the decomposition agents operated at $T = 0.15$ to foster structural variety in entity discovery, while the Code Generator used a deterministic setting ($T = 0.0$) to minimize syntax errors. Threshold values were frozen after a two-stage optimization phase on the validation set, as illustrated in Figure 2:

1. **Semantic Matching:** We performed a 10-fold cross-validation on the extracted entities. As shown in Figure 2a, maximizing the F1-score against manual labels identified the optimal cosine similarity threshold at $\tau_{sem} \approx 0.45$.
2. **Termination Criteria:** We set $\tau_{conv} = 0.96$ for the convergence loop and $\tau_{score} = 3.14$ for the quality scorer loop based on stability peaks observed in the validation set.

3.2 Evaluation Metrics

Adopting the Quality Framework by Moody (2005), we assessed:

- **Syntactic Correctness:** The *Syntax Validity Rate* of generated PlantUML code, ensuring the output is renderable.
- **Semantic Completeness:** Precision (P), Recall (R), and F1-Score calculated on the extracted graph elements (Classes, Attributes, Relationships) against the ground truth. This metric penalizes both hallucinations (extra elements) and omissions.
- **Pragmatic Quality:** Assessed via the *Scorer* agent's penalty system, which quantifies layout clarity and naming convention adherence.
- **Efficiency:** The *Convergence Rate*, measuring the mean iterations required to reach stability, used to proxy the computational overhead of the multi-agent system.

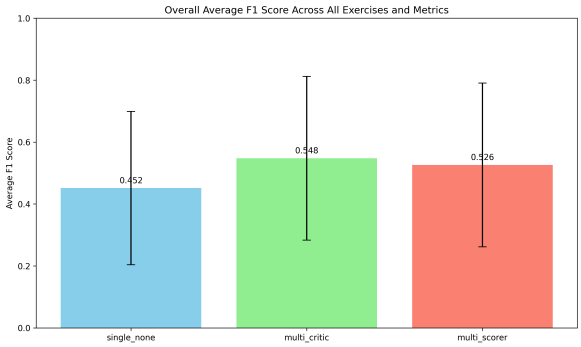


Figure 3: **Overall Performance Comparison.** Comparison of the three pipelines. The *Multi-Agent (Critic)* pipeline achieves the best performance (0.548), closely followed by the *Multi-Agent (Scorer)* pipeline (0.526). This confirms that both feedback mechanisms (qualitative critique and numerical scoring) are more effective than the single-agent baseline (0.452).

3.3 Results

RQ1 & RQ2: Accuracy and Consistency. Table 1 and Figure 3 summarize the comparative performance. The **Single-Agent** baseline demonstrates adequate performance on simple tasks but exhibits significant instability in complex domains. Specifically, in the "Online Shopping" scenario, it struggles to correctly map entity connections, resulting

in a Relationship F1 of 0.65 compared to 0.84 for the best model.

The **Multi-Agent Critic** achieves the highest fidelity and robustness across all metrics. As shown in the *Overall Average* section of Table 1, it outperforms the baseline in every structural category. Most notably, it demonstrates a substantial improvement in defining diagram topology, reaching an average **Relationship F1 of 0.65**, which represents a **55% relative improvement** over the Single-Agent baseline (0.42). This confirms that the Critic's iterative feedback loop effectively refines logical inconsistencies that single-shot models typically miss.

A slight trade-off is observed in attribute recall (0.23 for Single vs. 0.24 for Critic), indicating that the system correctly prioritizes high-level structural correctness (Classes and Relations) over fine-grained detail retention.

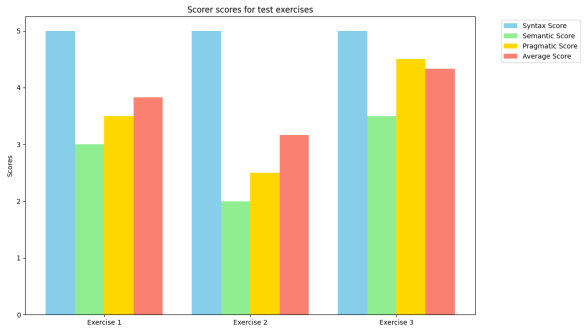


Figure 4: **Scorer's Qualitative Evaluation.** Breakdown of the quality scores across the three test scenarios as evaluated by the internal Scorer agent. The system achieves near-perfect *Syntactic Correctness* and competitive *Pragmatic Quality*, while the variance in *Semantic Completeness* reflects the intrinsic challenge of capturing fine-grained requirements.

To further investigate these nuances, Figure 4 illustrates the breakdown of quality scores (Syntactic, Semantic, Pragmatic) assigned by the Scorer across the test scenarios. The visual analysis highlights a critical strength of our pipeline: the **Syntactic Score** is consistently perfect (5.0), demonstrating that the *Syntax Fixer* successfully eliminates all compilation errors. **Pragmatic Quality** (yellow bars) also remains high, indicating readable layouts. The bottleneck is confirmed to be **Semantic Completeness** (green bars), which fluctuates significantly (e.g., in Exercise 2). This aligns with the lower Attribute Recall observed in Table 1, confirming that while the system produces valid and clean code, capturing the full depth of domain requirements remains the primary challenge.

RQ3: Error Correction and Threshold Optimization. We analyzed the termination criteria to validate the self-correction mechanism (Figure 2). For the **Critic workflow**, a grid search identified that a strict threshold of $\tau_{conv} = 0.96$ maximizes effectiveness. This high threshold acts as a stability filter, preventing the loop from terminating prematurely when the diagram is still undergoing significant structural flux. For the **Scorer workflow**, a grid search pinpointed $\tau_{score} = 3.14$ as the optimal cutoff, ensuring the loop continues only when significant quality gains are detected, thus avoiding infinite oscillations on marginal improvements.

4 Conclusion

In this paper, we introduced **A4Design**, a multi-agent framework designed to automate the generation of software design artifacts from natural language requirements.

By decomposing the generation process into specialized phases, namely Planning, Code Generation, and Iterative Review, we addressed the common limitations of Large Language Models in maintaining structural consistency and adherence to syntax constraints.

Our empirical evaluation indicates that the **Multi-Agent Critic** workflow performs slightly above the standard set by single-shot baselines. While the iterative feedback loop provided a noticeable improvement in relationship mappings (with a 55% relative gain), the moderate overall results highlight that the text-to-model transformation task remains intrinsically complex and prone to semantic ambiguity. We observe that while syntactic validity is largely solved, semantic fidelity is still a major hurdle. To bridge this gap, future work should explore integrating the Critic agent after every intermediate step (e.g., immediately after entity extraction) rather than only at the end, to intercept errors before they propagate. Furthermore, given the sophisticated reasoning required to model high-density domains, experimenting with larger, reasoning-oriented models may be necessary to achieve production-ready reliability. We believe that agentic workflows like A4Design represent a promising direction, but significant architectural and model-scale optimizations are required to surpass the current plateau.

References

LangChain AI. 2024. LangGraph: Building stateful, multi-actor applications with llms. https://github.com/langchain-ai/langgraph. Accessed: 2026-01-10.

Harrison Chase. 2022. LangChain: Building applications with llms through composability. https://github.com/langchain-ai/langchain. Accessed: 2026-01-10.

Daniele De Bari, Giacomo Garaccione, Riccardo Coppola, Marco Torchiano, and Luca Ardito. 2024. 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*. Association for Computing Machinery.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Preprint*, arXiv:1810.04805.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Vladimir I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10(8).

Daniel L. Moody. 2005. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data Knowl. Eng.*, 55(3).

NVIDIA. 2023. Nemotron-3: Technical report and model card. https://huggingface.co/nvidia/NVIDIA-Nemotron-3-Nano-30B-A3B-BF16. Accessed: 2026-01-28.

OpenAI. 2023. OpenAI API: Structured outputs and function calling. https://platform.openai.com/docs/guides/function-calling. Accessed: 2026-01-10.

Inc OpenRouter. 2023. OpenRouter: The unified interface for llms. https://openrouter.ai. Accessed: 2026-01-10.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Arnaud Roques. 2009. PlantUML: Open-source tool for drawing uml diagrams. https://plantuml.com. Accessed: 2026-01-10.