Article

Not peer-reviewed version

An Empirical Evaluation of Large Language Models Applying Software Architectural Patterns

Christos Hadjichristofi *, Michail Tsilimigkounakis, Georgios Sotiropoulos, Vassilios Vescoukis *

Posted Date: 31 March 2026

doi: 10.20944/preprints202603.2486.v1

Keywords: software architecture; software requirements; generative AI; LLM

 doi:10.20944/preprints202603.2486.v1

Article

An Empirical Evaluation of Large Language Models Applying Software Architectural Patterns

Christos Hadjichristofi [1,*], Michail Tsilimigkounakis [1], Georgios Sotiropoulos [1] and Vassilios Vescoukis [1,2,*]

[1] Software Engineering Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens, 15773 Athens, Greece
[2] School of Rural, Surveying and Geoinformatics Engineering, National Technical University of Athens, 15773 Athens, Greece
* Correspondence: hadjichristofi_ch@mail.ntua.gr (C.H.); v.vescoukis@cs.ntua.gr (V.V.)

Abstract

Except for coding, Large Language Models (LLMs) are increasingly explored as assistants for software design and architectural tasks. However, it remains unclear to what extent LLMs can reliably apply explicitly requested software architectural patterns when provided with user-defined specific requirements. In this paper, we empirically evaluate the ability of multiple LLMs to instantiate specific architectural styles under controlled conditions. We conduct a series of experiments in which models are prompted with problem descriptions expressed at different levels of structure, ranging from free-structured requirement lists to complete Software Requirements Specification (SRS) documents. The models are instructed, using single-shot prompts, to generate architectures in four representative styles: client–server, 3-tier, Model–View–Controller (MVC), and microservices. The authors assess the generated architectures with respect to structural correctness, requirement coverage, and adherence to the requested architectural pattern. Our results show that while LLMs can correctly apply simpler architectural patterns, performance decreases as architectural complexity and problem size increase. Model size and requirement representation significantly influence pattern adherence, whereas Retrieval Augmented Generation (RAG) exhibits mixed effects depending mainly on the material context and the LLMs capacity. These findings provide insight into the current capabilities and limitations of LLMs in architectural pattern application and inform the design of future AI-assisted architectural tools.

Keywords: software architecture; software requirements; generative AI; LLM

1. Introduction

Software architecture defines the high-level structure of a system by organizing its components, their responsibilities, and their interactions [1,2]. Architectural patterns such as client–server, 3-tier, Model–View–Controller (MVC), and microservices provide reusable solutions to recurring design problems and are widely adopted in industrial practice. Correctly applying such patterns requires interpreting requirements and translating them into coherent structural decisions that satisfy both functional and non-functional constraints.

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in source code generation [3,4], documentation synthesis [5], and various software engineering tasks [6,7]. As a result, LLMs are increasingly being explored as assistants for higher-level design activities, including architectural modeling. While recent studies explore the use of LLMs for generating software design artifacts and architectural models from natural language prompts [8,9], systematic evaluation of their ability to correctly apply explicitly requested architectural patterns that satisfy specific user-provided requirements remains limited.

Understanding this capability is essential before LLMs can be trusted for more autonomous or complex architectural design tasks. In particular, it is important to evaluate whether LLMs can (i)

 doi:10.20944/preprints202603.2486.v1

adhere to the constraints of specific architectural patterns, (ii) achieve complete requirement coverage, and (iii) produce structurally coherent designs under different prompting and model configurations.

In this paper, we empirically evaluate the ability of LLMs to apply explicitly requested software architectural patterns. We conduct controlled experiments in which multiple LLMs are prompted with problem descriptions expressed at different levels of structure, ranging from semi-structured requirement lists to complete Software Requirements Specification (SRS) documents. The models are instructed, using a single self-contained prompt, to generate architectures in four specific styles: client–server, 3-tier, MVC, and microservices.

The authors assess the generated architectures with respect to structural correctness, requirement coverage, and adherence to the requested architectural patterns. Through this study, we analyze how model size, model family, requirement representation, and Retrieval Augmented Generation (RAG) influence architectural pattern application.

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

1. To what extent can large language models correctly apply explicitly requested software architectural patterns by creating specific architectural diagrams?
2. How does requirements representation affect the ability of LLMs to apply architectural patterns?
3. How does RAG affect the quality of the LLM-generated diagrams?
4. Can LLMs calculate reliable quantitative metrics regarding the diagrams they generated?

2. Related Work

The automatic generation of software design models from textual descriptions has gained significant attention in recent years, driven by advances in LLMs. Earlier attempts relied largely on NLP pipelines supported by heuristic rules or ontology-based mappings, which enabled partial automation but required highly structured input and frequent manual intervention. These methods commonly struggled with ambiguity in requirements statements, resulting in incomplete or inconsistent models. Recent work therefore examines whether modern LLMs can interpret natural-language requirements and generate structured representations that approximate architectural intent.

Eisenreich, Speth, and Wagner proposed a structured six-step process for deriving software architectures from textual requirements [10]. Their framework begins with automatically generating a domain model and use-case scenarios, followed by manual refinement and the automatic derivation and evaluation of multiple architecture candidates. Their exploratory analysis using LLaMA2-70B and GPT-3.5 showed that, although LLMs can identify domain concepts, they often misunderstand prompts and struggle to produce correct structured representations such as PlantUML, highlighting the sensitivity of such approaches to prompt formulation and the continued need for human supervision.

At the level of UML extraction and diagram generation, several studies report similar limitations. Yang and Sahraoui [11] proposed a pipeline combining sentence classification with pattern-based fragment assembly and observed substantial semantic loss when transforming natural-language specifications into structured models. Subsequent empirical evaluations of LLM-generated UML artifacts further highlight these challenges. Cámara et al. [12] identified scalability issues and semantic inaccuracies in class diagrams enriched with OCL constraints. De Bari [13] found that, despite comparable syntactic quality, semantic errors were significantly more frequent in LLM-generated diagrams than in human-produced ones. Similar findings were reported by Al-Ahmad et al. [14], who observed moderate correctness levels and variability across different UML diagram types in student-centered evaluations, and by Krishnan et al. [15], who identified frequent syntactic and semantic errors, including incorrect relationship mappings and hallucinated elements, in the context of automated Use Case diagram generation. Collectively, these results indicate that while LLMs can assist in diagram generation, semantic reliability remains a persistent challenge.

Beyond diagram generation, recent work has explored whether LLMs can support architectural reasoning. Dhar et al. [16] investigated the automatic generation of Architectural Design Decisions and found that GPT-4 can produce coherent decisions but still falls short of human quality, while

 doi:10.20944/preprints202603.2486.v1

smaller fine-tuned models achieve comparable performance. Schindler and Rausch [17] addressed the inverse task of inferring architectural rules from code-level dependencies, showing that symbolic learners outperform LLMs, which tend to generate syntactically correct but semantically inconsistent rules when contextual cues are limited.

In addition, recent studies have begun examining the use of LLMs in software architecture practice more broadly. Jahić and Sami [18] report that, although LLMs are increasingly adopted for architectural tasks, practitioners note recurring issues such as non-reproducible outputs, shallow architectural reasoning, hallucinations, and limited ability to justify design choices, indicating that current models offer only partial support for architectural design activities. Ferrari and Spoletini [19], focusing on requirements engineering, warn that LLM-generated artifacts raise concerns of correctness and trustworthiness unless supported by formal specification and verification techniques. Although situated at the RE level, their arguments highlight the broader need for a careful evaluation of LLM-generated technical artifacts, including architectural models.

The present study examines how LLMs interpret and transform textual requirements into architectural structures across varying prompt formats and complexity levels. Unlike prior work that primarily evaluates diagrammatic quality or artifact-level correctness, this study focuses on the controlled application of explicitly requested architectural patterns. We examine whether LLM outputs adhere to defined structural constraints, reflect the intended architectural logic, and satisfy both functional and non-functional requirements across varying input representations and model configurations.

3. Experimental Design

Our experimental investigation was organized as a two-phase study on the ability of LLMs to generate software architectures, represented as UML class diagrams, that conform to predefined architectural patterns. Phase 1 examined three common architectural styles in the context of a small "toy" application, while Phase 2 focused on a more complex, modular application intended for deployment as a software-as-a-service system. In all cases, a single, self-contained prompt was used for every run. The prompt was not iteratively refined and no corrective feedback was provided. This decision was intentional in order to evaluate the models' ability to interpret a fixed problem specification and produce an architectural design without interactive guidance or prompt engineering. Each scenario was executed only once, as the objective of the study was to evaluate architectural pattern application under controlled input conditions rather than response variability across repeated runs. This strategy ensured that all executions were fully automated and directly comparable, allowing us to examine how different input representations and model configurations impact the structural quality of generated architectural designs when provided with the same information.

Across both phases, we varied the model type, input format, and retrieval configuration. The generated architectures were evaluated by the authors using a predefined assessment rubric that measured structural correctness and requirements-related quality. The main experimental variables were as follows:

- **Requirements format**: functional and non-functional requirements expressed either as textual lists (two variants) or as an SRS document, also in two variants, as discussed below.
- **Model type and size**: locally executed models with varying numbers of parameters, as well as publicly available commercial models.
- **Use of RAG**: enabled or disabled.
- **RAG source material**: architectural reference material obtained either from academic textbooks or from curated web-based descriptions of architectural patterns.
- **Embedding and retrieval configuration**: embedding model, chunking strategy, and retrieval parameters used in the RAG pipeline.

To instruct LLMs to generate architectures in a structured and machine-readable form, we requested the diagrams in Diagram-as-Code (DaC) format, and specifically PlantUML, which expresses UML diagrams using a concise textual syntax that can be generated directly by language models,

 doi:10.20944/preprints202603.2486.v1

which can then be rendered into a graphical representation using open source tools. Compared to XML-based UML formats such as XMI, PlantUML provides a significantly simpler textual representation that is easier for LLMs to generate and for researchers to inspect. At the same time, compared to other text-based diagram notations such as Mermaid, PlantUML offers broader UML support and a more mature ecosystem of tools and documentation. UML class diagrams were selected as the architectural representation because PlantUML does not offer adequate support for the more generic UML component diagrams. Still, class diagrams are suitable for this investigation, as they provide a way of describing the structural organization of software systems, with semantics for roles, responsibilities, and relationships between classes, which, in our context may represent any kind of component. Intentionally, we did not consider asking LLMs to produce diagrams in the form of images.

All materials used in the experiments, including prompt templates, requirements documents, RAG sources, and all generated outputs (PlantUML diagrams and metadata), are publicly available in an online repository to support reproducibility[1].

3.1. Experimental Setup

Local models were executed through the Ollama platform, which allows LLMs to run locally on the experimental hardware. A Python-based pipeline implemented with LangChain was used to construct prompts, invoke models, and optionally integrate RAG. When RAG was enabled, requirements documents and reference materials were first segmented into textual chunks, from which vector embeddings were generated and stored in the Chroma vector database. These embeddings were then used to retrieve relevant context for each prompt according to a consistent retrieval strategy. All experimental scenarios were defined in a structured table, where each row represented a single scenario. Each scenario specified the architectural pattern, the requirements materials, the prompt template, the model configuration (including RAG settings and embedding model), and the output path for generated artifacts. A Python script iterated over this table to automatically execute all scenarios, log execution metadata, and store the raw LLM responses, including the generated PlantUML diagrams.

3.2. Phase 1: Dummy Coordinate Converter

Phase 1 experiments asked LLMs to produce the architecture of "Dummy Coordinate Converter (DCC)" [20], a small application that manages point coordinates in Cartesian and Polar form. DCC supports conversion, storage, retrieval, modification, and deletion of points. The system is intentionally simple so that the experiments can focus on how LLMs respond to variations in requirement representation and architectural pattern selection. Before running the experiments, we implemented DCC in Java using the same architectural patterns, namely client–server, 3–tier, and MVC, and produced the corresponding UML class diagrams. These diagrams served as reference material. Exact replication by the LLMs was not required; however, the generated architectures were expected to adhere to the primary responsibilities and structural rules associated with each architectural pattern.

For the **requirement representations**, we prepared two sets of functional requirements (FR) and two sets of non-functional requirements (NFR) describing the same system at different levels of granularity. The first set (FR1) is user-oriented and grouped by high-level features such as creation, storage, and deletion, with internal logic left implicit. The second set (FR2) is implementation-oriented and organized by logic, data operations, and user interactions, with explicit descriptions of operations, conversion steps, and display updates. For NFRs, the first set specifies basic technology choices, while the second introduces explicit object-oriented design guidelines. We paired FR1 with NFR1 and FR2 with NFR2 to form two requirement bundles with different levels of structural detail.

To study the effect of richer specification documents, we also prepared two variants of an SRS. The concise variant (SRS1) covers the system scope and main use cases, while SRS2 follows the structure recommended by IEEE 29148 [21] standard and additionally includes business context, user roles, constraints, suggested technologies, and assumptions. Both SRS versions include supporting

[1] https://github.com/ntua/ai4softwarearch/tree/main

 doi:10.20944/preprints202603.2486.v1

PlantUML diagrams. In all cases, the surrounding prompt structure remained fixed so that differences in outputs could be attributed only to the requirements representation, model configuration, and model behavior.

A **single prompt template** was used for the DCC experiments [2,3]. The template specified the task, the target architectural pattern, the system description with either FR/NFR or SRS content, and explicit instructions for producing UML class diagrams in PlantUML, including attributes, methods, associations, and basic package structure. The prompt style remained unchanged across runs.

We evaluated a set of **local and online LLMs** with different parameter counts. Local models, hosted via Ollama and quantised to Q4_0 where applicable, included Llama 3.1 (8B), DeepSeek-R1 (70B), Command-R, Gemma2 (27B), Mixtral (8×7B), and Phi-3 Medium (128K). Online models included Claude Sonnet, DeepSeek-R1, Gemini, GPT-4o, and several variants of OpenAI o-series models.

For RAG experiments, we used two **embedding models**: nomic-embed-text-v1.5 and text-embedding-3-large. Three **RAG configurations** were used in the DCC phase: (1) an academic source derived from the architecture chapters of Sommerville's Software Engineering textbook [2], (2) the same material split into pattern-specific files designed to emphasize the requested architecture, and (3) a composite reference document constructed from curated web-based descriptions of MVC, 3-tier, and client–server architectures obtained from practitioner-oriented sources. All materials used in the RAG corpus are archived in the online replication package.

For each architectural pattern, we varied the requirement bundle, model, RAG configuration, and embedding choice, resulting in approximately five hundred experimental scenarios. Each generated diagram was subsequently evaluated following the procedure described in Section 3.4.

3.3. Phase 2: MyCharts App, Architectured as Microservices

Phase 2 extended the study to a more complex, yet still experimental, software app, *MyCharts* [22], a web-based service that allows non-expert users to create, manage, and download charts in different formats. The application includes multiple modules and data flows, making it suitable for architectural design using a microservices pattern. Before conducting the experiments, we designed a reference microservices architecture based on established microservices design principles. A corresponding UML class diagram was created to serve as a reference during the evaluation process. Exact replication by the LLMs was not required; instead, the reference architecture was used to assess whether the generated diagrams respected key structural properties of microservices systems, such as service decomposition, data ownership, and inter-service communication.

As in Phase 1, we used both **FR/NFR requirement bundles** and a **complete SRS document** as inputs. The FR/NFR inputs describe system functionality, inter-service communication, and deployment constraints in list form, while the SRS provides a consolidated specification including functional behavior, non-functional requirements, and contextual constraints. Equivalent prompt templates were used for the two requirement representations[4,5]. Both templates explicitly required a microservices architecture, summarized the MyCharts system, provided design guidelines such as loose coupling and independent deployment, and requested UML class diagrams in PlantUML format. In addition to generating architectural diagrams, the models were asked to compute a set of quantitative architecture metrics derived from their own diagrams. The rationale was that, since the architecture was generated by the models themselves, computing structural metrics from that representation should be straightforward. These metrics are described in Section 3.5.

Based on the results of Phase 1, the model set for Phase 2 was restricted to configurations that demonstrated stronger architectural performance. The **local models** included larger variants such as Llama 3.3, DeepSeek-R1 (70B), Gemma2 (27B), Mixtral (8×22B), and a smaller Mistral model, all

[2] DCC Prompt template with FR/NFR
[3] DCC Prompt template with SRS
[4] MyCharts Prompt template with FR/NFR
[5] MyCharts Prompt template with SRS

 doi:10.20944/preprints202603.2486.v1

quantised to Q4_0. The **online models** included recent versions of Claude Sonnet 3.7, DeepSeek-R1, Gemini, GPT-4o, several OpenAI o-series models, a hosted Mistral model, and Grok.

For **RAG**, we used two focused sources on microservices architecture patterns: a textbook chapter by Richardson [23], which provides conceptual and pattern-level guidance, and a practitioner-oriented book by Malhotra [24], which summarises practical design rules. These materials were embedded and retrieved using the same Chroma-based pipeline described in Phase 1. Experimental scenarios were constructed by varying the requirement representation, model configuration, RAG setting, and the presence or absence of the metrics-related prompt component. Each generated architecture was subsequently evaluated according to the procedure described in Section 3.4.

3.4. Qualitative Evaluation Dimensions

The final step of our experiment involved a qualitative assessment of the generated diagrams. The architectures were evaluated independently by the four authors using a predefined rubric. Each evaluator assigned scores on a 0–5 ordinal scale, where 0 indicates that the criterion is not satisfied and 5 indicates full satisfaction.

The evaluation dimensions differed between the two experimental phases, reflecting the distinct architectural complexity of the systems under study.

Phase 1 Evaluation Criteria

Phase 1 focused on client–server, 3–tier, and MVC architectures. The evaluation rubric therefore included four architectural dimensions:

- **Adherence to architecture**: whether the LLM-generated diagram respects the structural principles of the requested architectural pattern and assigns responsibilities to appropriate components.
- **Correctness of class relationships**: whether associations, dependencies, and directions of interaction between classes are appropriate.
- **Cohesion and coupling**: whether related responsibilities are grouped within appropriate components while unnecessary inter-class dependencies are avoided.
- **Consistency with requirements**: whether the generated architecture satisfies the specific in-context functional and non-functional requirements described in the input.

Phase 2 Evaluation Criteria

Phase 2 examined microservices architectures for a more complex system. The four Phase 1 dimensions were insufficient to capture microservices-specific design concerns. Therefore, the evaluation rubric was expanded to include the following seven dimensions:

- **Functional alignment and responsibility distribution**: whether each microservice corresponds to a bounded context and implements a focused set of functionalities.
- **Coupling and deployment independence**: whether services are loosely coupled and designed for independent deployment.
- **Cohesion**: whether each service maintains strong internal cohesion and whether use cases involve the appropriate set of services.
- **Data management**: whether each service owns and manages its own data rather than relying on shared databases.
- **Data consistency**: whether the design includes mechanisms for maintaining data consistency across services, for example through events or transactions.
- **Communication and flow control**: whether service interactions are implemented using appropriate coordination mechanisms such as choreography, orchestration, messaging systems, or API gateways.
- **Non-functional requirements**: whether the architecture satisfies the system-level non-functional requirements described in the input specification.

 doi:10.20944/preprints202603.2486.v1

3.5. Quantitative Metrics for Microservices

To complement the qualitative evaluation, we used a set of quantitative metrics introduced in prior research on microservices architecture [25–27].

Since the architecture diagrams were generated by the LLMs themselves, the prompt explicitly requested the models to compute these metrics from the diagrams they produced. This setup allowed us to examine whether the models were able not only to construct a microservices architecture but also to identify its structural properties. Because the architectures were generated within the same response, computing relatively simple structural metrics from that representation was expected to be a straightforward task for the models. Most of the evaluated LLMs also supported sufficiently large context windows; therefore prompt length was not expected to prevent the models from performing this additional reasoning step. To ensure reliable measurements, the same metrics were also independently computed by the authors, by inspecting the generated diagrams. These manually derived values were used as the ground truth for the quantitative analysis and for comparison with the values reported by the models.

At the architecture level, the following metrics were defined:

- **Statelessness index**

$$SI = \frac{\text{\#stateless services}}{\text{total services}}$$

- **Data ownership coverage**

$$DOC = \frac{\text{\#services with their own datastore}}{\text{total services}}$$

- **Service transaction share**

$$SST = \frac{\text{\#transaction-aware services}}{\text{total services}}$$

At the service level, we also measured operational complexity and dependencies:

- **Service interface complexity**

$$SIC(S) = \text{\#operations of service } S$$

- **Afferent service coupling**

$$AIS(S) = \text{\#services invoking } S$$

- **Efferent service coupling**

$$ADS(S) = \text{\#services invoked by } S$$

These metrics were used to assist the interpretation of qualitative ratings. In particular, SIC relate to functional alignment and cohesion; SI, AIS, and ADS capture aspects of coupling and deployment independence; DOC reflects data ownership principles; and SST relates to the presence of consistency mechanisms.

Certain architectural properties, such as communication style or satisfaction of non-functional requirements, could not be reliably quantified and were therefore assessed exclusively through qualitative evaluation.

4. Results

This section presents the results of both experimental phases and answers the research questions introduced in Section 1. The analysis draws on the qualitative evaluation dimensions described in Section 3.4, the quantitative microservices metrics defined in Section 3.5, and the observed behavior of the models during the experiments. The results include both visual and quantitative representations

 doi:10.20944/preprints202603.2486.v1

of the generated architectures. Several PlantUML class diagrams produced by the LLMs are presented as illustrative examples, accompanied by figures summarizing overall evaluation outcomes. Figure 7 reports the average architectural evaluation scores for each model across all experiments. Figures 8, 9, 10, and 11 illustrate the impact of specific experimental conditions, such as requirement representation and RAG. In these figures, the reported values represent the difference between the average scores obtained under two alternative conditions for each model.

4.1. Examples of Generated UML Diagrams

This subsection presents selected UML class diagrams generated by the evaluated LLMs in PlantUML format. The models produced PlantUML code as part of their responses, which we rendered into UML diagrams. The selected examples show typical outputs produced by the models and are discussed further in Section 4.2.

Figure 1 shows an example of a correct generated 3-tier architecture of DCC System. The diagram clearly separates the presentation, logic, and data access layers and includes appropriate classes within each layer.

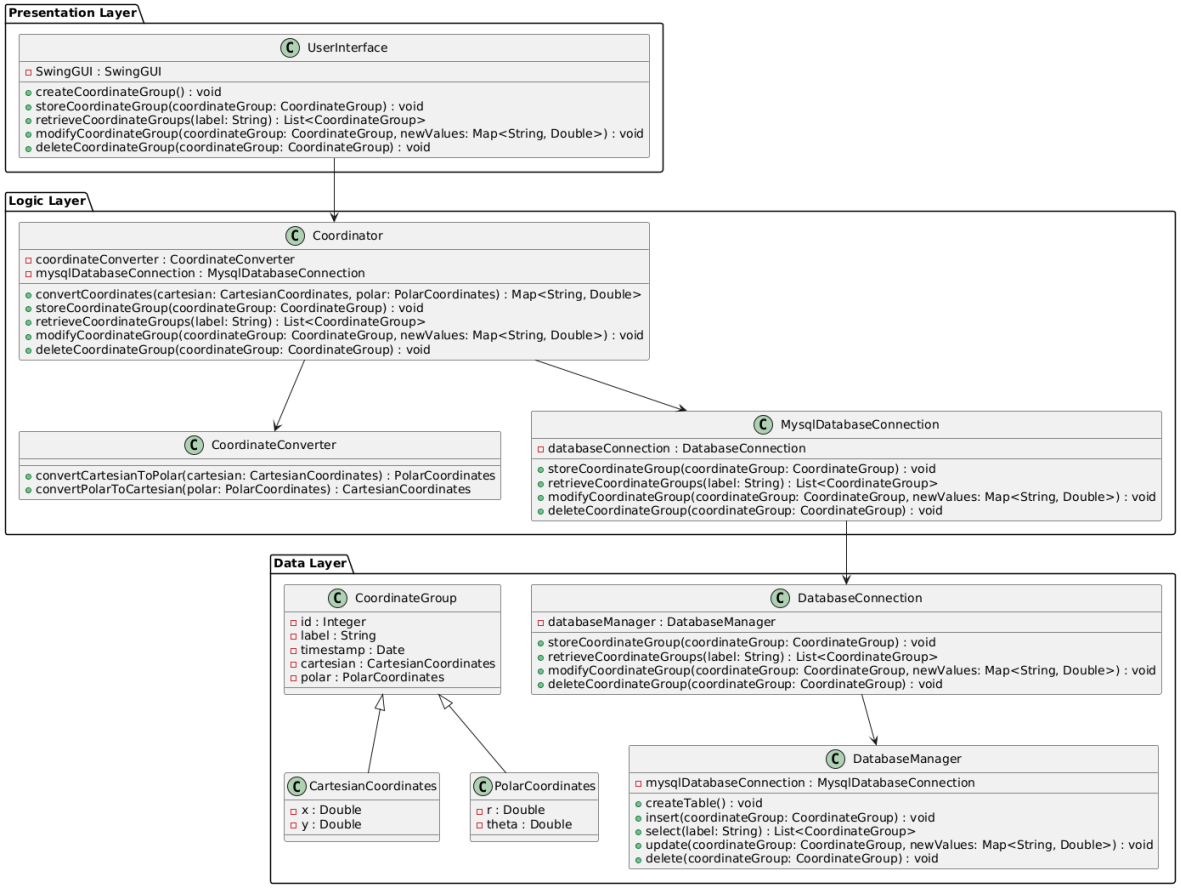

Figure 1. Example of a correct 3-tier architecture generated by Llama3.1 using FR1/NFR1 requirements. Available at: DCC FR-NFR 3-tier ID=85.

Figures 2 and 3 illustrate how requirement representation can influence the generated architecture. When prompted with the requirement lists, the commercial model produced an incomplete and structurally inconsistent MVC diagram (Figure 2). When prompted with the SRS, the same model generated a more coherent architecture (Figure 3).

 doi:10.20944/preprints202603.2486.v1

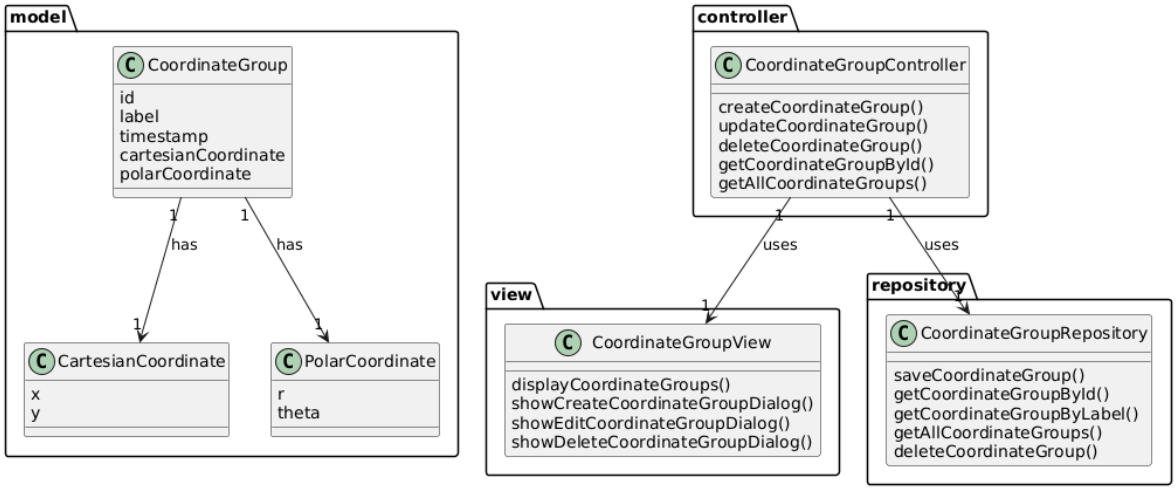

Figure 2. Example of an incorrect MVC architecture generated by Gemini using FR1/NFR1 requirements. Available at: DCC FR-NFR MVC ID=234.

 doi:10.20944/preprints202603.2486.v1

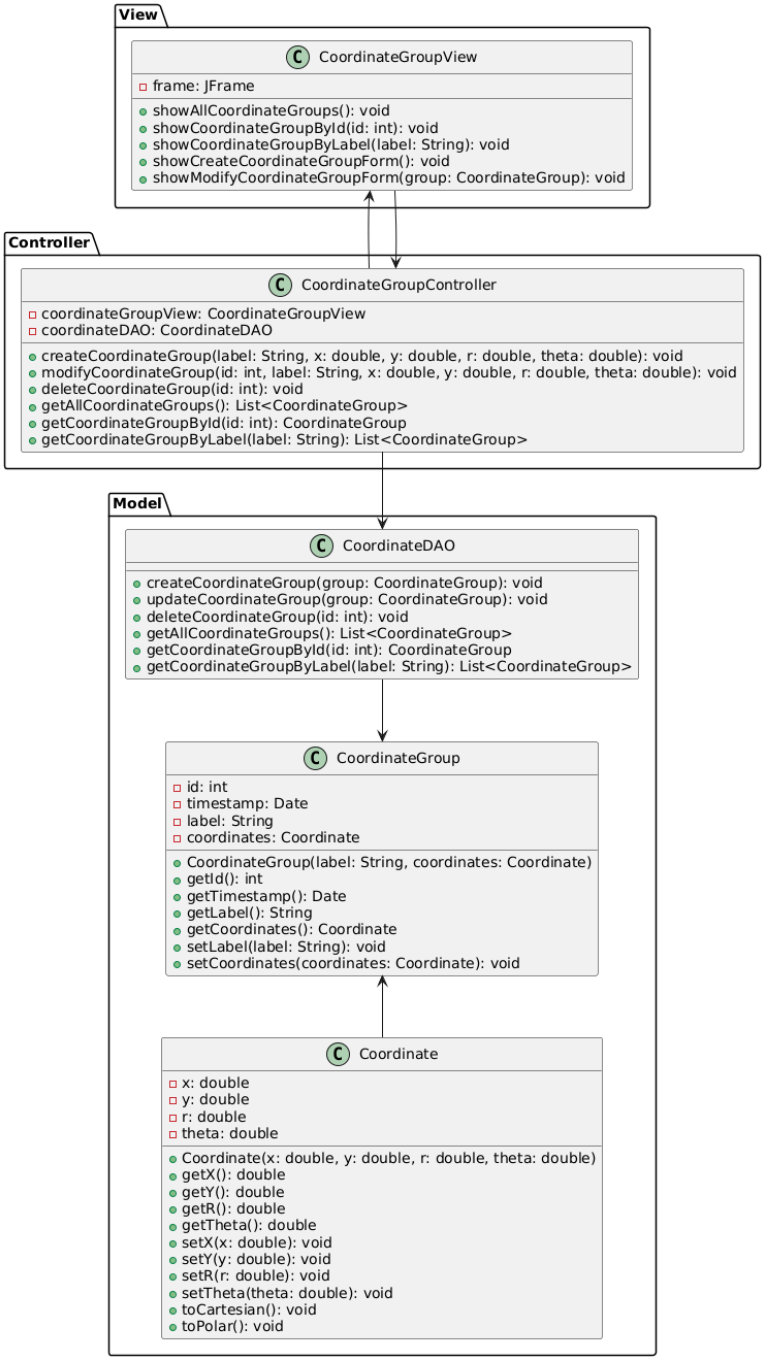

Figure 3. Example of a more coherent MVC architecture generated by Gemini using SRS1 requirements. Available at: DCC SRS MVC ID=69.

Figures 4 and 5 illustrate a case where the SRS representation resulted in a weaker architectural output for a local model. When using the SRS input, the generated diagram contains fewer methods and omits functionality (e.g. absence of conversion logic) expected from the target architecture (Figure 4). In contrast, the FR/NFR-based prompt produced a more detailed and structurally complete diagram (Figure 5).

 doi:10.20944/preprints202603.2486.v1

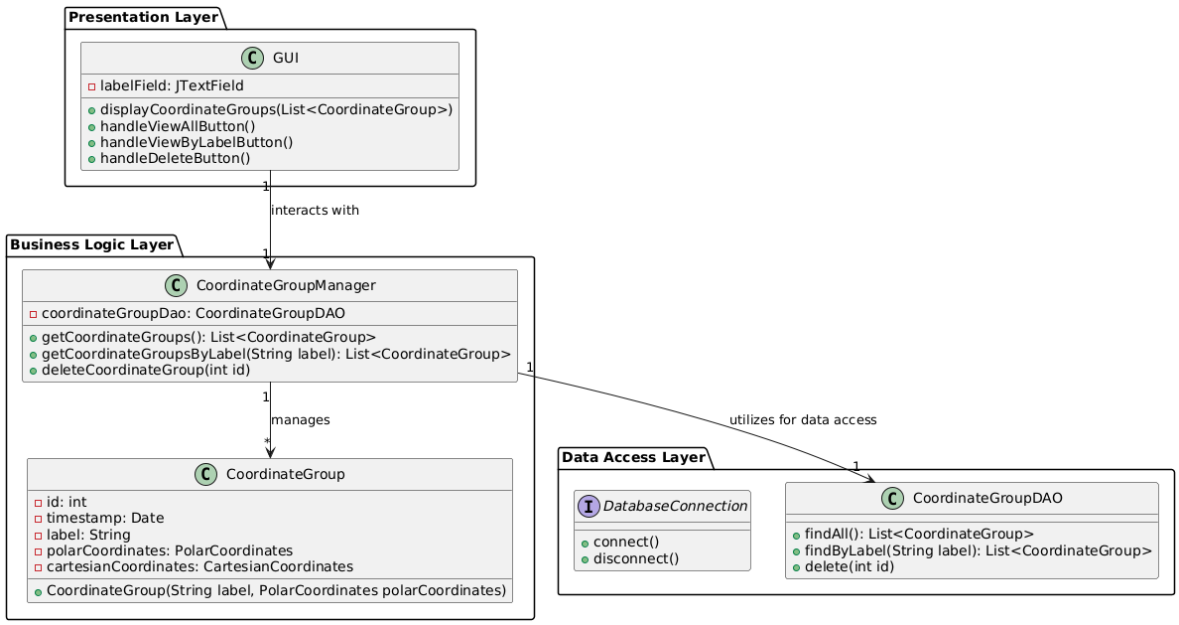

Figure 4. Example of a weaker 3-tier architecture generated by Gemma2:27b using SRS1 requirements with RAG enabled. Available at: DCC SRS 3-tier ID=10.

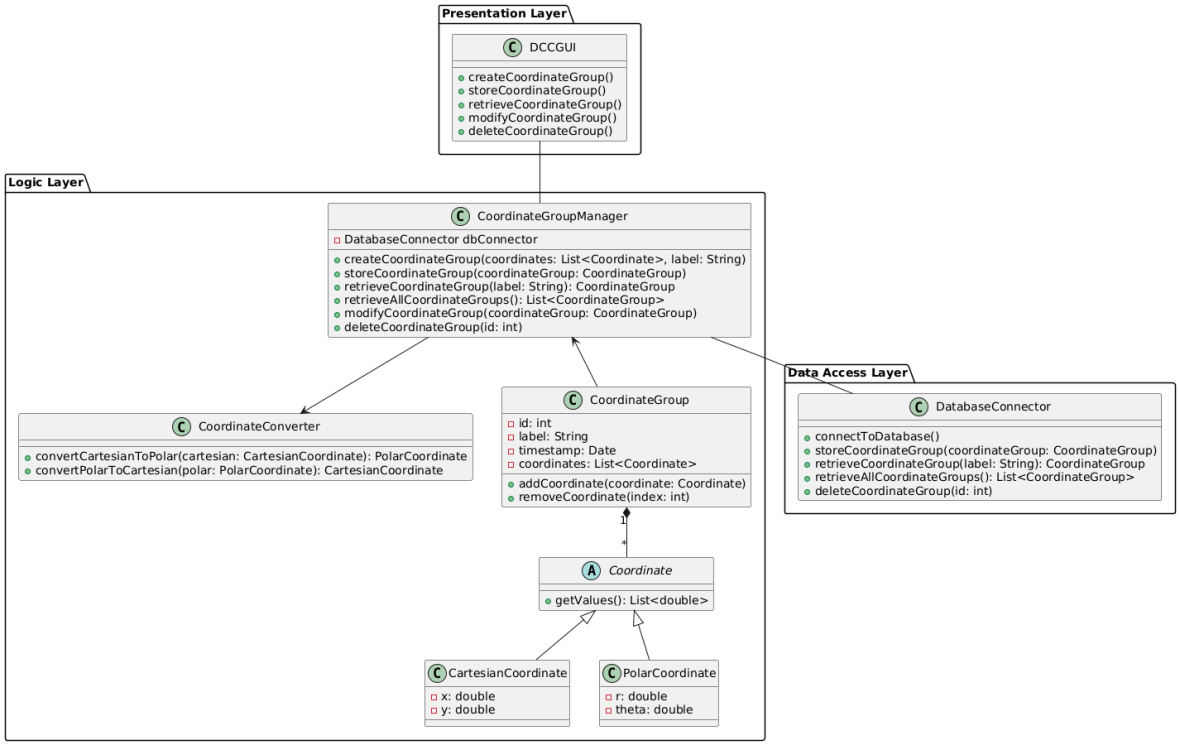

Figure 5. Example of a correct 3-tier architecture generated by Gemma2:27b using FR1/NFR1 requirements with RAG enabled. Available at: DCC FR-NFR 3-tier ID=23.

Finally, Figures 5 and 6 show the effect of RAG for the same model and prompt configuration. When RAG is applied, the generated diagram correctly instantiates the 3-tier architecture with proper layer separation (Figure 5). Without RAG, the model confuses responsibilities between layers and misrepresents several class relationships (Figure 6).

 doi:10.20944/preprints202603.2486.v1

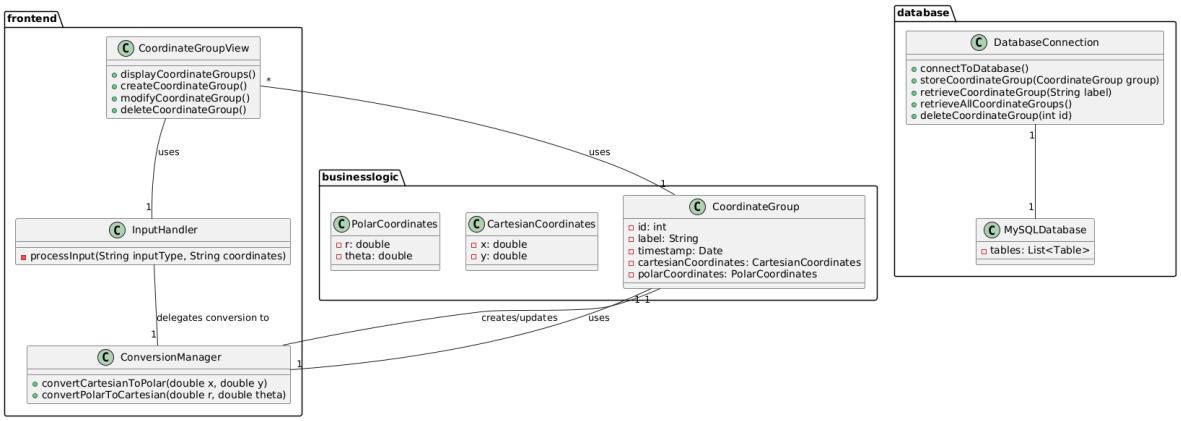

Figure 6. Example of an incorrect 3-tier architecture generated by Gemma2:27b using FR1/NFR1 requirements without RAG. Available at: DCC FR-NFR 3-tier ID=21.

4.2. Answers to Research Questions

4.3. RQ1: To What Extent Can Large Language Models Correctly Apply Explicitly Requested Software Architectural Patterns by Creating Specific Architectural Diagrams?

Figure 7 summarizes the average architectural evaluation scores achieved by each model across all experiments. Across both experimental phases, the models attempted to generate UML class diagrams that followed the requested architectural style. For the simpler "toy" system (DCC), most models produced diagrams that matched the general outline of the requested client–server, three-tier, or MVC patterns[6,7]. Even smaller and medium-sized models were generally able to apply these simpler patterns with reasonable structural alignment, particularly when provided with concise FR/NFR-based prompts (as illustrated in Figure 1). However, structural issues were common, including misplaced responsibilities (e.g., business logic within presentation layers), unnecessary tiers or classes, unclear separation of concerns, and inconsistencies with the requested pattern constraints. These deviations did not always break the overall architectural shape, but they required improvement to fully comply with the requested pattern.

[6] DCC FR-NFR client-server ID=3
[7] DCC SRS client-server ID=67

 doi:10.20944/preprints202603.2486.v1

Average performance per model / Experiment	dcc-srs-norag	dcc-srs-rag	mycharts-srs-norag	mycharts-srs-rag	dcc-text-norag	dcc-text-rag	mycharts-text-norag	mycharts-text-rag	Total
claudeSonnet3.7			5.0				5.0		**5.0**
o1-mini	4.5								**4.5**
claudeSonnet3.5	4.5				3.9				**4.2**
grok3			5.0				3.2		**4.1**
o1	4.5		4.4		3.4		4.0		**4.1**
gemini-2.0			4.2				3.9		**4.0**
gpt4oSAV	4.2				3.8				**4.0**
deepseek-r1	4.8		4.1				3.4		**4.0**
gemini-1.5	4.0				3.2				**3.6**
gpt4o	4.3		3.2		4.1		3.4		**3.6**
deepseek-r1:70b*	3.9	4.2	1.8	2.7	3.9	4.4	2.2	3.0	**3.1**
mistral-online			3.7				2.3		**3.0**
copilot					3.0				**3.0**
gemma2:27b*	3.5	3.2	1.9	2.0	3.0	3.5	3.0	3.3	**2.9**
o3-mini-high	2.9		1.5		2.8		4.2		**2.8**
llama3.3*			2.4	2.8			2.9	2.4	**2.6**
mixtral:8x7b*	2.3	2.1			2.6	3.0			**2.5**
command-r*	1.8	2.0			2.8	3.1			**2.4**
llama3.1:latest*	1.5	1.1			2.3	3.1			**2.0**
mixtral:8x22b*			1.1	2.4			2.6	0.1	**1.5**
phi3:medium-128k*	1.3	0.4			1.6	1.4			**1.1**
mistral*			1.4	0.9			2.2	0.0	**1.1**

Figure 7. Average performance per model across all experiments. Asterisk indicates local LLMs.

For the more demanding microservices architecture (MyCharts), correctness degraded noticeably. Only larger models, both commercial and large open-source, consistently preserved key microservices properties such as service-level data ownership, API-based loose coupling, and clear responsibility boundaries[8]. Smaller and medium-sized models frequently defaulted to centralized data storage, blurred service boundaries, or introduced structural inconsistencies, particularly when input complexity increased (e.g., SRS-based prompts)[9].

Overall, LLMs were only able to generate the high-level structure of well-known architectural patterns. However, the correct and in-context application of the desired architectural pattern varies by the complexity of the pattern itself and, of course, the model capacity. Model size is critical: larger models maintained structural integrity and pattern adherence more reliably, while smaller models exhibited increased degradation as architectural demands and input length increased.

4.4. RQ2: How Does Requirements Representation Affect the Ability of LLMs to Apply Architectural Patterns?

Requirements representation significantly influenced pattern application behavior. Structured FR/NFR lists were generally sufficient for simpler architectures such as the DCC system and often

[8] MyCharts FR-NFR msa ID=16
[9] MyCharts SRS msa ID=12

 doi:10.20944/preprints202603.2486.v1

guided LLMs to achieve clearer pattern alignment[10,11]. However, the internal organization of requirements text can also influence the final result apart from the form of problem description. The user-oriented FR1/NFR1 set, organized around high-level features such as creation and storage, did not implicitly guide LLMs to any specific architecture, and therefore tended to produce more unbiased outputs across patterns. In contrast, the implementation-oriented FR2/NFR2 set, which structures requirements by logic, data operations, and user interactions, introduced a noticeable bias: diagrams produced from FR2/NFR2 frequently applied 3-tier architectures, even when a different pattern was requested. In this example[12], the generated diagram follows a clear three-layer structure despite the prompt requesting an MVC architecture, likely influenced by the way the requirements are organized. This implies that in the context of this experiment, LLMs did not even attempt to satisfy the request to apply an explicit architectural pattern.

For more complex architectures, such as in the MyCharts system, however, FR/NFR lists still enabled large models to try to create a microservices design as requested, but the resulting architecture often lacked clear and well-defined service boundaries.

The effect of SRS documents that follow specific structures according to standards, also differed based on model size and capability. Even in the simple DCC experiment, only large online commercial models consistently produced better diagrams when given an SRS instead of freely-structured text, as depicted in Figure 8. We observed more complete coverage of the functional behavior and better targeting of the requested architecture by these LLMs (see Figures 2 and 3). For these models, the additional context and standardized semantics in the SRS, including DaC UML diagrams, were beneficial. In contrast, for local models, both small and large, the effect was neutral or even negative. With SRS inputs, these models often produced diagrams that neglected important elements or introduced inconsistencies that were less frequent with plain text, likely due to the substantial increase in prompt length. This suggests that, for such models, the extra size and complexity inherent in the SRS can minimize the benefit of the additional semantics.

In the MyCharts experiment, as shown in Figure 9, a similar pattern was observed: large models were able to exploit the SRS to improve the generated microservices architecture[13,14]. Most smaller models, however, degraded when moving to the SRS, generating architectures with more confusion in service boundaries and data responsibilities[15,16]. In this case, specific examples further highlight model size as the decisive factor: mixtral:8x22B, a very large local model, coped well with the SRS, whereas the smaller online model o3-mini-high performed worse. In summary, structured SRS documents help models that can cope with longer inputs and richer context, but they are not universally beneficial. For smaller models, a concise and well-structured requirements plain text appeared more effective than an SRS.

10 DCC FR-NFR client-server ID=539
11 DCC FR-NFR 3-tier ID=85
12 DCC FR-NFR MVC ID=296
13 MyCharts FR-NFR msa ID=17
14 MyCharts SRS msa ID=17
15 MyCharts FR-NFR msa ID=10
16 MyCharts SRS msa ID=10

 doi:10.20944/preprints202603.2486.v1

Evaluation dimension / Model	claudeSonnet3.5	gemini	deepseek-r1:70b	gemma2:27b*	chatgpt 4o	chatgpt o1	chatgpt o3-mini-high	llama3.1	mixtral:8x7b*	chatgpt 4oSAV	command-r*	phi3:medium-128k*
Effect of using reqs as SRS vs text in DCC (delta, max=5)												
adherence_to_architecture	0.6	0.6	-0.1	0.2	0.1	1.9	0.7	-1.8	-0.4	0.7	-0.9	-0.6
cohesion_coupling	0.7	0.7	-0.2	0.0	0.3	1.1	0.3	-1.6	-0.6	0.6	-1.1	-0.7
consistency_with_requirements	0.3	0.6	-0.8	-0.4	0.1	0.8	-0.5	-2.0	-1.0	0.1	-1.8	-0.8
correctness_relationships	0.9	1.1	0.0	0.0	0.5	0.8	0.1	-1.2	-0.9	0.5	-0.6	-0.4

Figure 8. Effect of using SRS vs text in DCC (Δ = SRS – text). Asterisk indicates local LLMs.

Evaluation dimension / Model	claudeSonnet3.7	gemini-2.0	deepseek-r1:70b*	gemma2:27b*	chatgpt 4o	chatgpt o1	o3-mini-high	llama3.3*	mixtral:8x22b	grok3	mistral*	mistral-online	deepseek-r1
Effect of using SRS vs text in MyCharts (delta, max=5)													
cohesion	0.0	1.0	-1.2	-1.7	0.5	0.5	-2.0	0.8	1.3	0.0	0.3	2.0	0.0
communication_flow	0.0	0.5	-1.9	-1.8	0.0	0.0	-3.0	0.2	1.3	2.0	-0.2	0.5	-0.5
coupling_independence	0.0	0.5	0.0	-1.3	1.0	0.5	-1.5	0.6	1.5	1.0	0.7	1.0	1.0
data_consistency	0.0	1.0	0.6	-1.5	-1.5	0.0	-4.0	-1.0	1.0	3.0	0.2	2.5	1.5
data_management	0.0	0.0	1.4	-2.7	-0.5	0.5	-4.5	0.3	0.0	4.5	0.3	4.0	3.5
functional_alignment	0.0	-0.5	-0.4	-0.2	0.5	0.5	-1.0	0.1	1.7	0.0	0.8	0.5	0.5
non_functional_requirements	0.0	0.0	-0.8	0.5	-1.0	0.5	-3.0	-0.2	0.7	2.0	-0.2	-0.5	-1.0

Figure 9. Effect of using SRS vs text in MyCharts (Δ = SRS – text). Asterisk indicates local LLMs.

4.5. RQ3: How Does RAG Affect the Quality of the LLM-Generated Diagrams?

In both experiments, RAG generally improved the quality of architectures produced by local models, but the effect was not uniform or consistent and depended on the input format and the model's capacity. Figure 10 shows the effect of RAG in the DCC experiment. When using FR/NFR input, most local models produced clearer architectures with better pattern alignment when RAG was enabled. Models such as command-r, llama3.1:latest, mixtral:8x7b, and gemma2:27b produced diagrams that followed the requested patterns more closely when additional contextual information was provided through RAG (see Figures 5 and 6). One exception was phi3:medium-128k, which showed a small decrease, indicating difficulty in integrating the additional information.

When RAG was applied together with SRS documents, the outcome was less favourable. The SRS inputs already increased the cognitive load for smaller models, and the added retrieval material sometimes amplified this effect. In these cases, RAG did not reliably improve results and at times even reduced clarity. Figure 11 presents the effect of RAG in the MyCharts experiment. Only, deepseek-r1:70B, the largest local model tested, showed clear and consistent improvement with RAG for both input types. For the other local models, the effect of RAG was inconsistent. This variability is due to the higher complexity of microservices architectures compared to simpler patterns, the smaller sample size for this configuration, and the limited ability of local models to process larger prompts.

 doi:10.20944/preprints202603.2486.v1

Evaluation dimension / Model	command-r*	deepseek-r1:70b	gemma2:27b*	llama3.1:latest*	mixtral:8x7b*	phi3:medium-128k*	command-r*	deepseek-r1:70b	gemma2:27b*	llama3.1:latest*	mixtral:8x7b*	phi3:medium-128k*
Effect of using RAG in DCC	reqs as text (delta, max=5)						reqs as SRS (delta, max=5)					
adherence_to_architecture	0.3	0.7	1.0	1.1	0.8	0.1	0.7	0.2	-0.5	-1.0	-0.3	-1.3
cohesion_coupling	0.2	0.5	0.3	0.7	0.3	-0.3	0.3	0.5	-0.3	-0.4	-0.1	-0.8
consistency_with_requirements	0.3	0.4	0.2	0.6	0.3	-0.2	0.1	0.3	-0.3	-0.3	-0.3	-0.8
correctness_relationships	0.1	0.6	0.5	0.5	0.4	-0.2	0.1	0.3	-0.3	-0.1	-0.2	-0.8

Figure 10. Effect of using RAG in DCC across input types (Δ = Average Scores with RAG applied – Average Scores with no RAG applied). Asterisk indicates local LLMs.

Evaluation dimension / Model	deepseek-r1:70b	gemma2:27b*	llama3.3*	mistral*	mixtral:8x22b		deepseek-r1:70b	gemma2:27b*	llama3.3*	mistral*	mixtral:8x22b
Effect of RAG in MyCharts	reqs as text (delta, max=5)						reqs as SRS (delta, max=5)				
cohesion	0.8	-0.3	-0.7	-4.0	-3.5		2.0	1.0	0.3	-0.5	2.3
communication_flow	3.3	0.3	-0.3	-3.0	-2.0		1.5	0.5	1.0	-1.0	1.5
coupling_independence	0.5	0.3	-0.2	-4.0	-3.3		0.0	1.3	0.5	-1.5	2.0
data_consistency	0.2	-1.5	-1.5	0.0	-1.0		2.0	-1.5	1.0	0.3	0.5
data_management	-0.8	1.8	1.0	0.0	0.3		-0.5	0.0	0.3	-0.3	0.3
functional_alignment	0.8	0.5	-0.3	-2.5	-4.0		0.8	0.3	0.0	-0.5	2.3
non_functional_requirements	1.0	1.3	-1.2	-2.0	-4.0		0.8	-1.0	-0.5	0.0	0.8

Figure 11. Effect of RAG in MyCharts across input types (Δ = Average Scores with RAG applied – Average Scores with no RAG applied). Asterisk indicates local LLMs.

The main empirical finding from the DCC experiment still holds. RAG can improve performance when the combined length of the base input and the retrieved context remains within the model's effective processing capacity. In MyCharts, this condition was met only by the largest local model. This indicates that, when specific conditions are met, even local models can be productive in architectural design tasks.

4.6. RQ4: Can LLMs Calculate Reliable Quantitative Metrics Regarding the Diagrams They Generated?

Architectural metrics, as mentioned in Section 3.5, were introduced explicitly only in the MyCharts microservices experiment, where the prompts described a set of microservices-related metrics and asked the models to compute these values in addition to generating the architecture. The metric values returned by the models themselves were often incorrect, indicating hallucination or "fabrication" of the result, as shown in Figure 12, which presents the success rate of metric computation, defined as the frequency with which the LLMs accurately computed the metric, where the values they produced matched those obtained through our manual calculations.

 doi:10.20944/preprints202603.2486.v1

Metric / Model	llama3.3*	deepseek-r1:70b*	mistral*	gemma2:27b*	mixtral:8x22b*	claudeSonnet3.7	deepseek-r1	gemini-2.0	gpt4o	o1	o3-mini-high	mistral-online	grok3
Correctness of LLM-reported metrics compared to human-calculated values													
SI	17%	33%	0%	33%	0%	0%	0%	50%	0%	0%	0%	0%	50%
DOC	33%	17%	0%	17%	33%	50%	0%	0%	50%	50%	0%	0%	0%
SST	17%	33%	0%	17%	33%	0%	0%	0%	50%	100%	0%	0%	50%
SIC	33%	33%	0%	50%	17%	50%	50%	0%	100%	0%	50%	50%	0%
AIS	33%	17%	0%	33%	0%	0%	0%	50%	0%	100%	0%	50%	0%
ADS	67%	0%	17%	33%	0%	50%	100%	50%	0%	100%	50%	50%	0%
TOTAL AVERAGE	33%	22%	3%	31%	14%	25%	25%	25%	33%	58%	17%	25%	17%

Figure 12. Correctness of LLM-reported metrics compared to human-calculated values. Asterisk indicates local LLMs.

These inconsistencies appeared across most LLM types, but their extent varied. Some larger LLMs produced metric values that were closer to the truth, obtained by manual calculation, while others exhibited substantial divergence. In several cases, LLMs seemed to extract and report metric values from general documentation about microservices, rather than from the elements they had actually just produced. Our experiments indicate that self-reported architectural metrics from LLMs should be treated with caution, given that nearly all models exhibited success rates below 50%. Even when the generated diagram is structurally reasonable, the accompanying metrics may be hallucinated and may not correspond to the actual diagram. At present, reliable use of metrics requires external computation, human validation, and further investigation.

5. Threats to Validity

As with any empirical study, several factors may affect the validity of the results. Following common practice in empirical software engineering research, we discuss potential threats in terms of internal, construct, and external validity.

5.1. Internal Validity

5.1.1. Subjectivity of Expert Evaluation

The architectural quality assessment relied on the authors' judgment. Four evaluators (two graduate engineers, one PhD researcher, and one experienced Software Architect, Professor of Software Engineering) independently scored each generated diagram. To reduce evaluation bias, a 0 to 5 ordinal scale with explicit anchors (0 = incorrect, 5 = correct) was provided for each evaluation criterion. Evaluators performed their assessments independently, and final scores were computed as the average of their ratings. Although this procedure reduces individual bias and does not depend on one single person, the evaluation of software architectures inherently involves a substantial degree of subjective interpretation. This is partially mitigated through the use of multiple independent evaluators and the aggregation of scores across reviewers; In all cases, the objective is to ensure that the results remain comparable

5.1.2. Single-Run Executions

Each experimental configuration (model × prompt type × RAG setting) was deliberately executed once. Considering that LLM outputs are not deterministic, repeated executions of the same configuration may produce different architectural diagrams. This study compensates for this limitation by evaluating a large number of scenarios across multiple models, prompt structures, requirement

 doi:10.20944/preprints202603.2486.v1

representations, RAG settings, and architectural patterns. The objective was to capture the general behavioral trends across models, rather than variability within the same model running the same prompt multiple times. Architectural diagram generation was designed as a single-step task that should not require iterative corrections. Iterative prompting and refinement strategies were intentionally avoided to ensure the results remained independent of any single user's approach or biases, therefore, preserving their comparability. Although iterative prompting may help the LLM apply the requested architectural pattern, our main focus was on comparability, which would be compromised by iterative prompting. Single-run executions therefore preserve the integrity of the comparative analysis by eliminating this potential source of subjective variability.

All generated outputs are archived and publicly available in an online repository, supporting transparency and reproducibility. Future work may extend the analysis by executing multiple runs per configuration in order to quantify output variance.

5.2. Construct Validity

5.2.1. Architectural Representation

The study evaluates generated architectures using UML class diagrams expressed in PlantUML. Class diagrams primarily capture static structural relationships and do not represent dynamic behavior, deployment topology, or runtime process views. Architectural frameworks such as the 4+1 view model emphasize the importance of multiple complementary views in architectural description. This limits the representational completeness of the evaluated architectures. However, the goal of this study is not to assess full architectural documentation or multi-view architectural reasoning. Instead, the objective is to evaluate whether LLMs can correctly instantiate explicitly requested structural architectural patterns. For the architectural styles considered in this study, client–server, 3-tier, MVC, and microservices, core architectural properties such as separation of concerns, layering constraints, service boundaries, and data ownership are primarily reflected in static structural relationships. UML class diagrams therefore provide a sufficient, controlled and comparable medium for assessing structural adherence to architectural patterns.

5.3. External Validity

5.3.1. Model Selection

The study evaluated a range of LLMs, including both commercial and open-source models available during the experiments, spanning multiple parameter scales. The goal was not exhaustive coverage of all existing LLMs, but representation across different model families and sizes. Consequently, the results primarily generalize to models with similar capabilities and parameter scales rather than to all existing or future LLMs. As the LLM ecosystem evolves rapidly, newly released models may exhibit different architectural reasoning capabilities.

5.3.2. Problem Scope and Architectural Patterns

The study evaluates LLMs using two experimental systems that represent commonly used architectural patterns: client–server, 3-tier, MVC, and microservices. Although these patterns are well established and widely documented in software architecture literature, they do not cover the full spectrum of architectural paradigms encountered in industrial systems. More complex architectures, such as distributed event-driven systems, domain-driven architectures, or highly domain-specific platforms, may introduce additional challenges that are not captured in this evaluation. The benchmark systems used in the experiments (DCC and MyCharts) were intentionally designed to provide controlled and interpretable evaluation scenarios. While this design enables systematic analysis of how LLMs interpret and instantiate architectural patterns, the systems are smaller and more constrained than typical industrial software systems. Consequently, the findings may not fully reflect the challenges associated with generating architectures for large-scale applications with complex domain constraints. In this regard, the reproducibility of good performance exhibited in our experiments is not guaranteed for larger and more complex software systems.

 doi:10.20944/preprints202603.2486.v1

Future research should therefore extend the evaluation to larger real-world systems and more diverse architectural styles.

6. Conclusions

This study investigated the ability of LLMs to generate software architectures from textual requirements while respecting explicitly requested architectural patterns. Through two experimental phases covering both simpler architectural styles (client–server, 3-tier, and MVC) and a more complex microservices architecture, we evaluated how requirement representation, model capacity, and RAG influence the structural quality of the generated architectures. A practical reason to consider using locally-hosted open-source LLMs for supporting architectural decisions would be to overcome the reluctance of developers to share proprietary software artifacts, such as requirement specifications, with LLMs. Our findings indicate that this could be possible under certain conditions.

The results indicate that LLMs are able to produce architectural designs that resemble well-known architectural patterns, particularly for simpler software systems. However, the correctness and structural consistency of these architectures depend strongly on both the capacity of the model and the structure of the requirements input. Larger models generally maintained architectural constraints more reliably, whereas smaller models exhibited increasing degradation as architectural complexity and prompt size increased. Requirement representation also influenced the results: concise lists of functional and non-functional requirements often produced clearer outputs for smaller models, while richer SRS documents benefited primarily models capable of processing longer and more complex inputs. Similarly, RAG improved the architectural quality in some scenarios by providing additional contextual knowledge, but its effectiveness depended on the relevance of the retrieved material and the model's ability to process the expanded prompt.

Beyond architectural generation itself, the study also examined whether LLMs can compute architectural metrics based on the diagrams they generate. While metrics calculated externally from the produced architectures proved useful for analysing structural properties of architectures, the metrics reported directly by the models were frequently incorrect and often inconsistent with the generated diagrams. This observation suggests that LLM-generated architectural artifacts should be interpreted cautiously and that automated evaluation of architectural properties still requires external verification mechanisms. It also indicates that requesting both architecture generation and metric computation within a single prompt may not constitute a reliable workflow. Future work could therefore explore staged approaches in which architectural models are generated first and structural metrics are subsequently derived through separate analysis steps or external tools.

Taken together, these findings highlight a broader implication: many of the longstanding challenges of requirements engineering and architectural design are reflected in LLM-based architectural generation. Requirements documents often vary in structure, completeness, and semantics, and architectural patterns themselves allow multiple valid interpretations. These variations are also present in the training data of LLMs and are likely to contribute to the diversity and inconsistency observed in the architectures they generate.

Several directions for future work follow from this study. First, diagram-as-code representations proved to be a practical format for interacting with LLMs, both for providing architectural context and for requesting structured outputs. Further investigation is needed to determine whether alternative notations and tools could improve model behavior. Although UML was used in this work due to its widespread adoption and established semantics, other approaches, such as the C4 model, may provide simpler and more architecture-oriented representations that could be better suited to LLM-based workflows. Similarly, evaluating additional diagram-as-code options beyond PlantUML may help identify representations that influence LLM reasoning more effectively.

Another promising direction concerns the role of XMI as a machine-readable representation of UML models. In principle, XMI should be able to support semantically-rich, structured and interoperable exchange of UML content [28]. In practice, however, our experiments showed that

 doi:10.20944/preprints202603.2486.v1

LLMs struggle to produce syntactically valid and semantically correct XMI representations, a difficulty also observed in recent studies [29]. Further research is required to determine whether improved prompting strategies or tool support could make XMI usable in LLM-driven architectural workflows.

A related research direction is the exploration of architecture-as-code approaches, in which software architectures are described directly using programming languages or executable specifications [30]. Such representations may provide clearer semantics and allow architectural metrics to be extracted automatically through static code analysis tools. Investigating whether LLMs can more reliably interpret and generate architecture-as-code representations may provide an alternative to diagram-based architectural modeling. The source code produced could be easily reverse-engineered to create any diagrammatic representation.

Finally, our experience indicates the need for a common benchmark for evaluating LLMs in software architecture generation. Current studies employ different systems, requirements descriptions, evaluation criteria, and prompting strategies, making it difficult to compare results across experiments. Establishing a shared benchmark consisting of representative architectural problems, requirement specifications, and agreed evaluation dimensions would support more systematic progress in this research area. The experimental setup and artifacts produced in this study aim to contribute towards this direction by providing an initial empirical foundation for understanding how requirement representation, model capacity, and retrieval mechanisms influence the ability of LLMs to generate software architectures.

Author Contributions: Conceptualization, C.H. and V.V.; methodology, C.H. and V.V.; software, C.H., M.T. and G.S.; validation, C.H., M.T. and G.S.; formal analysis, C.H., M.T. and G.S.; investigation, C.H., M.T. and G.S.; data curation, C.H.; writing—original draft preparation, C.H.; writing—review and editing, C.H., M.T., G.S. and V.V.; visualization, C.H.; supervision, V.V.; project administration, V.V. 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 data and materials supporting the results of this study are publicly available on GitHub at https://github.com/ntua/ai4softwarearch/tree/main and are distributed under a Creative Commons license.

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

References

1. International Organization for Standardization. ISO/IEC/IEEE 42010:2022, Software, Systems and Enterprise - Architecture Description, 2022.
2. Sommerville, I. *Software Engineering*, 10 ed.; Pearson Education Limited: Harlow, UK, 2016. Global Edition.
3. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code, 2021, [arXiv:cs.LG/2107.03374]. https://doi.org/10.48550/arXiv.2107.03374.
4. Wang, J.; Chen, Y. A Review on Code Generation with LLMs: Application and Evaluation. In Proceedings of the 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI), 2023, pp. 284–289. https://doi.org/10.1109/MedAI59581.2023.00044.
5. Naimi, L.; Bouziane, E.M.; Jakimi, A.; Saadane, R.; Chehri, A. Automating Software Documentation: Employing LLMs for Precise Use Case Description. *Procedia Computer Science* **2024**, *246*, 1346–1354. 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024), https://doi.org/10.1016/j.procs.2024.09.568.
6. Eramo, R.; Said, B.; Oriol, M.; Bruneliere, H.; Morales, S. An architecture for model-based and intelligent automation in DevOps. *Journal of Systems and Software* **2024**, *217*, 112180. https://doi.org/10.1016/j.jss.2024.112180.

 doi:10.20944/preprints202603.2486.v1

7. Bouzenia, I.; Devanbu, P.; Pradel, M. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 2188–2200. https://doi.org/10.1109/ICSE55347.2025.00157.
8. Ferrari, A.; Abualhaija, S.; Arora, C. Model Generation with LLMs: From Requirements to UML Sequence Diagrams. In Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW), 2024, pp. 291–300. https://doi.org/10.1109/REW61692.2024.00044.
9. Gheorghita, S.; Irimia, C.I.; Iftene, A. Automating Software Diagram Generation with Large Language Models. *Procedia Computer Science* **2025**, *270*, 713–722. 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025), https://doi.org/10.1016/j.procs.2025.09.191.
10. Eisenreich, T.; Speth, S.; Wagner, S. From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures. In Proceedings of the Proceedings of the 1st International Workshop on Designing Software, New York, NY, USA, 2024; Designing '24, pp. 52–55. https://doi.org/10.1145/3643660.3643942.
11. Yang, S.; Sahraoui, H. Towards automatically extracting UML class diagrams from natural language specifications. In Proceedings of the Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, New York, NY, USA, 2022; MODELS '22, pp. 396–403. https://doi.org/10.1145/3550356.3561592.
12. Cámara, J.; Troya, J.; Burgueño, L.; Vallecillo, A. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* **2023**, *22*, 781–793. https://doi.org/10.1007/s10270-023-01105-5.
13. 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 Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, New York, NY, USA, 2024; ESEM '24, pp. 393–399. https://doi.org/10.1145/3674805.3690741.
14. Al-Ahmad, B.; Alsobeh, A.; Meqdadi, O.; Shaikh, N. A Student-Centric Evaluation Survey to Explore the Impact of LLMs on UML Modeling. *Information* **2025**, *16*. https://doi.org/10.3390/info16070565.
15. G S, N.K.; S, A.; Thushara, M.G. Comparative Analysis of Large Language Models for Automated Use Case Diagram Generation. In Proceedings of the Proceedings of the 3rd International Conference on Futuristic Technology - Volume 2: INCOFT. INSTICC, SciTePress, 2025, pp. 465–471. https://doi.org/10.5220/0013594700004664.
16. Dhar, R.; Vaidhyanathan, K.; Varma, V. Can LLMs Generate Architectural Design Decisions? - An Exploratory Empirical Study. In Proceedings of the 2024 IEEE 21st International Conference on Software Architecture (ICSA), 2024, pp. 79–89. https://doi.org/10.1109/ICSA59870.2024.00016.
17. Schindler, C.; Rausch, A. Formal Software Architecture Rule Learning: A Comparative Investigation between Large Language Models and Inductive Techniques. *Electronics* **2024**, *13*. https://doi.org/10.3390/electronics13050816.
18. Jahić, J.; Sami, A. State of Practice: LLMs in Software Engineering and Software Architecture. In Proceedings of the 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), 2024, pp. 311–318. https://doi.org/10.1109/ICSA-C63560.2024.00059.
19. Ferrari, A.; Spoletini, P. Formal requirements engineering and large language models: A two-way roadmap. *Information and Software Technology* **2025**, *181*, 107697. https://doi.org/10.1016/j.infsof.2025.107697.
20. Tsilimigkounakis, M. Exploring the Utilization of LLM Tools in Software Architecture. Master's thesis, School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece, 2024.
21. International Organization for Standardization. ISO/IEC/IEEE 29148:2018, Systems and Software Engineering - Life Cycle Processes - Requirements Engineering, 2018.
22. Sotiropoulos, G. Investigation of AI tools performance in the definition of Microservices Software Architectures. Master's thesis, School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece, 2025.
23. Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications: Shelter Island, NY, USA, 2019.
24. Malhotra, N. Microservices Design Patterns. ValueLabs, 2023.
25. Engel, T.; Langermeier, M.; Bauer, B.; Hofmann, A. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In Proceedings of the Information Systems in the Big Data Era, Cham, 2018; pp. 74–89. https://doi.org/10.1007/978-3-319-92901-9_8.

 doi:10.20944/preprints202603.2486.v1

26. Bogner, J.; Wagner, S.; Zimmermann, A. Towards a practical maintainability quality model for service- and microservice-based systems. In Proceedings of the Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, New York, NY, USA, 2017; ECSA '17, pp. 195–198. https://doi.org/10.1145/3129790.3129816.
27. Bogner, J.; Wagner, S.; Zimmermann, A. Automatically measuring the maintainability of service- and microservice-based systems: a literature review. In Proceedings of the Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, New York, NY, USA, 2017; IWSM Mensura '17, pp. 107–115. https://doi.org/10.1145/3143434.3143443.
28. Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.5.1. Technical report, OMG, 2015.
29. Pan, F.; Petrovic, N.; Zolfaghari, V.; Wen, L.; Knoll, A. LLM-enabled Instance Model Generation. *arXiv preprint arXiv:2503.22587* **2025**.
30. Bucaioni, A.; Di Salle, A.; Iovino, L.; Pelliccione, P.; Raimondi, F. Architecture as Code. In Proceedings of the 2025 IEEE 22nd International Conference on Software Architecture (ICSA), 2025, pp. 187–198. https://doi.org/10.1109/ICSA65012.2025.00027.