Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Semi-Automated Qualitative Software Architecture Risk Analysis with ATAM using Retrieval-Augmented Generation

Huy Le Dac

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Sandro Speth, M.Sc Tobias Eisenreich, M.Sc (Technical University of Munich)
Commenced:	September 28, 2024
Completed:	March 28, 2025

Abstract

Context. The *Architecture Tradeoff Analysis Method (ATAM)* is an established approach for systematically evaluating software architectures, focusing on the identification of trade-offs, risks, and sensitivity points based on quality attribute scenarios. It has become a core method in the domain of qualitative software architecture analysis.

Problem. Despite its effectiveness, the traditional ATAM process is highly manual and resource-intensive. It requires considerable effort from stakeholders, domain experts, and architects, which often makes the process time-consuming and inefficient, particularly in early-stage architectural evaluations.

Objective. With the emergence of Artificial Intelligence and Large Language Models (LLMs), new opportunities arise for automating knowledge-intensive processes such as ATAM. This thesis proposes a semi-automated qualitative analysis tool based on Retrieval-Augmented Generation (RAG) to assist in identifying architectural trade-offs, sensitivity points, and risks from structured inputs.

Method. We began with a literature review of existing approaches for (semi-)automated software architecture evaluation. Subsequently, a prototype combining ATAM and RAG was developed, capable of processing structured inputs and retrieving relevant architectural knowledge from document corpora. Prompt engineering was applied to guide the model's responses. The prototype was evaluated through comparative experiments (manual vs. automated analysis) and a user study.

Result. The tool with its hardware configurations reduced analysis time by approximately 90% compared to the manual approach. Many generated risks, trade-offs, and sensitivity points were deemed relevant by experts. However, the tool struggled with deeply contextual insights, occasionally producing redundant or generic content. The user study revealed strengths in responsiveness and usability but also identified challenges in output content, especially lacking in depth.

Conclusion. The RAG-enhanced ATAM prototype demonstrates potential to augment architectural analysis by improving efficiency and partially automating risk identification. Nevertheless, it does not yet replace expert-driven judgment. Limitations include dependency on prompt quality, and occasional lack of depth in the results. Future work should focus on enhancing prompt design, integrating visual model interpretation, and improving guidance and explanation features to make the tool more robust and widely applicable.

Kurzfassung

Kontext. Die Architecture Tradeoff Analysis Method (ATAM) ist ein etabliertes Verfahren zur Bewertung von Softwarearchitekturen. Sie dient insbesondere der Identifikation von Trade-offs, Risiken und Sensitivitätspunkten im Zusammenhang mit Qualitätsanforderungen und bildet somit eine zentrale Grundlage für qualitative Architekturbewertungen.

Problem. Trotz ihrer Wirksamkeit ist die klassische Durchführung von ATAM stark manuell geprägt und mit erheblichem Zeit- und Ressourcenaufwand verbunden. Der Prozess erfordert umfangreiche Beteiligung von Architekt:innen, Stakeholdern und Domänenexpert:innen, was insbesondere in frühen Entwurfsphasen ineffizient ist.

Zielsetzung. Mit dem Aufkommen von Künstlicher Intelligenz und Large Language Models (LLMs) eröffnen sich neue Möglichkeiten zur (Teil-)Automatisierung wissensintensiver Prozesse wie ATAM. In dieser Arbeit wird ein semi-automatisches Werkzeug auf Basis von Retrieval-Augmented Generation (RAG) vorgestellt, das strukturierte Eingaben verarbeitet und relevante Architekturkenntnisse zur Identifikation von Risiken, Trade-offs und Sensitivitätspunkten generiert.

Methodik. Aufbauend auf einer Literaturrecherche zu bestehenden Ansätzen zur (teil-)automatisierten Architekturbewertung wurde ein Prototyp entwickelt, der ATAM mit RAG kombiniert. Die Generierung erfolgt auf Basis von gezieltem Prompt-Design und der Einbindung kuratierter Architekturdokumente. Der Prototyp wurde durch einen Vergleich zwischen manueller und automatisierter Analyse sowie durch eine begleitende Nutzerstudie evaluiert.

Ergebnisse. Die Analysezeit konnte im Vergleich zum manuellen Vorgehen um etwa 90% reduziert werden. Viele der automatisch erzeugten Risiken, Trade-offs und Sensitivitätspunkte wurden von Expert:innen als relevant eingeschätzt. Gleichzeitig zeigte sich, dass das Werkzeug Schwierigkeiten hatte, tief kontextbezogene Aspekte zu erkennen, und teilweise redundante oder generische Inhalte erzeugte. Die Nutzerstudie unterstrich die Vorteile in Reaktionsgeschwindigkeit und Bedienbarkeit, verwies aber auch auf Herausforderungen bei der Eingabeformatierung und Ergebnisinterpretation.

Fazit. Der RAG-unterstützte ATAM-Prototyp zeigt Potenzial zur Effizienzsteigerung in der Architekturanalyse und zur (Teil-)Automatisierung der Risikobewertung. Eine vollständige Ablösung menschlicher Expertise ist jedoch nicht möglich. Einschränkungen bestehen in der Abhängigkeit von Prompt-Qualität, der fehlenden Fähigkeit zur Verarbeitung visueller Architekturmodelle und in begrenzter Ergebnistiefe. Zukünftige Arbeiten sollten sich auf die Verbesserung der Promptgestaltung, die Integration visueller Modelle sowie auf erklärende Ausgaben zur Steigerung der Robustheit und Praxistauglichkeit konzentrieren.

Contents

List of Figures

List of Tables

Acronyms

AADL Architecture Analysis and Design Language. 14

AI Artificial Intelligence. 1

ATAM Architecture Tradeoff Analysis Method. ix

DSL Domain Specific Language. 23

LLM Large Language Model. 1

LQN Layered Queueing Network. 15

LQNS Layered queueing network solver and simulator user manual. 15

OCL Object Constraint Language. 16

PCM Palladio Component Model. 14

RAG Retrieval Augmented Generation. 1

SDM System Description Model. 16

SE Software Engineering. 18

SSM Security Specification Model. 16

UML Unified Modeling Language. 15

1 Introduction

The design and evaluation of software architectures in the early stages of development are crucial [LBHB99] to ensure that software systems meet key quality attributes such as performance, modifiability, security and reliability [BJ00]. A widely used qualitative approach to architectural evaluation is the Architecture Tradeoff Analysis Method (ATAM) by Kazman et al. [KKB+98]. ATAM is a structured, scenario-based method for identifying trade-offs and sensitivity points in a software architecture. It involves systematic collaboration between stakeholders, architects and domain experts to evaluate architectural decisions against predefined quality requirements [KKB+98].

While the ATAM provides valuable qualitative insights [KKB+98], it is complemented by quantitative modeling approaches such as Palladio [BKR09] and other simulation-based techniques. Qualitative methods enable architects to analyze multiple architectural variants from different perspectives and evaluate their impact on quality attributes [KBAW94; KKB+98]. In contrast, quantitative frameworks like Palladio offer detailed performance predictions [BKR09], whereas ATAM focuses on identifying risks, trade-offs, and sensitivity points using a scenario-based approach without relying on empirical data [KKB+98]. This is particularly relevant during design-time analysis, as decisions made at this stage significantly impact the system's long-term evolution [TOW17]. Therefore, a combination of qualitative (e.g., ATAM) and quantitative (e.g., performance simulations) evaluation techniques is frequently employed to achieve a comprehensive architectural assessment [AG04]. Especially in the early stages of development, qualitative methods like ATAM can provide valuable insights into the architecture's design without necessitating detailed models or empirical data [KKB+98].

Despite its advantages, the traditional ATAM process is labor-intensive, requiring significant manual effort and expert knowledge [YWL+24]. Recent advancements in Artificial Intelligence (AI), especially Large Language Model (LLM)s such as GPT [Sha24], DeepSeek [GYZ+25], and Meta's LLaMA [DJP+24], offer new possibilities for automating and enhancing architectural analysis. LLMs have demonstrated strong capabilities in text comprehension, summarization, and structured dialogue [BMR+20; ZSG+20], making them promising tools for streamlining the ATAM process. They can analyze natural language descriptions of architectural decisions and correlate them with best practices, known anti-patterns, and potential risks based on existing knowledge [ATO25]. This automation can reduce manual effort, improve consistency, and accelerate evaluations in the ATAM workflow.

However, a key limitation of pre-trained LLMs is that their knowledge is static, based on pre-existing training data, and may become outdated [BGMM21]. To address this, we propose integrating Retrieval Augmented Generation (RAG) into the ATAM process. RAG, initially introduced by Lewis et al. [LPP+20], combines information retrieval with LLMs, allowing access to up-to-date and relevant documents before generating responses. Addressing ATAM, this approach could potentially enhance the time efficiency and relevance of architectural evaluations by incorporating the latest knowledge rather than relying solely on pre-trained information.

During the thesis, we developed a prototype, called the "RAG-ATAM-Tool" that automates a part of ATAM by performing a qualitative analysis, focusing on identifying trade-offs, sensitivity points, and risks in architectural decisions of an architecture layout based on predefined quality attribute scenarios. To evaluate the tool, we focused on the following research questions:

RQ1 How does the use of Retrieval Augmented Generation in ATAM compare to manual analysis regarding the relevance of the results and time efficiency?

RQ2 How well does the RAG ATAM Tool support the analysis process in terms of usability?

We evaluated the overall performance of the RAG enhanced ATAM tool by comparing the relevance of the points analysed and the time efficiency of the automated approach with the traditional manual method. Specifically, we had both approaches analyse the same model problem and compared the results. In addition, we evaluated the usability of the tool through a task-based evaluation where participants used the tool to analyse the risks, sensitivity points and trade-offs of a given architectural decision and quality attribute scenario.

Thesis Structure

The bachelor thesis is structured as follows:

Chapter 2 – Foundations: This chapter provides an overview of the background and foundations of the thesis.

Chapter 3 – Related Work: We present related work in the field of (semi-) automated architectural analysis and Large Language Models.

Chapter 4 – Requirements Engineering: We present the requirements for the semi-automated ATAM with RAG.

Chapter 5 – Concept for Semi-Automated ATAM with RAG: This chapter describes the concept of the semi-automated ATAM with RAG.

Chapter 6 – Implementation: This chapter describes the implementation of the prototype.

Chapter 7 – Evaluation: This chapter explains the evaluation of the RAG enhanced ATAM and interprets its results.

Chapter 8 – Conclusion: We conclude our thesis and provide an outlook on future work.

2 Foundations

In this chapter we will introduce the basic concepts that are essential for understanding the foundations of the ATAM and RAG methodologies that form the basis of our approach. The foundations section will cover methodologies and concepts such as the *Architecture Tradeoff Analysis Method (ATAM)*, *Prompting Strategies*, and *Retrieval-Augmented Generation (RAG)*, which form the basis of our approach.

2.1 Architecture Tradeoff Analysis Method (ATAM)

ATAM [KKB+98] is a structured and systematic approach to evaluating the architectural design of software systems. It provides a framework for analysing the potential impact of architectural decisions on system quality attributes such as performance, modifiability, security and reliability. ATAM helps to identify risks, sensitivity points and trade-offs early in the development phase to ensure that the architecture meets the project's goals and requirements.

The method is based on the premise that the software architecture plays a critical role in influencing the system to meet its requirements [BCK03]. By systematically evaluating architectural approaches against elicited scenarios representing the quality attributes of the system, stakeholders can identify potential conflicts and problems in the architecture. This enables them to make informed decisions about the architecture, prioritise requirements and ensure that the system meets its quality objectives.

The primary objectives of ATAM include:

- **Risk Identification** by analysing how certain architectural decisions may negatively impact the system's quality attributes. Risks are architectural choices that may adversely affect the ability of the system to satisfy the quality attributes.
- **Trade-off identification** between competing quality attributes and architectural decisions, highlighting decisions that may affect multiple attributes. Trade-offs are decisions where improving one quality attribute may degrade another.
- **Identifying Sensitivity Points** to identify architectural decisions that have a significant impact on quality attributes. Sensitivity points are architectural decisions that have a high impact on the achievement of certain quality attributes.
- **Refining the Architecture** by providing insights and recommendations to reduce risks to stakeholders.
- **Improve communication between stakeholders** through a structured and systematic approach to architectural evaluation.

By addressing these objectives, ATAM enables teams to proactively mitigate risk in the early stages of development, prioritise requirements and make informed decisions about the architecture [CKK01]. In our context, we focus on automating the analysis phase of ATAM, which focuses on identifying trade-offs, risks and sensitivity points of a given architecture candidate against predefined quality scenarios representing quality attributes. Figure 2.1 illustrates the ATAM process overview, highlighting the analysis phase to be automated in this thesis.

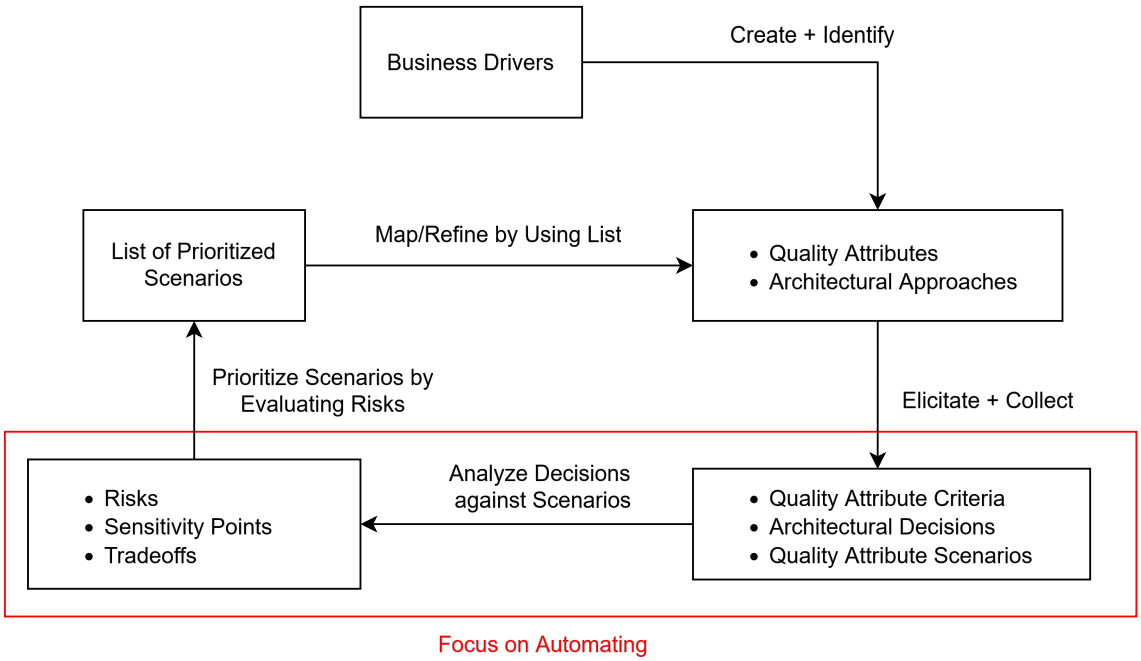

Figure 2.1: ATAM Overview [KKB+98].

ATAM follows a structured process divided into four key phases with multiple steps, as Figure 2.1 shows [KKB+98]:

1. **Presentation Phase (Steps 1-3):** Prior to the ATAM workshop, the architecture team defines the system goals, quality attribute requirements and identifies relevant stakeholders. Initial documentation of different architectural approaches and potential scenarios is prepared. Presentations typically include overviews of system goals, assumptions and constraints to ensure that all participants have a common understanding of the system context. This phase ensures stakeholder alignment and sets the stage for subsequent analysis.
2. **Investigation and Analysis Phase (Steps 4-6):** One of the most critical phases of ATAM is the Exploration and Analysis phase, where stakeholders systematically analyse architectural choices against quality attribute scenarios to identify risks, sensitivities and trade-offs. This phase involves the generation of a quality attribute utility tree, as illustrated in Figure 2.3, which breaks down and prioritises quality attributes (such as performance, availability and modifiability) into concrete scenarios elicited by stakeholders. For example, a scenario might ask: "How does the system handle a sudden increase of 1000 concurrent users?" if performance is a critical requirement. After defining the quality attribute scenarios, which represent concrete situations describing how the system should behave under specific

conditions to fulfill the requirements of the quality attribute, stakeholders map them to architectural appraoches and their corresponding architectural decision. Having defined the quality attribute scenarios, which are concrete situations describing how the system should behave under certain conditions to meet the requirements of the quality attribute, the stakeholders map them to architectural approaches and their corresponding architectural choices. An architectural approach represents a general strategy for structuring a system, such as microservices, layered architectures or event-driven architectures. An architectural decision refers to a design choice made within that approach, such as choosing asynchronous message queues for communication between microservices to improve scalability.

With this mapping in place, the next step is to evaluate the impact of these decisions using quality attribute criteria. These should provide measurable benchmarks for assessing the system's performance, security, and other quality attributes. An example of a quality attribute criterion might be: "The system should deliver video content in real-time to users with a latency of less than 200 milliseconds."

At this point in the process, the analysis team evaluates how different architectural approaches affect these scenarios, documenting risks, sensitivity points, and trade-offs present in architectural decisions. Each scenario is then documented with an assessment of the risks, tradeoffs, and sensitivity points present in architectural decisions, as exemplified in Table 2.1. That means, for each scenario, the team identifies the architectural decisions that impact the quality attribute response, the risks associated with these decisions, the sensitivity points where decisions have a significant impact, and the trade-offs between different decisions. For example if we have 3 scenarios, 2 architectural approaches with each 3 architectural decisions, we will have 18 potential entries with risks, sensitivity points, and tradeoffs in the table. As mentioned before, the thesis will focus on automating this phase, where the quality attribute scenarios will be analyzed against the current architecture. To fully conduct the automation of the analysis, we need scenarios representing a quality attribute, architectural approaches with a description and their decisions, and a list of quality attribute criteria, as shown in Figure 2.2.

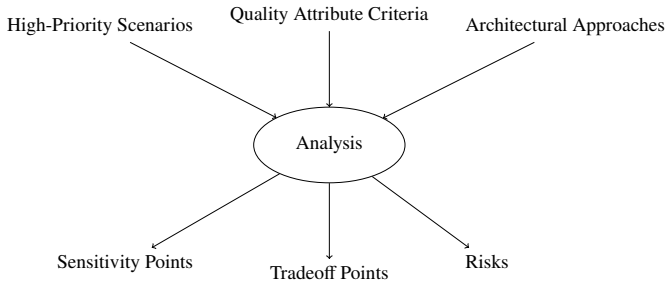

Figure 2.2: Analysis Phase of ATAM visualized [KKB+98].

3. **Testing and Evaluation Phase (Steps 7-8):** After identifying and validating outputs from the previous phase, stakeholders expand the set of scenarios based on the utility tree and prioritize them through a structured voting process. This phase helps the team to align the stakeholders about the most important quality attributes scenarios. Also, the architecture team will use this to refine their architectural approaches and decisions. After that, these prioritized scenarios then serve as test cases for another analysis of the architectural

approaches, uncovering additional risks, sensitivity points, and trade-offs which potentially negatively impact the quality attributes of the prioritized scenarios. This procedure will be repeated until the architecture team is confident that the architecture aligns with the project's goals and requirements.

4. **Present Results Phase (Step 9):** The ATAM team compiles and presents their findings, including architectural styles, scenarios, attribute-specific questions, the utility tree, identified risks, sensitivity points, and trade-offs, to stakeholders. A detailed report may be produced, outlining proposed mitigation strategies and recommendations. Final presentations emphasize transparency and clarity, enabling stakeholders to make informed decisions based on the documented results and analyses.

ATAM as a method is highly collaborative and iterative with many benefits for evaluating and improving software architectures [CKK01]. One of its primary advantages is the early identification of risks. By systematically analyzing architectural decisions, ATAM helps uncover potential weaknesses and vulnerabilities in the architecture before implementation, saving resources and time in the long run [KKB+98].

Another significant benefit of ATAM is the ability to identify tradeoffs between competing quality attributes [KKB+98]. Many decisions in software architecture are not entirely straightforward and may involve tradeoffs in different aspects. By explicitly recognizing and analyzing these tradeoffs, stakeholders can make informed decisions that balance priorities according to the system's objectives [KKB+98].

Regarding quality attributes, ATAM clarifies quality attribute requirements by eliciting and prioritizing these requirements in detail, ensuring that the architecture is evaluated not only for functional requirements but also for non-functional aspects that are critical to long-term success [KKB+98]. Another important outcome of ATAM is the creation of a documented basis for architectural decisions. The method records the rationale behind design choices, tradeoffs, and priorities, providing stakeholders with clear justifications for decisions. This documentation supports ongoing architectural evolution and facilitates knowledge transfer to new team members.

Ultimately, the most important result of ATAM is the development of improved architectures. The method aids in eliciting sets of quality requirements across multiple dimensions, analyzing their effects individually, and understanding the interactions between them. This should ensure that the final system meets both business and technical requirements effectively [KKB+98].

<table>
<tr><td colspan="4">Scenario: <a scenario from the utility tree or from scenario brainstorming></td></tr>
<tr><td colspan="4">Attribute: <performance, security, availability, etc.></td></tr>
<tr><td colspan="4">Environment: <relevant assumptions about the environment in which the system resides ></td></tr>
<tr><td colspan="4">Stimulus: <a precise statement of the quality attribute stimulus (e.g., failure, threat, modification, ...) embodied by the scenario></td></tr>
<tr><td colspan="4">Response: <a precise statement of the quality attribute response (e.g., response time, measure of difficulty of modification)></td></tr>
<tr><td colspan="4">Architectural Decisions</td></tr>
<tr><td>Decisions</td><td>Risk</td><td>Sensitivity</td><td>Tradeoff</td></tr>
<tr><td><list of decisions affecting quality attribute response></td><td><risk #></td><td><sens. point #></td><td><tradeoff #></td></tr>
<tr><td colspan="4">Reasoning:</td></tr>
<tr><td colspan="4"><qualitative and/or quantitative rationale for why the list of architectural decisions contribute to meeting the quality attribute response requirement></td></tr>
<tr><td colspan="4">Architecture diagram:</td></tr>
<tr><td colspan="4"><diagram or diagrams of architectural views annotated with architectural information to support the above reasoning. These may be accompanied by explanatory text.></td></tr>
</table>

Table 2.1: Architectural Approach Documentation Template [KKB+98].

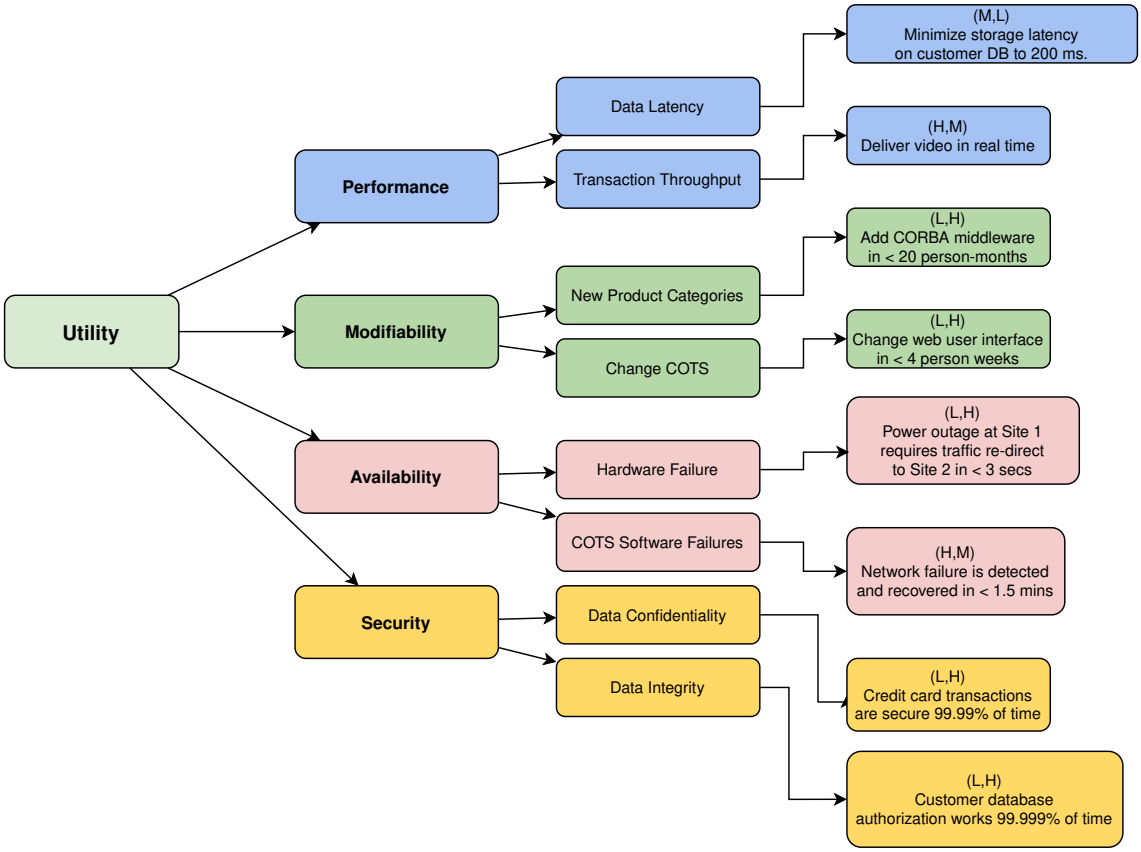

Figure 2.3: Example Utility Tree for Quality Attribute Scenarios with Priority Ratings [KKB+98].

2.2 Retrieval-Augmented Generation (RAG)

Enhancing the quality of outputs from LLMs is crucial for ensuring accuracy and reliability in natural language processing tasks [PSX+24]. One effective technique to achieve this is RAG, first introduced by Lewis et al. [LLP+20]. According to Lewis et al., RAG combines a retrieval-based approach with LLMs to enhance output accuracy and contextual relevance. It retrieves relevant documents or data from a predefined database to provide external information to the model, which can be used to generate more accurate and relevant responses. This is particularly useful for tasks requiring factual consistency, domain-specific knowledge, and up-to-date information [Pal23]. It operates by integrating a dual-component system: the *Retriever* and the *Generator* according to Lewis et al. [LLP+20]. The Retriever is responsible for identifying and retrieving relevant information from a predefined knowledge base or document corpus, often containing domain-specific and current knowledge. Using sophisticated search algorithms, such as BM25 [RZ09], cosine similarity [GSB18], or dense retrieval [FZA+24], the retriever can fetch information that directly correlates with the query. The Generator, on the other hand, is responsible for generating responses based on the retrieved information and the initial prompt [LLP+20].

Implementing RAG in the context of ATAM can be practical, as it has the potential to enhance the model's performance by providing it with domain-specific information [Swa23]. In the case ATAM, we map the domain-specific knowledge with architectural knowledge with architectural knowledge about different architectural approaches, their associated risks, sensitivity points, and trade-offs. Additionally, it brings advantages specific to the LLM workflow and output quality, such as:

- **Hybrid Memory Consistency:** RAG combines parametric memory (pre-trained LLM) with non-parametric memory (external knowledge base) to provide a more comprehensive and context-aware response. This feature falls into our hands since the responses of the model should mostly rely on factual and up-to-date information about architectural approaches. Finally, the model can access multiple documents and sources to reduce the risk of outdated or incorrect information [LLP+20].
- **Knowledge-Intensive Tasks:** It is particularly useful for tasks requiring factual consistency, domain-specific knowledge, and up-to-date information, due to the reliability of external knowledge sources [LLP+20].
- **Reduction in Hallucinations:** By grounding responses in retrieved factual data, RAG significantly minimizes the risk of generating inaccurate or fabricated information [ZYW+24].
- **Prohabilistic Marginalization:** RAG supports generating responses by marginalizing across multiple retrieved documents, ensuring outputs are robust even when exact matches are not found [LLP+20].
- **Dynamic Knowledge Updating:** RAG allows hot-swapping of document indexed to update the knowledge base of the LLM without retraining the model, enabling adaptability to new information [LLP+20].
- **Improved Interpretability and Controlability:** Responses generated by RAG can be traced back to specific retrieved documents, enhancing both transparency and trustworthiness [ZYW+24].

As the example in Figure 2.4 shows, the RAG process consists of multiple steps. It begins with a user query that serves as the input. According to Lewis et al. [LLP+20], the query is first passed through an indexing phase, where external documents are pre-processed into smaller manageable chunks and then transformed into vector embeddings using an embedding model. These embeddings capture the semantic meaning of the text and are stored in a database, enabling efficient search and retrieval. Following the indexing phase, the retriever component uses the query to search the database for relevant documents. During that phase, the query is also transformed into vector embeddings [LLP+20]. Its embedding is then compared to the embeddings of the documents in the database to identify the most relevant ones, using a sophisticated search algorithm like Cosine Similarity [GSB18] or BM25 [RZ09]. Depending on the amount of external information needed, the algorithm then retrieves the top-k documents that are most similar to the query [LLP+20]. The retrieved documents are then passed to the generator component, which uses the information to generate a response to the user query [LLP+20]. The generation phase then combines the user query with the retrieved documents. Using structured prompts as mentioned in Section 2.3, the LLM synthesizes a response based on both the query and retrieved documents, ensuring that the output is contextually relevant and factually accurate. The prompt explicitly instructs the model to consider the information provided in the chunks when forming its response.

Finally, the output phase demonstrates the added value of RAG. Without RAG, the LLM only relies on its pre-trained knowledge, often resulting in generic and uninformed responses, possibly having bias and hallucinations [LBBH25]. In contrast, with RAG, the LLM produces a detailed and context-aware response, that is grounded in factual information from the retrieved documents, enhancing the accuracy and relevance of the generated content, reflecting a deeper understanding of the query [BBK+24].

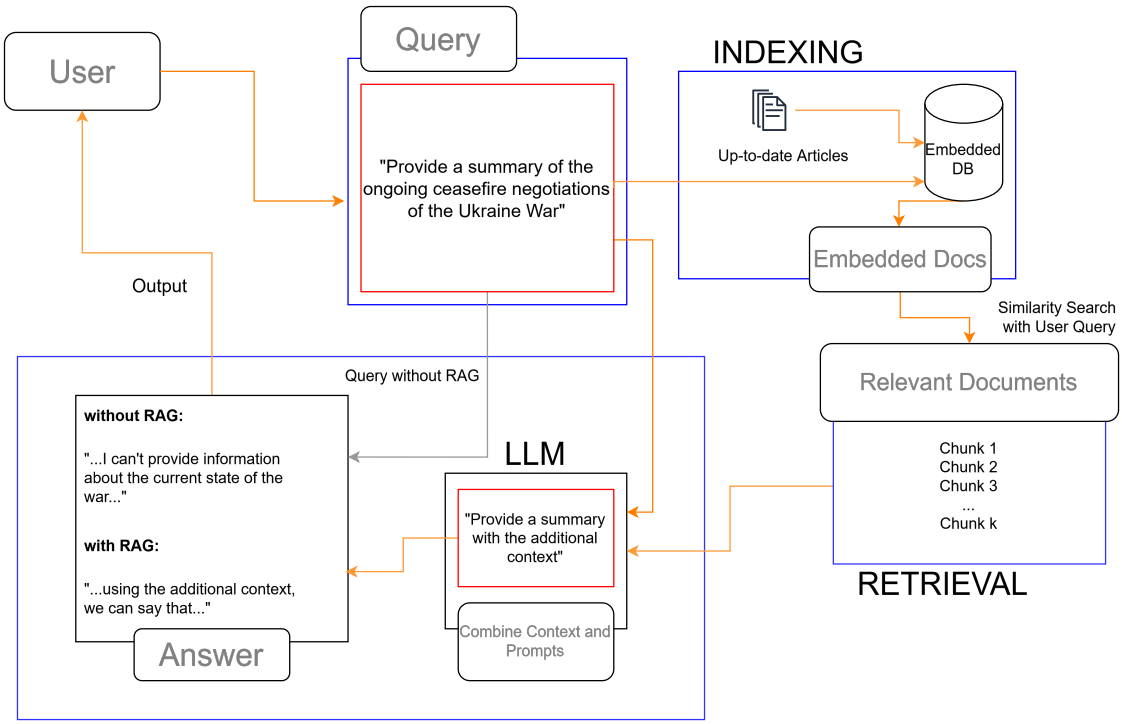

Figure 2.4: Overview of Integrating RAG into the LLM Workflow [GXG+24].

2.3 Prompting Strategies

The quality of text generated by LLMs depends on several factors, including the quality of the training data and the underlying model architecture [LZL+23]. However, one of the most influential factors in achieving high-quality outputs is Prompt Engineering. Crafting effective prompting patterns can significantly influence the quality of the model's output, guiding it to generate more accurate, homogeneous, and contextually appropriate responses [AWA24].

In the context of RAG, structured and consistent prompts are essential to ensure the generation of reliable and well-formatted outputs. These outputs must not only meet the requirements of our analysis but also be easily processable by other systems and programs [MSM+24].

This becomes particularly important in zero-shot learning scenarios, where the model relies solely on the information provided in the prompt to generate responses without prior examples [Li23]. In such cases, it is critical to deliver clear context, precise instructions, and relevant details to the model to maximize performance and relevance. White et al. [WFH+23] provides a comprehensive catalog of prompt patterns that can be utilized to enhance the efficiency and effectiveness of LLMs,

as shown in Table 2.4. As mentioned before, we want to have uniform outputs, which can be easily processed by other systems. For zero-shot learning tools, we need to want to customize the output to a certain degree, which can be achieved by using the *Template* pattern from the *Output Customization* [WFH+23]. According to White et al., template prompting is a structured approach to designing prompts that guide the model to generate responses in a specific format or layout. By providing a template with placeholders for the model to fill in, we can ensure that the generated outputs are consistent and follow a predefined structure. Table 2.2 shows an example of how a template prompt can be used to guide the model to generate responses in a specific format. As for the format, we can choose existing formats like JSON, XML, or YAML, or create a new format that fits our needs. This can be helpful for handling the outputs of LLMs in a structured way, which can be easily processed by other systems and programs.

Contextual Statements
Act as persona X
Provide outputs that persona X would create

Table 2.2: Persona Prompting Contextual Statements [WFH+23].

When conducting complex analysis tasks (e.g.,ATAM), it is obvious that the amount of provided input is cast and complex. Too much information can lead to loss of focus and confusion for the model, especially for LLMs with small context windows and limited parameters [PTZ+24]. For that reason, Bsharat et al. suggest breaking down complex tasks into smaller, more manageable subtasks, which can be processed more effectively by the model [BMS24]. For example, we want to analyze customer feedback data to identify key pain points. Rather than feeding the entire dataset into the model at once, the task can be broken down into subtasks such as categorizing feedback by topic (e.g., pricing, usability, support), summarizing sentiment within each category, and extracting representative quotes or recurring issues. By breaking down the task into smaller, more focused subtasks, we can guide the model to generate more accurate and relevant responses, improving the overall quality of the analysis. This can also be applied to other analysis tasks, such as ATAM.

Besides using the *Template* and *Chunking* pattern, we can also use the *Persona* pattern to let the model generate responses from a specific perspective or role [WFH+23]. This can be useful when we want the model to act like a domain expert. Table 2.3 shows an example of how the persona pattern can be used to guide the model to generate responses from a specific perspective. In this case, our model should act like a software architect which should have domain knowledge about different patterns and their pros and cons. It should then provide output that a domain expert would provide when analyzing the architecture against a specific scenario.

Contextual Statements
I am going to provide a template for your output
X is my placeholder for content
Try to fit the output into one or more of the placeholders that I list
Please preserve the formatting and overall template that I provide
This is the template: PATTERN with PLACEHOLDERS

Table 2.3: Template Prompting Contextual Statements [WFH+23].

Aside from prompt patterns, we also want to consider the text structure and content of the prompts, since the organization and flow of information within the prompt can greatly influence the model's ability to comprehend and process the request. For that, Bsharat et al. proposed some principles that can be helpful when writing prompts [BMS24]. As suggested for *Prompt Structure and Clarity*, we used *"<(section)>"* and *"</(section)>"* to mark the beginning and end of a specific context like *"instruction"*, *"template"*, or *"architecture input"* [BMS24]. Also, we want to consider the following points to improve the specialty and information quality of the prompt according to Bsharat et al. [BMS24]:

- Employ affirmative directives such as "do" while steering clear of negative language like "don't".
- Use Delimiters
- If you prefer more concise answers, no need to be polite with LLMs
- Clearly state the requirements that the model must follow to produce content, in the form of keywords, regulations, hints, or instructions
- Repeat a specific word or phrase multiple times within a prompt.

Together, these methods create a powerful and systematic approach to prompt engineering, optimizing the performance of LLMs in generating accurate, context-aware, and actionable outputs.

Pattern Category	**Prompt Pattern**
Input Semantics	Meta Language Creation
Output Customization	Output Automater Persona Visualization Generator Recipe Template
Error Identification	Fact Check List Reflection
Prompt Improvement	Question Refinement Alternative Approaches Cognitive Verifier Refusal Breaker
Interaction	Flipped Interaction Game Play Infinite Generation
Context Control	Context Manager

Table 2.4: Prompt Pattern Categories [WFH+23].

3 Related Work

In this chapter, we present an overview of the existing literature on automated software architecture analysis and evaluation, focusing on both quantitative and qualitative approaches. We discuss the limitations of current methods and tools, highlighting the need for a more comprehensive and automated solution that integrates architectural trade-off analysis and risk identification.

3.1 Literature Research Methodology

The literature research for this thesis was conducted using a combination of search engines and academic databases to identify relevant papers and articles related to the topics of Large Language Models (LLMs), Architectural Trade-off Analysis Method (ATAM), and Retrieval-Augmented Generation (RAG). The primary search engine and database used was Google Scholar[1], as it provides a comprehensive collection of academic papers, articles, and publications from various disciplines. For a general search, terms such as ATAM and RAG were used to identify papers and articles related to the topics. After the general search, the snowball method was used with the help of the *Research Rabbit App* [2] to find relevant papers and articles. The app provides an interface that provides a list of papers and articles based on the search query and suggests related articles based on the references of the papers. For the query, we used the following keywords and their combinations:

- Architecture Tradeoff Analysis Method (ATAM)
- Large Language Models (LLMs)
- Automated Architecture Analysis/Evaluation
- Retrieval-Augmented Generation (RAG)
- Qualitative Software Architecture Analysis
- Quantitative Software Architecture Analysis
- AI for Software Architecture Analysis

To ensure the quality and relevance of the selected literature, the following inclusion and exclusion criteria were applied:

Inclusion Criteria:

- Peer-reviewed journal articles, conference papers, or reputable preprints (e.g., arXiv[3]).

[1] https://scholar.google.de/
[2] https://researchrabbitapp.com
[3] https://arxiv.org/

- Papers that present case studies, frameworks, or evaluations relevant to LLM integration or software quality attributes.
- Publications in English.
- Works that explicitly discuss methodologies such as ATAM, automated architecture evaluation, or the application of AI in software architecture.

Exclusion Criteria:

- Non-academic sources (e.g., blogs).
- Articles not available in full text.
- Publications that do not provide sufficient methodological detail or theoretical grounding.

3.2 Automated Quantitative Frameworks for Software Architecture Analysis

Automated software architecture analysis aims to streamline the evaluation and optimization of software systems by leveraging computational models and search-based techniques. Among the tools in this space are *PerOpteryx* [BFK19] and *ArcheOpterix* [ABGM09], both of which represent highly automated, model-driven approaches focused on quantitative quality attributes and design space exploration.

PerOpteryx is a model-based optimization tool tailored for component-based architectures. It automates the generation and evaluation of architecture candidates by operating on models defined using the Palladio Component Model (PCM) [BKR09], which focuses on component-based architectures, it supports automated design exploration by substituting components, changing allocations, and scaling resources. PCM includes specifications for components, their interactions, and deployment on hardware, as well as annotations for performance and cost metrics. The tool supports the automatic generation of architecture candidates and design exploration by substituting components, changing allocations, and scaling the resources of a pre-existing architecture. The generation of the candidates is identified by the degrees of freedom in the architecture, such as component allocations, server configurations, and hardware selection. Those variables define the potential changes in the architecture that can affect the quality attributes. As far as the algorithm is concerned, it uses the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [DPAM02], to navigate large design spaces and generate potential candidates. PerOpteryx then evaluates the candidates against defined metrics using performance models like *Layered Queueing Networks (LQN)* [FAW+08]. This approach predicts key performance indicators like response times and resource utilization. The tool then generates Pareto-optimal candidates, which are presented to the software architects for further analysis and decision-making. This approach allows software architects to balance trade-offs and make informed design decisions based on the generated candidates.

ArcheOpterix, on the other hand, is an Eclipse plug-in tool focused on optimizing embedded system architectures. It uses the *Architecture Analysis and Design Language (AADL)* [ABGM09] to model system architectures and evaluate them against quality attributes. The tool consists of several modules, including the *AADL Model Parser*, which parses the AADL model and extracts relevant

information, the *Architecture Analysis Module (AAM)*, which evaluates the architecture against quality attributes using evolutionary algorithms such as Pareto fronts and weighted sums, and the *Optimisation Module*, which generates optimized architectures based on the evaluation results.

Despite these advancements in automated software analysis, they fall short in addressing qualitative aspects of software architectures required for risk analysis and trade-off identification. PerOpteryx primarily focuses on performance metrics and uses pre-defined architectural models, which makes it less suitable for early-stage design exploration and qualitative reasoning. ArcheOpterix, on the other hand, is limited to embedded systems and lacks support for broader software architectures and quality attributes. It also primarily emphasizes optimization rather than comprehensive risk assessment, which is also central to ATAM.

Another notable framework for quantitative software architecture analysis is the *rule-based approach* proposed by Avritzer et al. [Xu08]. This automated framework diagnoses performance bottlenecks and recommends improvements through both configuration and design changes, using performance models derived from Unified Modeling Language (UML) diagrams.

It distinguishes between long path problems, where critical execution paths lead to significant delays, and bottleneck problems caused by resource saturation. The mentioned issues often reinforce each other, as resource contention can extend execution paths, and long paths may increase the load on shared resources. To address these challenges, the framework applies techniques such as ShrinkExec, Batching, Caching/Prefetching, parallelism, and asynchronous processing. These help reduce resource holding times, optimize execution flow, and enhance overall system throughput.

Key components include an Inference Engine that applies predefined rules using the Jess Rule Engine [Fri03], and a Performance Model that translates UML diagrams into Layered Queueing Network (LQN) [FAW+08] for simulation-based analysis. Role Modules handle diagnostics, model transformation, mitigation planning, and logging. Performance evaluation is conducted via the Layered queueing network solver and simulator user manual (LQNS) [FMW+05], with results expressed in metrics like response time and resource utilization. The framework follows an iterative process alternating between runtime configuration tuning and architectural design changes. Design alternatives are assessed and ranked using metrics such as Design Change Efficiency (DCE) and Cost Efficiency (CE).

While effective for diagnosing performance issues, the framework focuses strictly on quantitative analysis and requires extensive upfront modeling. It lacks support for qualitative assessments, such as identifying architectural risks, sensitivity points, or trade-offs [Xu08], limiting its applicability in early-stage design when only high-level models and stakeholder input are available.

3.3 Qualitative Assessment Techniques in Software Architecture

While tools like *PerOpteryx* [BFK19] and *ArcheOpterix* [ABGM09] focus on optimizing software architectures by exploring new candidates, other techniques like scenario-driven risk identification [AGI13] and automated software architecture assessment using Bayesian Belief Networks [NF96] have been developed to identify performance bottlenecks and relationships between quality attributes.

The paper by Bosch [GB00] introduces *SAABNet (Software Architecture Assessment Belief Network)*, a framework that supports early architecture assessment using *Bayesian Belief Networks (BBNs)* [NF96]. This approach addresses the lack of quantifiable data in early development by leveraging qualitative domain knowledge by modeling architectural characteristics and their impact on quality attributes (e.g., performance, maintainability) as nodes in a directed acyclic graph, with conditional dependencies between them. Bayesian inference is then used to propagate new evidence through the network and update probability distributions.

While SAABNet provides a structured, probabilistic means of modeling architectural impacts, it lacks the interactive, scenario-driven nature of ATAM, which relies on stakeholder input and the exploration of trade-offs. Furthermore, SAABNet depends on predefined relationships between architectural elements, making it less adaptable to evolving or context-specific requirements. In contrast, a semi-automated approach combining ATAM with RAG could offer a more dynamic, flexible, and context-aware solution—retrieving relevant knowledge on demand and supporting both qualitative and quantitative aspects of architecture evaluation.

An additional approach to automated software analysis with a focus on risk identification is presented by Almory et al. [AGI13]. They propose a formalized, signature-based security risk analysis method using the Object Constraint Language (OCL) [CG12], enabling automated and flexible rule-based detection of vulnerabilities. The method separates system concerns into a System Description Model (SDM) [RHJ21] and a Security Specification Model (SSM) [RHJ21], allowing for targeted risk analysis while preserving modularity. It leverages the *STRIDE* [RHJ21] model and CAPEC database [NAH+01], alongside tools like the EOP Card Game [DKS13], to map common attack patterns (e.g., injection, man-in-the-middle, DoS) to system entities. This supports structured scenario-based risk identification and mitigation strategies. Security signatures, formalized with OCL, define detection rules for risks such as insecure communication or missing input validation across architecture and design levels.

The methodology includes an OCL-based System Architecture Analyzer [AGI13], which integrates metrics like attack surface and defense-in-depth, providing visual trade-off analysis (e.g., radar charts). Despite that, while this approach aligns with ATAM's scenario-driven risk identification, it is limited to security risks, lacking coverage of other quality attributes, highlighting the need for a more comprehensive, multi-attribute evaluation framework.

Another notable qualitative assessment technique is the *Architecture-Level Modifiability Analysis (ALMA)* method, proposed by Bengtsson et al. [BLBV04]. ALMA is a scenario-based approach aimed at evaluating the modifiability of software architectures during the early design phases. It consists of five structured steps: *Goal Selection*, where the purpose of the analysis is defined (e.g., maintenance effort prediction or architectural comparison), *Architecture Description*, involving documentation of architectural components, interactions, and deployment, *Change Scenario Elicitation*, where expected future changes are gathered, *Scenario Evaluation*, assessing the impact of each change on the architecture, and *Interpretation*, where the results are analyzed to identify architectural risks and modifiability bottlenecks.

A qualitative approach related to ALMA is the *Software Architecture Analysis Method (SAAM)* [KBAW94]. Similarly to ALMA, SAAM is a scenario-based method that assesses software architectures concerning various quality attributes, with an emphasis on modifiability. Like ALMA, it employs usage scenarios to examine how effectively an architecture can accommodate specific changes. By comparing architectural alternatives based on these scenarios, SAAM helps

identify strengths and weaknesses, facilitating informed decision-making. Despite their similarity, the key differences between SAAM and ALMA lie in their focus and scope: while ALMA concentrates on modifiability, SAAM considers a broader range of quality attributes [KBAW94]. ALMA offers detailed, repeatable techniques for each analysis step, enhancing consistency across evaluations [BLBV04]. In contrast, SAAM provides a more general framework, relying on the assessor's expertise [KBAW94]. Last but not least, SAAM does not explicitly distinguish between different analysis goals (e.g., maintenance prediction, risk assessment, architecture comparison), as ALMA does [BLBV04].

ALMA's and SAAM's structured and repeatable process makes it particularly suitable for identifying potential maintenance challenges early on. However, it has a limited scope on modifiability, whereas ATAM [KKB+98] provides a more comprehensive framework for evaluating multiple quality attributes and trade-offs. Also the lack of automated support for architecture analysis and risk identification, which could be addressed by integrating RAG [LPP+20] to enhance contextual understanding and provide on-demand knowledge retrieval.

3.4 Empirical Studies About Using Large Language Models for Assessing and Optimizing UML Models

With the current advancements of LLMs, there is a growing interest in leveraging these models for software engineering tasks, such as assessing and optimizing UML models. While traditional UML model evaluation relies on expert-driven manual assessment or rule-based tools, recent studies have explored the feasibility of using AI-based approaches to automate and enhance this process.

An empirical study of Wang et al. [WWLL24] investigates the capability of ChatGPT [Sha24] in evaluating UML models, including use case diagrams, class diagrams, and sequence diagrams. Their study introduces 11 evaluation criteria that were used to assess the quality of the three UML diagram types. They were then used in an experiment, comparing ChatGPT's grading performance against human experts. The study used UML models created by undergraduate students at Wuhan University as part of their coursework in a Requirements Analysis and Modeling course. These students had undergone a three-month training program before participating in the experiment and submitting their UML models. The dataset consisted of UML models collected from 40 students, with each student submitting one use case diagram, one class diagram, and one sequence diagram, resulting in 120 UML models.

Each UML diagram was then assessed by ChatGPT using a predefined set of 11 evaluation criteria. These criteria focused on structural correctness, completeness, and logical consistency within each diagram type. To establish a ground truth, human experts then independently assessed the same UML diagrams following the same grading criteria. Finally, the results from ChatGPT and the human evaluators were systematically compared to analyze accuracy, consistency, and grading discrepancies. Defined evaluation criteria covered different aspects of UML modeling, for example for use case diagrams, ChatGPT analyzed the correct identification of actors, use cases, and relationships. In class diagrams, it examined the completeness of class identification, attributes, operations, and relationships such as associations and inheritance. For sequence diagrams, the focus was on message sequencing, object interactions, and logical flow.

They found that ChatGPT demonstrates high accuracy (above 82.5%) in evaluating UML diagrams, particularly in assessing use case and sequence diagrams, but struggles with class diagram relationships and complex message identification. While ChatGPT's assessments closely align with expert evaluations, it exhibits overstrictness, occasional misinterpretations, and rigid grading tendencies, highlighting the need for improved contextual understanding and flexibility in AI-driven UML assessment.

Another empirical study by Rouabhia and Hadjadj [RH24] also investigates the capability of ChatGPT [Sha24] in dynamically enhancing UML class diagrams by integrating methods extracted from natural language use case tables. It explores the potential of approaches to automating the enrichment of class diagrams using AI, addressing a gap in existing methodologies that primarily focus on static structural components while overlooking dynamic aspects of system behavior.

The study utilized UML class diagrams created by master's degree students from Chahid Cheikh Laarbi Tebessi University as part of a project on a Waste Recycling Platform. First, the students developed 23 detailed use case tables, structured with information such as actors, descriptions, preconditions, triggers, and main scenarios. Based on these descriptions, they manually created an initial static class diagram containing core system entities, attributes, and basic relationships. However, the diagram lacked detailed methods to represent system behavior.

To enhance the class diagram, they employed an iterative AI-assisted methodology:

1. ChatGPT analyzed the natural language descriptions in the use case tables.
2. Based on extracted functionalities, ChatGPT suggested methods to be incorporated into the class diagram.
3. The class diagram was dynamically modified using PlantUML, integrating the suggested methods while preserving structural consistency.
4. The enhanced diagram was assessed to ensure that the added methods aligned with system requirements.

All criteria focused on the accuracy, efficiency, and completeness of the AI-enhanced diagrams compared to their original versions. The study found that the AI-driven approach significantly improved the accuracy and completeness of the UML class diagrams while reducing the manual effort required [RH24]. The number of methods included in the class diagram increased from 0 to 22, and relationships between entities were refined based on AI-generated insights, indicating that AI-driven enhancements align well with Agile software development practices, facilitating rapid iterations and continuous improvement of UML models, demonstrating that ChatGPT can be a valuable tool for software modeling, automating tedious manual processes while improving the quality and maintainability of UML class diagrams.

In addition to the previously mentioned studies, another study by Speth et al. [SMB24] investigates the capability to comprehend and utilize UML class and sequence diagrams for generating educational exercises in Software Engineering (SE). The study evaluated ChatGPT's performances across different input types, including textual (MermaidDSL[4]) and graphical (draw.io[5], Mermaid-rendered) representations of UML diagrams.

[4] https://mermaid.js.org/
[5] https://app.diagrams.net/

They evaluated ChatGPT's understanding of UML concepts through a structured repeatable experimental process using a mix of prompt-based comprehension checks and multiple input formats. It was conducted using two modeling scenarios across textual and graphical formats, applying structured prompts to test understanding of UML concepts and diagram elements. They systematically recorded whether each feature was correctly, partially, or not understood, and found that text accessibility (e.g., markable vs. non-markable PDFs) significantly influenced comprehension quality. Their findings show that ChatGPT generally performs best with textual input, demonstrating a strong understanding of basic UML concepts like classes, attributes, and operations. However, it struggled with more complex features like compositions, aggregations, and multiplicity characteristics, which frequently led to confusion for ChatGPT. For graphical inputs, comprehension was generally limited by the diagram's structure (e.g., overlapping elements in Mermaid-rendered PNGs or unselectable text in PDFs).

While these studies showcase the potential of LLMs in automating UML model evaluation, optimization, and understanding, they primarily focus on structural correctness and completeness, overlooking broader architectural risk, trade-off, and quality attribute analysis. Moreover, the knowledge of the LLMs used in these studies is limited to the training data, which may not cover the full spectrum of architectural patterns and design principles, making RAG a valuable addition to enhance the contextual understanding and relevance of AI-driven assessments.

4 Requirements Engineering

As mentioned in Chapter 1, the goal of the thesis is to develop a RAG-enhanced prototype that semi-automates ATAM by conducting a qualitative risk analysis of an architecture and investigate the general performance of using this approach. The prototype should focus on identifying trade-offs, sensitivity points, and risks of an architectural decision against a predefined quality scenario representing a quality attribute. In this chapter, we present the requirements of our automated "RAG ATAM tool", focusing on the input data structure, the LLM model, the RAG model, and the user interface.

4.1 Requirements Elicitation

From the vision paper by Eisenreich et al. [ESW24], which proposes a method to semi-automatically generate software architectures using LLMs, we could derive a few aspects about which tools/methods could be used. There was also consultation with the author of the paper to ensure that the interests were aligned.

We also make sure that the practices in the analysis process of ATAM [KKB+98] will be implemented correctly by extracting the relevant processes from the paper. To make the requirements achievable, we evaluated the capabilities and limitations we intend to use, such as RAG and LLMs, to see how realistic the implementation will be. Time and resource constraints also influenced the selection of requirements, since the whole implementation is only limited to six months. We prioritized features critical to demonstrating the prototype's value while deferring less essential functionalities for future development.

4.2 Documentation of Requirements

The requirements for the prototype are derived from the foundational sources mentioned in Chapter 2. Based on the sources above, we established the following requirements for the prototype.

4.2.1 Structure and Documentation of the Input Data for RAG-driven ATAM

The input must be captured and processed in a way so the LLM, which is responsible for the automated part of ATAM, can handle the input correctly. The main part of this requirement will be the uniform structure and format of how the input will be illustrated, especially for input that is originally graphical like the documentation of different architectural views. For the structure, we use the terms and structure based on the ATAM paper by Kazman et al. [KKB+98] and separate the different parts of the input data for clear structure.

Documentation of the System Context

The user input must contain a high-level overview of the system and its purpose and domain. The description covers the presentation phase of ATAM, meaning it provides the overview of the product, such as the system's goals/business drivers, initial documentation, potential scenarios, and constraints. For our analysis tool, we focus on the following points:

System Purpose: This part defines the overarching goals and business drivers of the system, such as improving operational efficiency, enabling scalability, or ensuring compliance with specific standards. It should also outline the primary use cases the system addresses.

Technical Constraints: If already existent, the section should specify, limitations or requirements tied to technology choices, such as programming languages, frameworks, operating systems, or hardware dependencies. Examples include "must support cloud deployment" or "must run on low-power devices."

System Interactions: This part should detail how the system interacts with external entities, such as other systems, third-party services, or users. Key aspects include data flow, integration points, and APIs the system relies on or provides.

Documentation of the Architectural Approaches

The prototype must be able to detail the architectural strategies employed in the system design. It includes parts of the presentation phase and the investigation phase, where different architectural approaches or styles are presented, that could potentially be part of the final architecture. The implementation should include the following points for each approach:

Name of the approach/style: The documentation must include the name of the architectural style. Examples of the approach name are *Microservices*, *Event-Driven Architecture*, *Layered Architecture*, or *Monolithic Architecture*.

Description of the approach: The documentation must explain the key principles and characteristics of the approach. For example:

- *Microservices:* A decentralized architecture where the application is composed of small, independent services, each responsible for a specific function [TG19].
- *Event-Driven Architecture:* A design paradigm where system components communicate through the production and consumption of events, ensuring loose coupling [CB11].
- *Layered Architecture:* A design where the system is divided into layers (e.g., presentation, business logic, and data access) to promote separation of concerns [SM09].

Architectural decisions: This part is essential for the documentation since the ATAM process is about analyzing the risks, trade-offs, and sensitivity points of the architectural decisions against quality attribute scenarios. Therefore, architectural decisions about how the components of the system would be implemented in the specific approach must be detailed. This could include:

- How a specific functionality would be implemented.
- Trade-offs associated with the approach (e.g., performance vs. flexibility).

- How this implementation would align with the quality attribute goals.

Documentation of different architectural views: The documentation must also include the different architectural views of the system. This attribute provides a comprehensive subset of views, inspired by the 4+1 view by Krutchen [Kru95], to represent the architecture from different perspectives. After consultation with the authors of the vision paper [ESW24], we decided to focus on the following views:

- *Development View:* Focuses on the organization of code and development artifacts, such as modules and subsystems. This view allows us to address the system's static organization in the development environment [Kru95].
- *Process View:* Describes the system's dynamic behavior, including how components interact and communicate during execution. This view helps us understand the system's runtime behavior and performance characteristics [Kru95].
- *Physical View:* Represents the deployment of the system on hardware or infrastructure, showing how components are distributed across servers, nodes, or containers. This view helps us understand the system's physical deployment and hardware topology [Kru95].

Since we are working with mostly textual inputs and LLMs, the documentation of the different views must be in a textual format to keep everything uniform. For that, a **Domain Specific Language (DSL)** [DKV00] is required, which is able to describe the views and can be easily processed by the LLM.

Documentation of Quality Attribute Scenarios

A possibility to define a set of well-defined quality attribute scenarios must be included for analyzing the architecture against them since ATAM is a scenario-based method [KKB+98]. Table 2.1 already provides us with a template that can be easily applied for the analysis, since it already has a textual structure. The following points need to be captured for each scenario:

Scenario name: It provides a meaningful name that succinctly describes the purpose of the scenario.

Examples:

- "High Traffic Load Management"
- "Data Security During Transmission"
- "System Recovery After Failure"

Attribute: It specifies the quality attribute being tested in the scenario. A quality attribute is a property or characteristic of a system that determines how well a system meets specified requirements [HW02].

Examples include *performance*, *security*, *availability*, and *modifiability*.

Environment: This part defines the operational context in which the scenario occurs.

It includes details such as:

- System state (e.g., under normal operation, during peak load, or after a failure).
- Hardware or infrastructure conditions (e.g., specific servers, cloud configurations).
- External factors (e.g., network latency, user behavior).

Stimulus: A stimulus describes the triggering event or condition for the scenario.

Examples:

- "A sudden surge in user requests reaching 10,000 requests per second."
- "An unauthorized user attempts to access sensitive data."
- "A server failure occurs in the primary data center."

Response: It mentions the expected behavior or outcome of the system in response to the stimulus.

Examples:

- "The system scales horizontally by adding instances to handle the increased load while maintaining a response time under 2 seconds."
- "The system denies access and logs the unauthorized attempt, notifying the security team."
- "The system seamlessly redirects traffic to a backup server with a recovery time of under 5 minutes."

Documentation of the Quality attribute criteria:

This attribute defines the questions and criteria that define the quality attribute. It marks metrics that the system must meet to satisfy the quality attribute. For example, for the quality attribute of *performance*, the criteria could be the response time of the system under a certain load. The documentation must be in a textual format so that the LLM can process the input.

Structure of Handling Input Data for the RAG Model

The input data must be structured in a way that multiple components, including the LLM, the RAG components, and the user interface of the prototype can handle it. Also, the format must be kept uniform and consistent to avoid errors in data processing and easier handling. Since the documentation is textual, the structure and format must be in a textual format as well.

4.2.2 Generator LLM for Handling the Input Data

Since LLMs are the essential part of the prototype's automated analysis pipeline, their performance and capabilities directly impact the prototype's effectiveness. A performant LLM must meet high standards in multiple dimensions to ensure the effectiveness of the RAG-driven ATAM process.

Degree of Domain-Specific Knowledge

First, the LLM must demonstrate robust domain-specific knowledge in software architecture. This includes familiarity with architectural styles, patterns, and the nuances of trade-off analysis as outlined in the ATAM framework, meaning its training data should include a diverse set of architectural descriptions and quality attribute scenarios. Using that, the LLM can assist in identifying trade-offs, risks, and sensitivity points in complex scenarios. In addition to that, the LLM must be able to understand the DSL that is used to describe the different architectural views. For that, LLMs needs a lot of pre-training on large-scale datasets [XAA24] that are related to the software architecture domain.

Analytical Reasoning Capabilities

Additionally, the LLM must possess advanced analytical reasoning capabilities to evaluate conflicting quality attributes, synthesize information from diverse inputs, and provide actionable recommendations. This requires the ability to parse and comprehend natural language inputs, including system descriptions, and design constraints. To ensure this requirement, we need LLMs with a high level of pre-training with applied optimized pre-training strategies on large-scale datasets [ZDD+24].

Process Large Volumes of Data

Furthermore, the LLM must be able to process large volumes of data. This includes handling complex architectural descriptions, quality attribute scenarios, and trade-off analyses within a reasonable time frame (less than 1 hour), without compromising the quality of its outputs. It should also handle the retrieved chunks from the RAG model in addition to the input data (see Section 2.2). For that, large context windows for capture more tokens simultaneously and a large number of parameters for LLMs are needed to ensure the capture of a huge amount of relevant information while still maintaining the quality of the output [CND+22]. But while looking for performant models we should consider, that the LLM should not be too complex/large, since we have technical constraints that limit the size of the model (see Section 6.2). Using a model that is too large in relation to the hardware can lead to performance issues and long processing times [SKB24].

4.2.3 RAG-Driven Analysis

As mentioned in Section 2.2, the RAG model is used to retrieve relevant information from an external database to support the analysis process by providing additional context and knowledge to the generator LLM [LPP+20]. The RAG model should be able to retrieve relevant information from a database of architectural knowledge, especially for the architectural approaches and their risks, sensitivity points, and trade-offs. Resulting from that, the following points need to be considered:

Database and Indexing:

The RAG model's ability to effectively manage a database of reliable information (in our case architectural knowledge) is critical to its overall performance and reliability [HPP+25]. The following requirements must be addressed for robust database management:

- **Scalability:** The database must be able to dynamically scale and store new data as the database grows. This should also include the other way around, meaning that the database should be able to delete data that is not needed anymore to let the retriever work faster and with more relevant data.
- **Data Quality, Content and Chunking:** The data must contain reliable information related to different architectural approaches and their risks, sensitivity points, and trade-offs. Since the data can be large, it should be chunked into smaller parts for easier retrieval and less confusion for the LLM analysis process [LLH+23].
- **Indexing and Storage Format:** The database must be able to index and store the data before retrieval for a faster and more efficient process.
- **Data Format:** The data should be stored in a structured format that can be easily retrieved and processed by the RAG model.

Retrieval of Information and Prompt Design:

The RAG model must be able to retrieve relevant data from the database and send it to the LLM for further analysis. Since the LLM model is not able to retrieve information from the database itself, we need a mechanism that can do that. The following requirements must be considered for the retrieval process:

- **Retrieval Query/Model:** The RAG model must have a retrieval mechanism that can fetch relevant data from the database based on the input data. This should include a query that includes the most important information that the Retrieval Model needs to know to retrieve the correct chunks of data.
- **Prompt Design for the LLM:** To ensure consistent and accurate analysis output from the LLM, the retrieval component must generate prompts that guide the LLM in processing the retrieved data [PATK25]. That must include a structured design that the LLM can understand and process, as well as clear instructions on how to handle the input data. It should be able to differentiate between different parts of the input data, including context from the database, user input data, and task instruction. For that, Section 2.3 already provides best practices for prompt design. Since ATAM is a structured process with organized report generation and analysis steps, the prompts should be designed to generate the results in the exact structure for every invocation. Last but not least, the context size of the prompts must be limited to a proper size to avoid hallucinations.
- **Clear analysis instructions:** The RAG model must provide clear instructions on how to handle the given user input and the retrieved database data. This includes a description of the situation and the role of the model in the analysis process as an architecture evaluator. Also,

the model must be instructed to find risks, trade-offs, and sensitivity points in the input data with the help of the architectural approaches, quality attribute scenarios, quality attribute criteria, and retrieved data chunks from the database.

Output Format

The RAG model must be able to generate structured output that can be further processed by other components of the tool. The following requirements should be considered for the output format:

Uniform Output Structure: The uniform format must be ensured for further processing of other components. Especially for frontend components we need a format that can be easily handled and displayed for the end user.

Content: As shown in Table 2.1, the output must contain the risks, trade-offs, and sensitivity points of the architectural decisions of an approach against the quality attribute scenarios. Besides that, the output should also view the sources of the retrieved data from the database, so that the user knows where the information comes from.

4.2.4 User Interface and Interaction

To ensure that the tool is user-friendly and accessible, the user interface should contain the following features:

Textual input fields: ATAM is a process that is mostly documented in natural language [KKB+98], so the user must be able to input the data with text fields. In Section 4.2.1, we already defined the input data that the user should provide. For the architectural views, we must provide one text field for each view, where the user can input the diagrams in textual format (like *PlantUML*[1]), since the models we will use are not able to process graphical input.

Overview of the used documents of the database: The user must be able to see which documents are stored in the database and which documents are used for the analysis. There should also be a possibility to add new documents to the database or delete existing documents.

Buttons to upload and run the analysis: The user must be able to upload the input data and run the analysis by interacting with the front end (e.g., by clicking a button). But before that, the system must check if the input data is correct and complete. After submitting the input data, the system should display a loading screen to indicate that the analysis is in progress.

Output field to display the results: The results must be displayed in a structured format that is easy to understand. Since ATAM is a structured process with organized report generation in textual format [KKB+98], the output should be displayed in a tabular format, showing the risks, trade-offs, and sensitivity points with the architectural decisions, architectural approach, and the quality attribute scenarios as the header of each topic. It should be similar to the reporting template provided in Table 2.1. Below the table, the sources of the retrieved data from the database should be displayed to provide transparency and traceability. And since the output can be large, the results should be presented in a collapsable format like toggle lists.

[1] https://plantuml.com/

5 Concept for Semi-Automated ATAM with RAG

This chapter introduces the conceptual design of the proposed RAG-based ATAM prototype used for qualitative analysis of the software architecture designs, creating a bridge between the requirements and the actual implementation. The concept design outlines the system architecture, data flow, and interaction between the components, providing a high-level overview of the tool's functionality and operation. For the general RAG concept, we used the RAG methodology from Section 2.2 and mapped them with the implementation guide from *Langchain*[1]. We then extended the concept with additional functionalities for integrating the user and the ATAM process, focusing on the analysis phase of the ATAM process.

5.1 Overview of the Concept

The concept visualized at Figure 5.1 combines the concept of RAG from Section 2.2 with ATAM elements to create a semi-automated tool for architectural analysis. We categorized the components similarly to the RAG methodology while considering extra components for integrating the user and additional functionalities for the ATAM process.

Five main components need to be considered for implementation: the User Interface, the Backend API, the Retrieval Model, the Indexing and Embedding Model, and the Generation Model (LLM).

The User Interface captures input data from the user, displays the results, and provides an interactive platform for user interaction with the tool. The Backend API acts as an intermediary between the User Interface and other components, such as the database and the retrieval and embedding models. It handles database operations (adding and deleting documents), forwards embedding operations and retrieval queries to the embedding models, and sends fetch requests to the LLM model. The Retrieval Model, which also serves as an embedding model, manages the chunking and embedding of raw documents, retrieves relevant data based on user input, and generates prompts for the LLM model. The LLM for analysis processes prompts from the User and Retrieval Model, analyzes the input data, and generates output results.

Finally, the database stores architectural knowledge and data chunks used by the Retrieval Model for retrieval operations. It consists of two parts:

- **Raw data storage**, which allows users to access the list of documents easily.
- **Vectorized Data Storage**, which is used for retrieval and embedding operations.

[1] https://python.langchain.com/v0.2/docs/tutorials/local_rag/

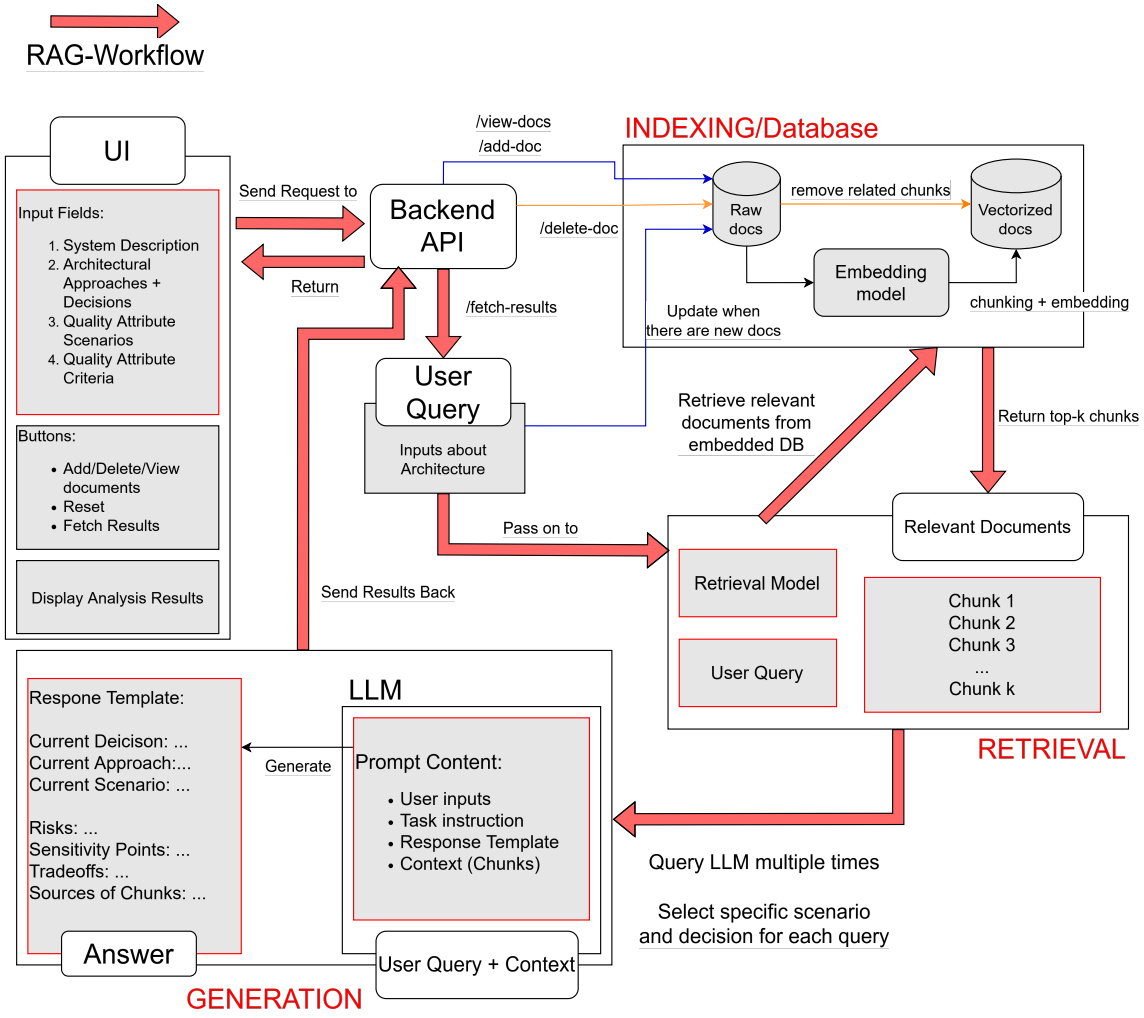

Figure 5.1: System Overview of the Analysis Flow of the RAG-based ATAM Prototype

For the prototype, we emphasize the analysis phase of the ATAM process, focusing on identifying trade-offs, sensitivity points, and risks in architectural decisions based on predefined quality scenarios provided by the user as input data.

For the analysis flow of the system, which is also the main part of the analysis, we divided the process into four main parts, as shown in Figure 5.1. The process starts when the user sends a request through the User Interface, which is forwarded by the Backend API to the Retrieval Model. The Retrieval Model processes the query, checks if relevant database entries exist, retrieves embeddings from the Vector Database, fetches raw documents if necessary, and generates a prompt for the LLM. The LLM then processes the structured prompt, analyzes architectural decisions against quality attributes, and generates a structured JSON response containing insights such as trade-offs, risks, and sensitivity points. Finally, the Backend API sends the results back to the User Interface, where they are displayed to the user.

5.2 Documents for Collecting Domain Specific Knowledge

For adapting the RAG technology to the ATAM process, we need an external source of architectural knowledge to provide the necessary information for the qualitative analysis process. The prevent false information from the documents, we need to ensure that the documents are from a reliable source and contain the necessary information for the analysis process. Therefore, we propose using scientific papers and literature on software architecture, such as books, articles, and research papers, as the primary source of architectural knowledge. The reliability of the documents is ensured by selecting reputable sources, such as academic institutions, research organizations, and industry experts, to provide accurate and up-to-date information on software architecture.

By using scientific papers and literature, we decided that the format of the documents for the raw document storage should be in PDF format, which is a widely used format for academic papers and research articles. Additionally, the PDF format is suitable for the analysis process, as there are already many tools and libraries available for extracting text and metadata from PDF documents, such as the *PyPDFDirectoryLoader* and the *RecursiveCharacterTextSplitter* from the *Langchain*[2] library.

With efficient chunking and embedding of the raw documents, the Retrieval Model can retrieve relevant data based on user input, generate prompts for the generator component, and process the analysis results.

5.3 Uniform Input Data Format for Processing Between Components

To ensure structured input data for the analysis process, we need to define a specific format for the documentation of the input data. The input data consists of four main parts: system context, architectural approaches, quality attribute criteria, and quality attribute scenarios. Since ATAM is mostly a process consisting of natural language, we need to provide a format with clear and separated sections for each part of the input data. And since we are working with code and textual LLMs, the format must fit frontend and backend processing.

5.3.1 Using JSON as Structured Input Data

We propose to use JSON as the format for the input data, as it is a widely used and easy-to-read format that can be easily processed by both the frontend and backend components. It allows for structured data representation, ensuring that each part of the input data is clearly defined and separated. After the user uploads the data in the User Interface from various sections and fields, the data will be collected and formatted into a JSON object which will be sent to the Backend API for further processing. The JSON object will be structured as follows:

[2] https://python.langchain.com/

```
{
    "systemContext": {
        "systemPurpose": (Enter the system purpose here.),
        "technicalConstraints": [(Enter the technical constraints here.)],
        "systemInteractions": [(Enter the system interactions here.)]
    },
    "architecturalApproaches": [
        {
            "approach": Approach 1 (Enter the approach name here),
            "description": (Enter the approach description here.),
            "architecturalDecisions": [(Decision 1), ... , (Decision n)],
            "architecturalViews": [(Development View), (Physical View), (Process View)]
        },
        {
            "approach": Approach 2 (Enter the approach name here),
            "description": (Enter the approach description here.),
            "architecturalDecisions": [(Decision 1), ... , (Decision n)],
            "architecturalViews": [(Development View), (Physical View), (Process View)]
        }
    ],
    "qualityAttributeCriteria": [
        {
            "name": Criterion 1 (Enter the criterion name here),
            "metrics": [(Metric 1), ... , (Metric n)]
        },
        {
            "name": Criterion 2 (Enter the criterion name here),
            "metrics": [(Metric 1), ... , (Metric n)]
        }
    ],
    "qualityAttributeScenarios": [
        {
            "name": Scenario 1 (Enter the scenario name here),
            "qualityAttribute": Quality Attribute 1 (enter the quality attribute here),
            "environment": (Enter the environment here.),
            "stimulus": (Enter the stimulus here.),
            "response": (Enter the response here.)
        },
        {
            "name": Scenario 2 (Enter the scenario name here),
            "qualityAttribute": Quality Attribute 2 (Enter the quality attribute here),
            "environment": (Enter the environment here.),
            "stimulus": (Enter the stimulus here.),
            "response": (Enter the response here.)
        }
    ]
}
```

Listing 5.1: JSON Schema for Architectural Analysis Input

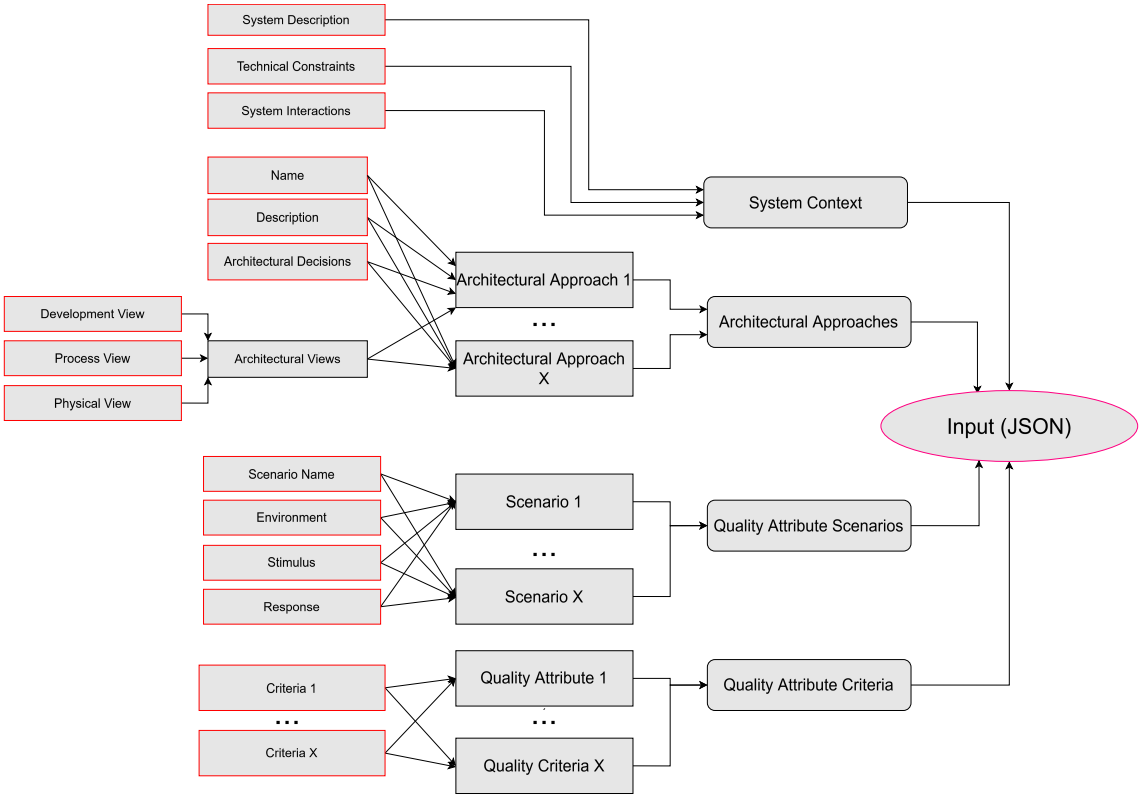

Figure 5.2: Structured Input Data Format in JSON. The red marked parts is the input which should be provided by the user.

Figure 5.2 shows the inputs from the requirements, that were defined in Section 4.2.1. The JSON object consists of four main parts: system context, architectural approaches, quality attribute criteria, and quality attribute scenarios. The red marked parts are the input data which should be provided by the user in text.

5.3.2 Documenting Architectural Views

In ATAM, architectural views are essential for understanding the system's structure and behavior [KKB+98], providing a visual representation of the system's components and their interactions. To ensure that the LLM model can process the architectural views correctly, we propose using *PlantUML*[3], a textual DSL for creating UML diagrams. PlantUML allows users to define UML diagrams using a simple and intuitive syntax, ensuring that the architectural views are accurately represented in a textual format. We also propose to include a limited number of views, such as the Development View, Physical View, and Process View, to provide a comprehensive overview of the system's architecture.

[3] https://plantuml.com/

Using Unified Modeling Language (UML) for Architectural Views

The 4+1 View Model by Krutchen [Kru95] is a widely used architectural view model that provides a comprehensive perspective on the system's architecture. Using the view model, we decided to include UML diagrams for the Development View (UML Component Diagram), Physical View (UML Deployment Diagram), and Process View (UML Sequence Diagram) in the input data. UML provides a standardized set of diagram types and symbols, ensuring consistent representation across various systems and projects [Rum16]. This standardization facilitates documenting and capturing architectural views in an understandable manner. And from Wang et al. [WWLL24] and Rouabhia et al. [RH24], we know that ChatGPT can process and evaluate UML diagrams to a certain degree, making it an attractive choice for the analysis process using other pre-trained models.

Using PlantUML for Textual Documentation of Architectural Views

PlantUML is a textual DSL for creating all kinds of UML diagrams, providing a simple and intuitive syntax for defining UML diagrams in a textual format. The PlantUML syntax is easy to learn and use, allowing users to create UML diagrams quickly and efficiently. Also regarding the analysis process, the textual format is more suitable for the LLM, since they are mostly primarily designed to process and generate human language by analyzing vast amounts of text data [HDW+23]. PlantUML syntax is also suitable for reading and understanding the architectural views since there are already studies that found out that both GPT-4 and Llama 2 could generate and interpret PlantUML syntax [Här23], suggesting that using more advanced models have the potential to interpret the PlantUML syntax. Referring to Section 5.3.1, the PlantUML syntax should be provided in the JSON object as a string for each view.

5.4 User Interface (Frontend)

The User Interface serves as the primary interaction point, enabling users to input data, view analysis results, and interact with the database. Given that the system relies on a textual LLM, all input data must be provided in a textual format to ensure accurate processing. Since the primary focus is on the analysis phase of the ATAM process, the User Interface is designed to be minimalistic yet effective, incorporating only the essential features necessary for supporting the workflow while maintaining usability. Therefore, it follows a single-page design, allowing users to seamlessly input data, run analyses, and review results without navigating through multiple screens, as shown in Figure 5.3.

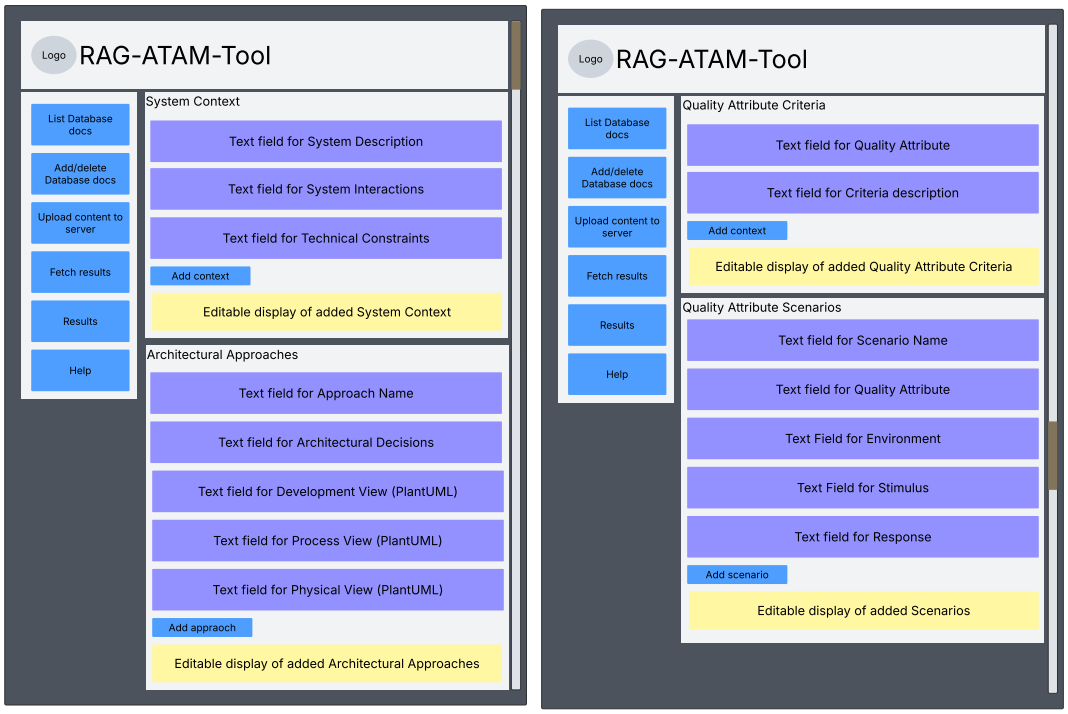

Figure 5.3: Blockframes of the User Interface

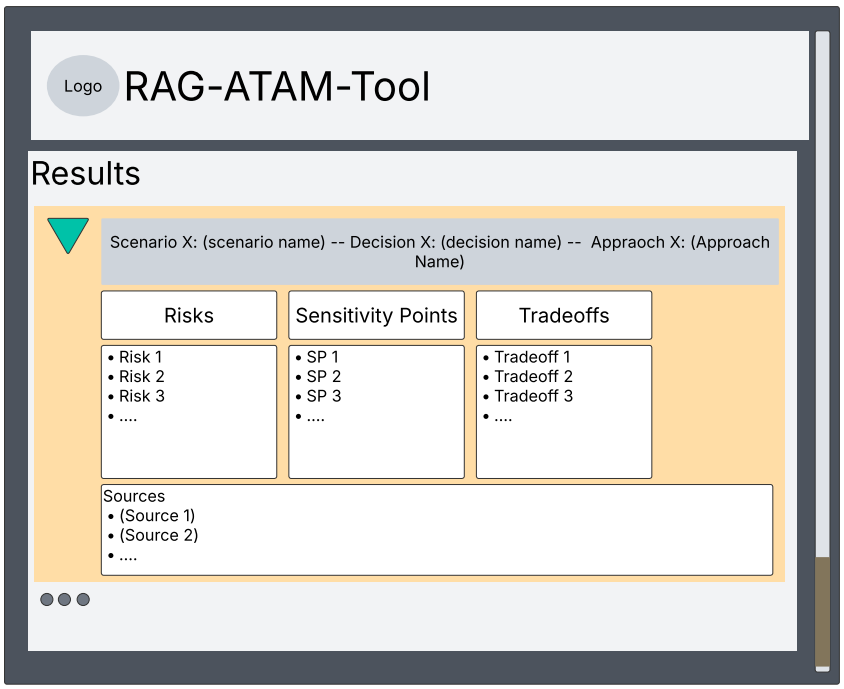

Figure 5.4: Results part of the User Interface

5.4.1 Input Fields

To meet the minimum UI requirements outlined in Section 4.2.4, the interface organizes input data into distinct sections, covering system context, architectural approaches, quality attribute criteria, and quality attribute scenarios. As illustrated in Figure 5.3, the system context section will include three text fields for specifying system purpose, technical constraints, and system interactions, as outlined in Section 4.2.1.

The architectural documentation section will incorporate fields for defining the approach name, description, architectural decisions, and architectural views. For the architectural views, users will be able to input PlantUML-based views in text format, ensuring representation of the system's structure and behavior.

For quality attribute scenarios, input fields will be available to capture details such as scenario name, quality attribute, environment, stimulus, and response. Similarly, the quality attribute criteria section will allow users to specify criteria names and required metrics.

5.4.2 Buttons and Functionalities

The user interface (UI) provides several interactive buttons to facilitate data management and analysis. Users can upload new data, initiate the analysis process, and manage stored documents within the system.

The Figure 5.3 suggests a rough structure of the front end. Each section (System Context, Architectural Approach, Quality Attribute Scenarios, and Quality Attribute Criteria) should include a button for adding the text data from the input fields. This button allows users to transform the data into JSON format and store it in an editable display field below for further review.

After the data is transformed into JSON format, the user can click an upload button to send the data to the Backend for further processing. The JSON objects from each input field will be combined into one JSON object (see Section 5.3.1) before sending. For starting the RAG pipeline and fetching the results, users can click a button, which will trigger the analysis process. After the analysis is done, the results will be displayed in a structured format within a dedicated output field, as shown in Figure 5.4.

The requirements also say that database interactions upload, delete, list, and download stored documents in the database. Using buttons for interaction, an additional display (e.g., a popup window) should then be displayed for document management, allowing users to interact with the database through the User Interface.

Besides the main functionalities, a "Reset" button is available to clear all input fields and results, ensuring a fresh workspace for iterative workflows.

Each button click triggers a corresponding request to the Backend API, which handles key functionalities. All communication between the frontend and backend is handled via HTTP requests, ensuring a structured data exchange in JSON format.

5.4.3 Results Display

The analysis results will be presented in a structured format within a dedicated output field at the end of the page. The output format will include a similar design as Table 2.1. But instead of writing all decisions in one table for one scenario, we split up the decisions into single parts for each scenario, decision, and approach. They are then organized by architectural decisions, approaches, and quality attribute scenarios, providing a clear overview of the risks, trade-offs, and sensitivity points identified in the analysis. The analysis results will be displayed in a collapsible table, organized by architectural decisions, approaches, and quality attribute scenarios, as depicted in Figure 5.4. For enhanced transparency, retrieved data sources will be explicitly shown. All the results will be taken from the JSON response from the Backend API and displayed in the output field (see Section 5.6.3).

5.5 Backend API

The Backend API serves as the intermediary between the User Interface and the RAG-pipeline, handling input data and ensuring efficient processing. It is designed to support modular, scalable, and maintainable interactions between the system components. The API is built using endpoints for different interactions, allowing seamless communication while maintaining flexibility for future extensions. Following Backend API endpoints must implemented to support the functionalities required by the system:

- **Database Managment:** This includes uploading new docs from the frontend to the database, deleting existing docs and their embeddings, and listing and downloading stored documents. This will be triggered by the user through the user interface by the interactive buttons mentioned in Section 5.4.2.
- **Fetch results by starting the RAG pipeline**: A button triggers the front end to send the user input in JSON format to the Backend API. It then triggers the retrieval model to fetch relevant data from the database, generates prompts for the LLM model, and processes the analysis results. The results are then sent back to the User Interface for display.
- **Resetting the system**: When the user clicks the reset button, the API clears temporary directories and response files to maintain a clean workspace.

By integrating these functionalities, the Backend API ensures smooth, structured, and efficient interaction between the User Interface, database, and Retrieval Model, forming the backbone of the system's architectural analysis workflow.

5.5.1 Access and Management of Document Database from the Backend API

The database is responsible for storing architectural knowledge and data chunks used by the Retrieval Model for retrieval operations. But besides the usage in the RAG pipeline, the user should be able to manage the database by adding new documents, deleting existing documents, listing stored documents, and downloading documents for further use.

Adding a Document

After the user sends a request to add a new document to the database, the Backend API should process the document by just adding it to the raw data storage. Since updating the vectorized data storage every time a new document is added is inefficient, the Backend API should only update the vectorized data storage when the user wants to fetch the results.

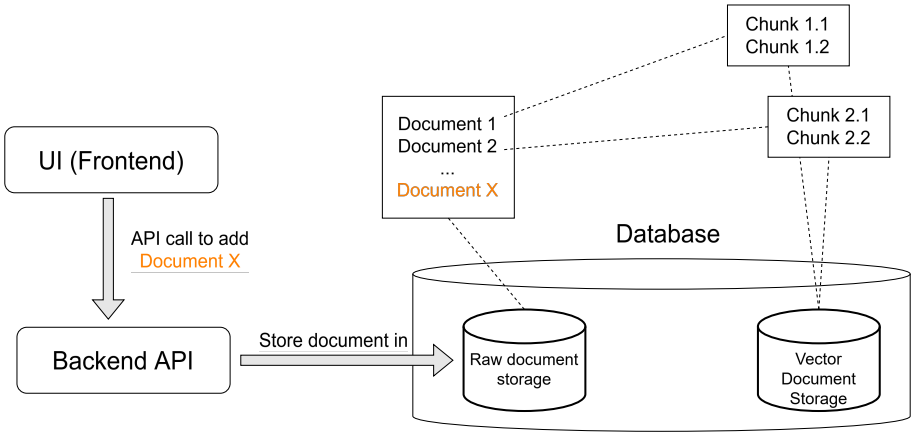

Figure 5.5: Adding a new Document to the Database.

View the List of Documents

When the User Interface send a HTTP request to the Backend API to list all stored documents, the Backend API should return a list of all stored documents of the database. The User Interface then displays the list of documents to the user, who can select a document to download it. To enable it to work, the Backend API endpoint should return a list of all stored documents in the raw storage database.

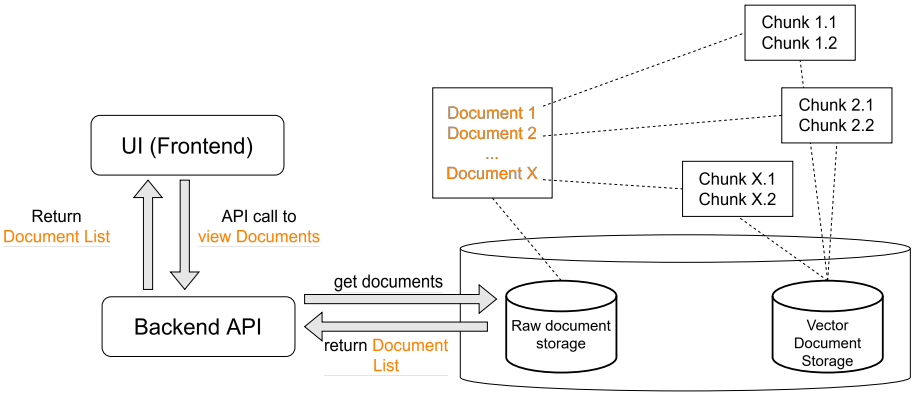

Figure 5.6: View the List of Documents from the Database.

Download the Documents

When the user wants to download a stored document, the Backend API should return a list of all stored documents in the database. In this case, we only need to access the raw data storage and download the necessary documents. To identify the document, the User Interface provides a list of all stored document names, which the user can select to download the document. The HTTP request to the Backend API then triggers downloading the selected document to the User Interface.

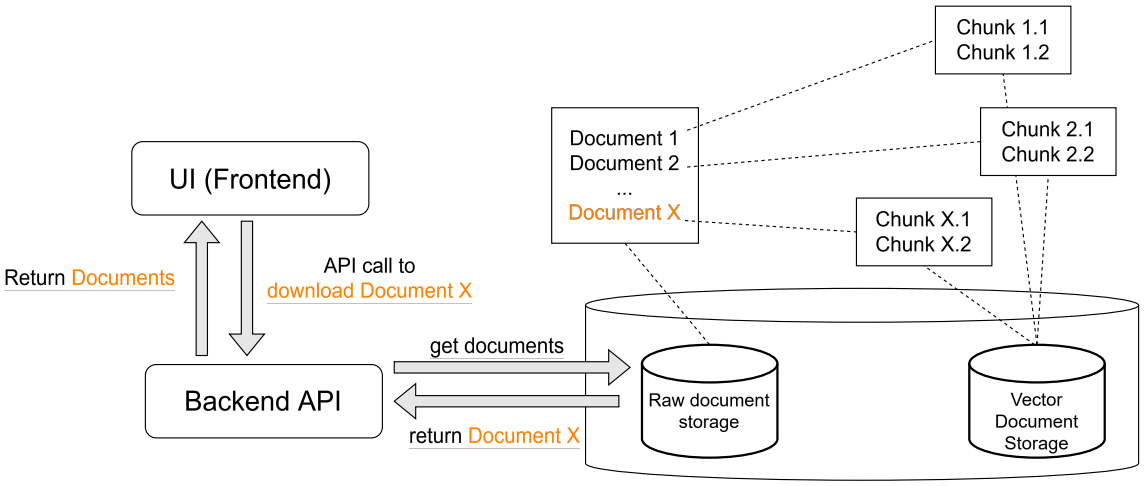

Figure 5.7: Download a Document from the Database.

Deleting a Document

When the user wants to delete a stored document, the Backend API should remove the document from the raw data storage and the vectorized data storage. In this case, the embedded chunks related to the document should also be deleted to ensure that the database is up-to-date. This can be done directly with an HTTP request because we just have to delete them without any embeddings or other operations.

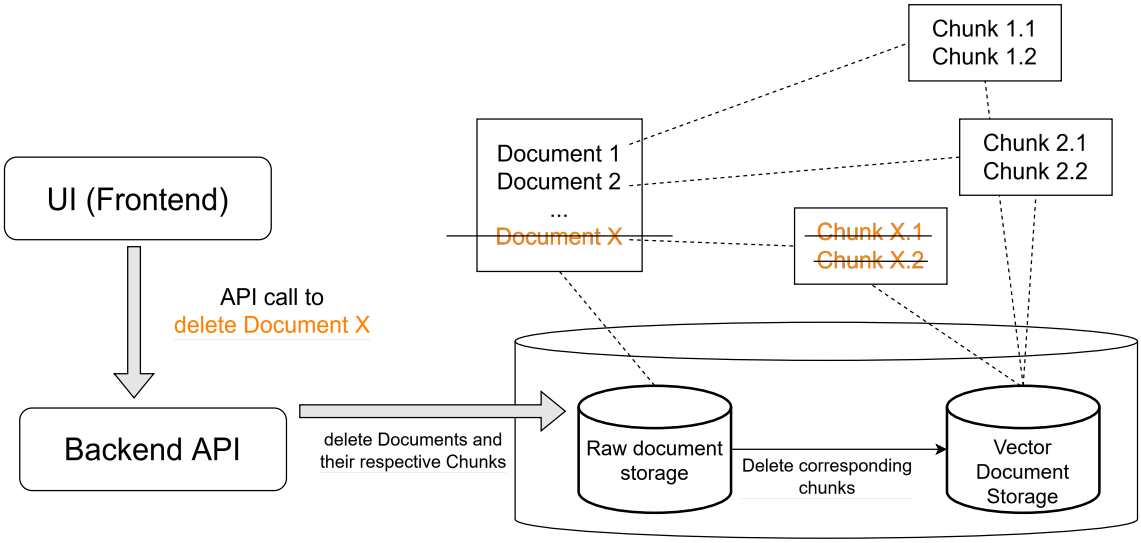

Figure 5.8: Delete a Document from the Database.

The deletion of the documents in the raw data storage and the vectorized data storage ensures that the database is up-to-date and that no outdated data is used in the analysis process. Section 5.6.1 provides a detailed explanation of the deletion process.

5.6 The RAG Pipeline for Qualitative Architectural Analysis

The RAG pipeline is responsible for managing the retrieval and embedding operations, generating prompts for the LLM model, and processing the analysis results. Figure 5.1 shows the rough flow of the RAG pipeline, which consists of the Retrieval Model, the Generator model, and the Database consisting of raw data storage and vectorized data storage. In the following sections, we will discuss the single steps of the RAG pipeline in detail.

5.6.1 Creating/Updating the Vectorized Database

When the user wants to fetch the results, the Backend API endpoint triggers the RAG pipeline. The first step in the pipeline is to check whether the embedded data storage already created and up-to-date. In both cases we used a text splitter component and an embedding model are used to chunk and create the embeddings from the raw documents.

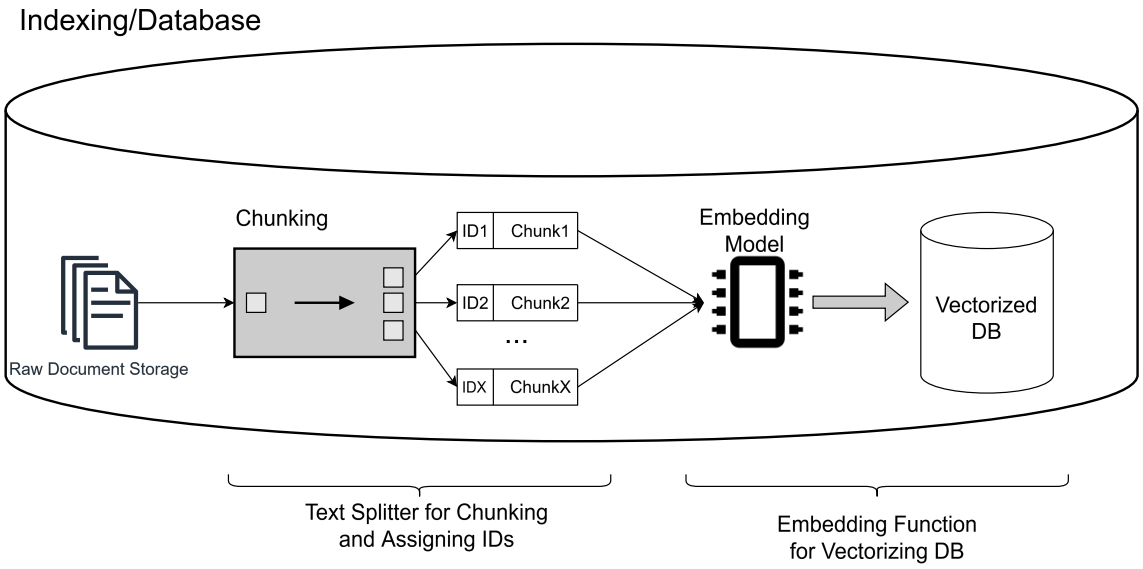

Figure 5.9: Creating/Updating the Vectorized Database Visualized.

Text Splitter Component

The text splitter component is responsible for chunking the raw documents into smaller parts. This can be relevant since raw documents with too many tokens could cause hallucinations in the LLM used for the analysis process [LLH+23].

Algorithm 5.1 shows the algorithm for loading, splitting, and assigning chunk IDs to the raw documents. For the splitting process, we use a text splitter component that splits the raw documents into smaller chunks. Chunk IDs are then assigned based on metadata, such as the source and page number of the document, ensuring that each chunk has a unique identifier. The structure of the ID is **`source:page:chunkIndex`**, where `source` is the document source (e.g, name of the document), `page` is the page number, and `chunkIndex` is the index of the chunk on the page. The `chunkIndex` is incremented for each new chunk on the same page, ensuring that each chunk has a unique ID. For a new page or source, the `chunkIndex` is reset to 0. Finally, the text splitter returns a list of text chunks with its unique metadata IDs.

Algorithm 5.1 Load, Split, and Assign Chunk IDs

```
1: Input: Directory containing raw documents
2: Output: List of text chunks with unique metadata IDs
3:
4: function GetChunks
5:     docs ← Directory containing raw documents
6:     chunks ← SplitDocuments(docs)
7:     AssignIDs(chunks)
8:     return chunks
9: end function
10:
11: function SplitDocuments(docs)
12:     splitter ← Initialize TextSplitter instance
13:     chunks ← splitter.Split(docs)
14:     return chunks
15: end function
16:
17: function AssignIDs(chunks)
18:     lastPageID ← None
19:     chunkIndex ← 0
20:     for all chunk in chunks do
21:         source ← chunk.metadata["source"]
22:         page ← chunk.metadata["page"]
23:         currentPageID ← source:page
24:         if currentPageID == lastPageID then
25:             chunkIndex ← chunkIndex + 1
26:         else
27:             chunkIndex ← 0
28:             lastPageID ← currentPageID
29:         end if
30:         chunk.metadata["id"] ← currentPageID:chunkIndex
31:     end for
32: end function
```

Embedding Model to Create/Update the Vectorized Database

The embedding model is responsible for converting the text chunks into embeddings, which are then stored in the vectorized database. Algorithm 5.2 describes the procedure for creating the vectorized database. It takes as input the raw documents, an embedding model (to transform the documents into vector representations), and a persistence directory for storing the database.

The process begins by initializing or loading the database with the provided embedding function and persistence directory. Next, the raw documents are broken down into smaller, meaningful chunks using the GetChunks function from the text splitter component (Section 5.6.1). The algorithm then checks the existing entries in the database by retrieving them with GetExistingEntries(). It extracts the unique identifiers (IDs) of these existing entries to identify which chunks are already stored in

the database. Any chunks that are already present in the database are filtered out. This is done by comparing the metadata of each chunk to the IDs of the existing entries. Only chunks with IDs not already present in the database are considered new. If no new chunks are found, the algorithm indicates that no new chunks need to be added and terminates. For any new chunks identified, the algorithm extracts their corresponding IDs and adds them to the database.

The algorithm concludes by indicating the number of new chunks added to the database. This approach ensures that the vector database is updated efficiently, with only unique and new chunks being added, thereby preventing redundant data from entering the system.

Algorithm 5.2 Vector Database Creation

```
1:  Input: Raw documents, Embedding Model, Persistence directory
2:  Output: Updated vector database with unique chunks
3:
4:  function CreateDatabase
5:      db ← Initialize/Load database with embedding function and persistence directory
6:      chunks ← GetChunks(Directory containing raw documents)
7:
8:      existing_items ← db.GetExistingEntries()
9:      existing_ids ← Extract IDs from existing_items
10:
11:     new_chunks ← Filter chunks where chunk.metadata[''id''] not in existing_ids
12:
13:     if new_chunks is empty then
14:         Print("No new chunks to add")
15:         return
16:     end if
17:
18:     new_chunk_ids ← Extract IDs from new_chunks
19:     db.EmbedAndAdd(new_chunks, ids=new_chunk_ids)
20:     Print("Added len(new_chunks) new chunks to database")
21: end function
```

Deleting all Chunks of a Document

When the user wants to delete a document, the Backend API should remove all chunks of the document from the vectorized data storage. This can be done directly with the HTTP request by passing over the name of the document to the Backend API.

Algorithm 5.3 shows the algorithm for deleting all chunks of a document from the vectorized data storage and the raw data storage. The algorithm takes the document name as input and deletes the document from the raw data storage. It then initializes the database and retrieves the metadata entries. The algorithm filters the metadata entries based on the document name and deletes the corresponding entries from the database. This ensures that all chunks associated with the document are removed from the vectorized data storage.

Algorithm 5.3 Delete Document and Associated Chunks

```
Input: Document Name
Output: Updated raw document storage with vector database without document chunks

function DeleteDocumentFromDatabase(Document Name)
    Try:
    raw_doc_path ← JoinPaths(Raw Document Directory, Document Name)

    if File raw_doc_path exists then
        DeleteDocumentFromFilesystem(raw_doc_path)
    else
        Error: "Document {raw_doc_name} not found."
    end if

    db ← InitializeDatabase(Persistence Directory)
    metadatas ← db.GetMetadatas()
    filtered_metadatas ← FilterMetadata(maetadatas, Document Name)
    for metadata in filtered_metadatas do
        db.Delete(metadata.get("id"))
    end for

end function

Catch:
return Error: "Failed to delete Documents and related chunks"
```

5.6.2 Retrieval Model

The Retrieval Model is responsible for managing, locating, and identifying the Vector Database [NBS+24], retrieving relevant data based on the input data, and generating prompts for the generator LLM. The following tasks are provided by the Retrieval Model:

- **Summarizing User Input:** To condense user queries for effective similarity search, we need to employ a model for this task. It was sufficient for extracting key terms and minimizing irrelevant tokens, contributing to reduced retrieval latency. Since the summarization task does not require deep analytical thinking, we can use more lightweight models to improve performance [FLK+24].
- **Embedding Operations:** An other model, that is more focused on embedding tasks, converts the user input into embeddings for retrieval operations at the database like query embedding.
- **Data Retrieval:** The Retrieval Model processes retrieval queries from the Backend API, fetching relevant data chunks based on the input data.
- **Prompt Generation:** The model generates prompts for the LLM model based on the retrieved data and user input, providing structured input for analysis.

- **Output Structuring:** Besides the analysis instructions for the LLM, the prompt also includes an additional task to output the solutions in a structured format for easier processing.

The rough flow of the Retrieval Model is illustrated in Figure 5.10. With this pipeline in place, we want to integrate the RAG methodology into the analysis process, ensuring that the LLM model receives the most important context from the database and the user input data.

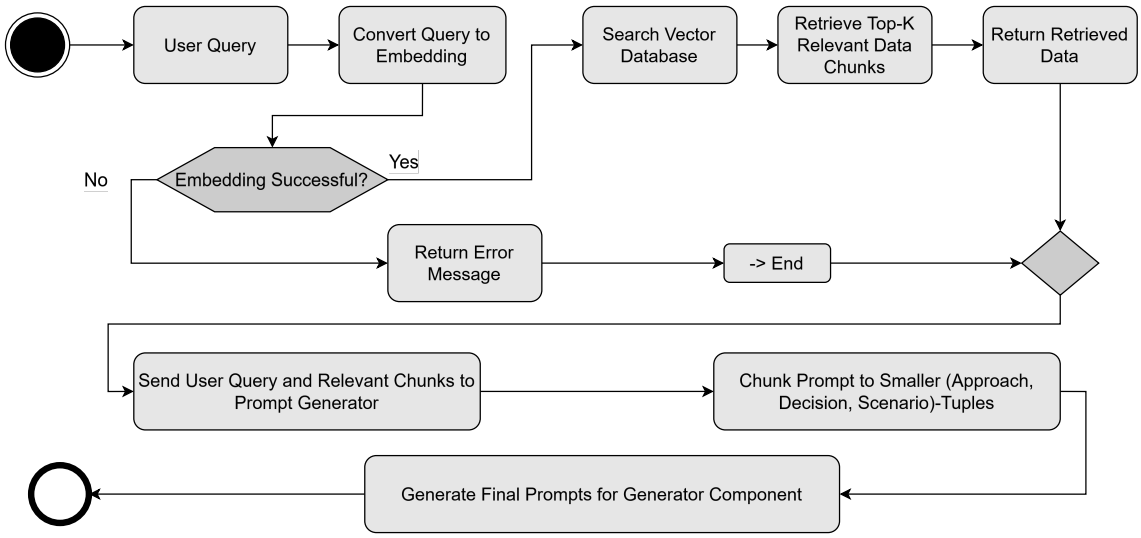

Figure 5.10: Rough Flow of the Retrieval Model

Embedding Operation for the User Query

The embedding operation is responsible for converting the user input data into embeddings for retrieval operations. To ensure that the user input data is correctly processed, the Retrieval Model uses the **same embedding model as the one used for creating the vectorized database**. This ensures that the user input data is transformed into the same vector space as the database.

To improve finding the most relevant data chunks for the user input, we could compress the user query to only include the most important terms and phrases. This can be done by using another pre-trained LLM (besides the model for the qualitative analysis), which can compress the user query to a few sentences, which can then be used for embedding operations. For this task, we can use models with a fast response generation time to improve performance. Since summarization does not require deep analytical thinking, we can use smaller models with fewer parameters like *Mistral 7B* [JSM+23], which generally have a faster response time [XCW+25].

Data Retrieval

After the user query is summarized and embedded, the Retrieval Model processes the retrieval query.

The summarized query is converted into a dense vector representation using the same embedding function as the documents in the vector database. To identify relevant information, the system performs a similarity search between the vectorized user query and the stored document embeddings.

Cosine similarity [GSB18] is employed as the distance metric to measure how closely a document vector aligns with the query vector. This method captures the angular similarity between vectors, making it particularly effective for comparing textual semantic meaning. It basically calculates the cosine of the angle between the two vectors using the following formula:

$$\text{cosine\_similarity} = \frac{Q \cdot D}{\|Q\| \times \|D\|} \tag{5.1}$$

where Q is the query vector, D is the document vector, and $\|Q\|$ and $\|D\|$ are the magnitudes of the vectors [GSB18]. The ranges of the cosine similarity values are between -1 and 1, where 1 indicates that the vectors are identical, 0 indicates that the vectors are orthogonal, and -1 indicates that the vectors are diametrically opposed [GSB18].

The system retrieves the top-K most relevant document chunks based on their similarity scores. These chunks serve as contextual evidence for the final reasoning step performed by the LLM. This retrieval-augmented process ensures that responses are grounded in factual and contextually appropriate information.

Prompt Generation for the Generator Model (LLM)

Prompt structuring for the generator model (LLM) is crucial for ensuring that the model can process the input data correctly and generate accurate analysis results [CZLZ24]. The design of the prompt for qualitative analysis prompt invocation follows the outline presented in Listing 5.2, ensuring a systematic and structured approach to architectural evaluation, as outlined in Section 2.3. It consists of four key components: task description, user input, contextual snippets, and structured output format. The task section defines the LLM's role as an expert analyst, guiding it in assessing architectural decisions against quality attribute scenarios and generating a structured JSON response. The user input section provides essential details, including the architecture context, approach description, PlantUML-based views, quality criteria, and scenarios, ensuring that the model has the necessary context for its analysis. Additionally, the snippets from the articles section incorporate relevant database content, enriching the model's understanding with external knowledge. Finally, the output format enforces a structured JSON response, clearly outlining identified risks, trade-offs, and sensitivity points, with each insight appropriately sourced as LLM knowledge or database references.

To refine the ATAM process further, where the scenario gets analyzed against the architectural decisions, we want to split the whole analysis into smaller parts to ensure that the LLM can process the input data deeper and more accurately. The number of prompts can be calculated with the following formula:

$$(5.2)\quad \textbf{|Prompts|} = \sum_{|\text{Architectural Approaches}|} \left(\prod_{|\text{Scenarios}|} (\textbf{|Architectural Decisions of Approach|}) \right)$$

This means that for example if we have three scenarios, two architectural decisions, and two architectural approaches, we will have 12 prompts in total, ensuring that the LLM can process the input data more accurately and deeper since the input data is split into smaller parts. Regarding Listing 5.2, we only need to add the current decision, approach, and scenario to the prompt, so we can focus on the current part of the input data and leave out the rest, reducing the confusion for the LLM.

```
<TASK>
    - Role: Expert Analyst in Software Architecture
    - Objective: Conduct qualitative analysis of architectural decisions against
        quality attribute scenarios
    - Inputs: User-provided data and contextual information from the database
    - Outputs: Structured JSON detailing risks, trade-offs, and sensitivity points
    - Guidelines: Utilize relevant snippets from articles and user input data
</TASK>


<USER INPUT>
User query:
    - Architecture Context: [System's goals, interactions, and constraints]
    - Current Architectural Approach: [Details of current architectural approach]
    - Architectural Views: [Representations in PlantUML syntax]
    - Quality Criteria: [List of quality attributes and their respective
        criteria/questions]
    - Current Scenario: [Specific quality attribute scenario to be analyzed]
</USER INPUT>


<CONTEXT>
    [Chunk 1]
    [Chunk 2]
    [Chunk 3]
    <!-- Additional chunks dependent on configuration -->
</CONTEXT>


JSON Response format

{
  "architecturalApproach": [Current_Approach],
  "scenario": {
      "name": [Scenario_Name],
      "qualityAttribute": [Quality_Attribute]
  },
  "architecturalDecision": [Decision],
  "risks": [
      {
```

```
                "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
                "details": [Description of identified risk]
            }
            // Additional risk entries as necessary
        ],
        "tradeoffs": [
            {
                "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
                "details": [Description of identified trade-off]
            }
            // Additional trade-off entries as necessary
        ],
        "sensitivityPoints": [
            {
                "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
                "details": [Description of identified sensitivity point]
            }
            // Additional sensitivity point entries as necessary
        ]
    }
```

Listing 5.2: Rough Prompt Structure

5.6.3 Generator Model

After the Retrieval Model has fetched the relevant data from the database and generated the prompts for the LLM model, the Generator Model takes over the analysis process. The Generator Model acts as the expert analyst in the ATAM process, evaluating architectural decisions against quality attribute scenarios and identifying risks, trade-offs, and sensitivity points. After processing the input data and generating the analysis results, the Generator Model sends back the structured output in JSON format to the Backend API, which then forwards the results to the User Interface for display.

LLM for Analysis

The LLM should conduct the qualitative analysis process. As far as the LLM is concerned, we will use a proper pre-trained model that is able to handle the requirements from Section 4.2.2. In the context of ATAM, the LLM should act as the architecture expert, identifying trade-offs, risks, and sensitivity points in architectural decisions against quality attribute scenarios. It will be instructed by the prompts generated by the Retrieval Model, which will guide the LLM in processing the input data and generating structured output results. This whole process will be the key part of the semi-automation of the ATAM process, since we only need to provide and structure the input data and the LLM will take over the analysis process. After the analysis is done, the model will send their response back to the Backend API in the desired JSON format, which will then send the results to the User Interface for display. Considering the installation of the LLM model, we will use the Ollama library, which provides an easy and efficient way to locally install and run the model.

Output Structuring

Using the prompt instruction from Listing 5.2, the LLM model generates the analysis results in a structured JSON format for each prompt. The output structure includes the current architectural approach, scenario, and decision, along with identified risks, trade-offs, and sensitivity points. Each insight is attributed to either the LLM knowledge or the database knowledge, ensuring transparency and traceability of the analysis results. The responses of the analysis are qualitative since generating quantitative results would require a more complex model and a more detailed input data structure.

```
{
  "architecturalApproach": [Current_Approach],
  "scenario": {
      "name": [Scenario_Name],
      "qualityAttribute": [Quality_Attribute]
  },
  "architecturalDecision": [Decision],
  "risks": [
      {
          "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
          "details": [Description of identified risk]
      }
      // Additional risk entries as necessary
  ],
  "tradeoffs": [
      {
          "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
          "details": [Description of identified trade-off]
      }
      // Additional trade-off entries as necessary
  ],
  "sensitivityPoints": [
      {
          "source": [LLM KNOWLEDGE] or [DATABASE KNOWLEDGE],
          "details": [Description of identified sensitivity point]
      }
      // Additional sensitivity point entries as necessary
  ]
}
```

Listing 5.3: Generated Output Structure

For each (Scenario, Architectural Decision, Architectural Approach) combination, the LLM model generates a structured JSON response, outlining the identified risks, trade-offs, and sensitivity points. The amount of risks, trade-offs, and sensitivity points should very between one and three each, as the <TASK> section in the prompt structure (Listing 5.2) only requires the LLM to provide the most important insights for each combination. This ensures that the output does not contain too much redundant information and that the user can focus on the most important insights for the analysis process.

Besides the qualitative analysis results, the prototype should also add the sources of the chunks to the output, so that the user can see where the insights are coming from. This ensures that the user can trace back the insights to the database or the LLM model, making the analysis results

transparent and traceable. We can easily just add a source field (see Listing 5.4) to the JSON output by appending another JSON object to the output structure, which includes the IDs of the chunks (refer to Section 5.6.1).

```
{
 "sources": [
     "(Chunk ID 1)",
     "(Chunk ID 2)",
     "(Chunk ID 3)",
     ...
 ]
}
```

Listing 5.4: ChunkIDs stored in JSON

Finally, the composed JSON outputs are sent back to the Backend API, which then forwards the results to the User Interface for display.

6 Implementation

To implement the RAG-based analysis mechanism from Chapter 5, we need to implement a tool with all necessary components. The implementation of the RAG ATAM Tool is open-source and available on GitHub[1] or on Zenodo[2] as a ZIP file.

In general, the prototype consists of a frontend (Angular-based UI[3]) and a backend (Flask API[4] with retrieval and model execution capabilities) with several components for document storage, embedding, retrieval, and LLM interaction. For the backend, we chose Python and Flask due to their simple API implementation and integration with libraries needed for RAG, such as ChromaDB[5], LangChain[6], and Ollama[7]. The backend handles requests, data retrieval, query processing, and LLM interaction, while the frontend provides an interface for data input, analysis execution, and result display. A core backend component is the Retrieval-Augmented Generation (RAG) pipeline, ensuring the LLM receives relevant contextual information by leveraging ChromaDB embeddings.

Table 6.1 maps the central system components to their respective implementation files, providing an overview of the system architecture. As mentioned before, the RAG pipeline will be implemented using Python, because of the vast ecosystem of AI libraries available. The main library used for the RAG pipeline is `LangChain`, which provides an easy way to integrate the LLM models and the retrieval operations, such as the *ChromaDB* database, which is used for storing the embeddings of the documents. It also provides a way to interact with the `Ollama` model management tool, which allows us to run the LLM models locally on the GPU server.

The following sections will provide an overview of the system architecture and the implementation details of the key components.

6.1 System Architecture

The architecture of the prototype follows a monolithic architecture with frontend, backend, and modular components, integrating various technologies to facilitate the semi-automated architectural tradeoff analysis process. Even though monolithic is not desirable due to its scalability and technological inflexibility [BOP22], it was chosen for the prototype to simplify the development process and focus on the core functionality.

[1] https://github.com/HuyLeDac/RAG_ATAM_Tool

[2] https://zenodo.org/records/15089678?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6ImFlNWFhOTg1LTE1MGQtNDNhNS04NDhmLTdmYTZmYmQ4NWIxNyIsImRh
RcPWub_GUDkKEMzOH2Zh_fJQ02gwibgezuURkWWt0qSteT7ClzMT5KdRBg0B9P3FxDWTUnndSAT-BS0mOhoo7A

[3] https://angular.dev/overview

[4] https://flask.palletsprojects.com/en/stable/api/

[5] https://python.langchain.com/docs/integrations/vectorstores/chroma/

[6] https://www.langchain.com/

[7] https://ollama.com/

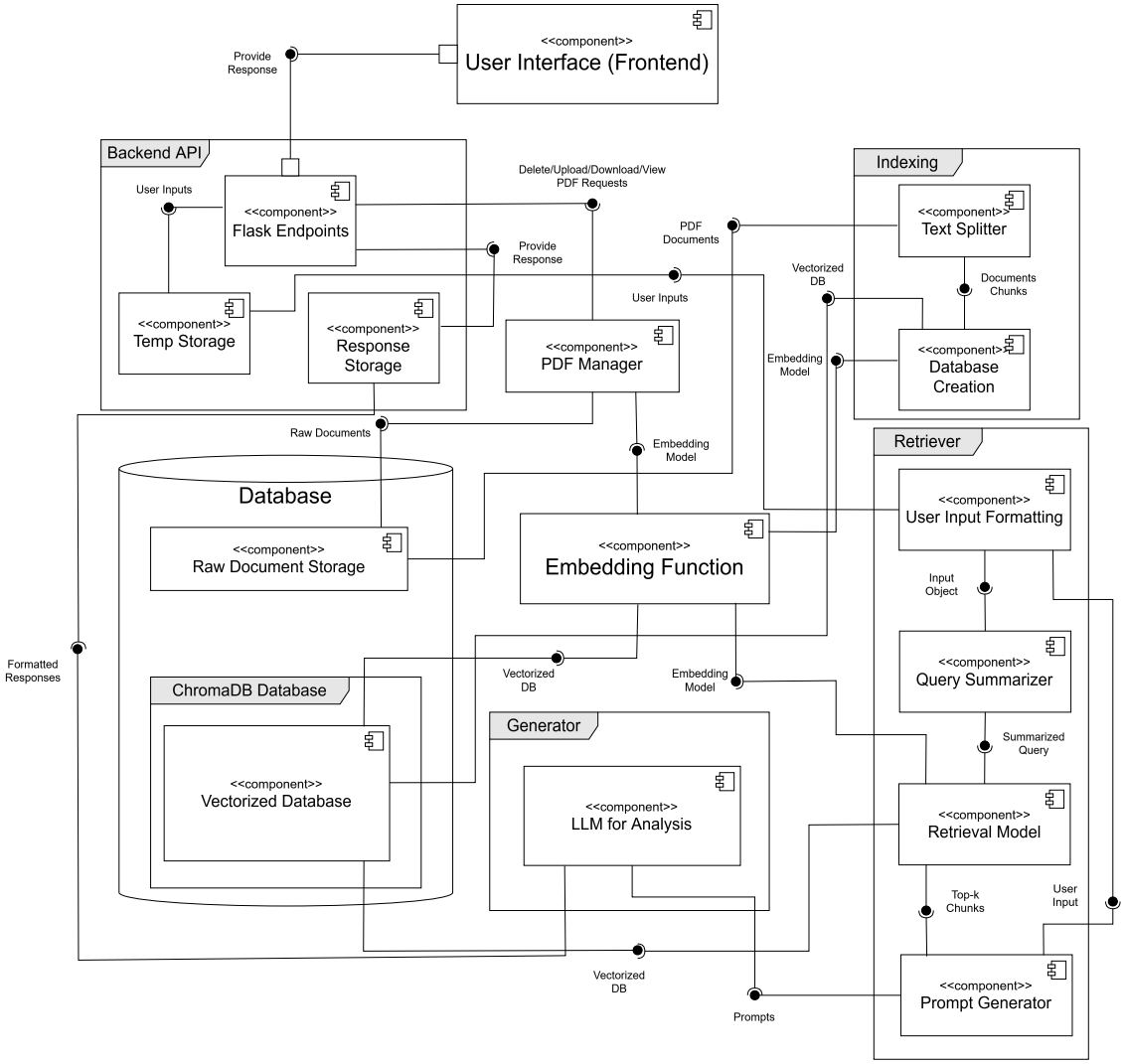

Figure 6.1: UML Component Diagram of the RAG ATAM Tool.

For our architecture design, we documented the components and their interactions in a UML component diagram, as shown in Figure 6.1. Table 6.1 maps the components to the implementation files of the RAG ATAM Tool for providing an overview of the project. We categorized the components into the following groups:

User Interface (Frontend): The component is an Angular-based UI that offers interaction with the user in terms of textual input fields, buttons, and result displays. Depending on the actions of the user, different requests are sent to the *Backend API* component, which then executes the corresponding operations.

Backend API: The main component of the system is the Endpoints provided by the Flask API, which routes the requests from the frontend to the appropriate processing components. For handling the raw document storage like uploading, listing, downloading, and deleting PDFs, the request will be forwarded to the *PDF Manager* component. Additionally, a *Temporary Storage* and *Response* component are used for temporary storage of the user input data and the final analysis results. The

temporary storage will be accessed by the *Retriever* component for further processing steps in the RAG pipeline. The response storage is used to store the final analysis results, which will be sent back to the frontend for display.

PDF Manager: The main tasks of the manager are to handle the document storage, adding, and deleting of raw documents (PDFs) in the database. For uploading, viewing, and downloading operations, the manager mainly accesses the *Raw Document Storage* component, which is the directory where the raw documents are stored. To delete the PDFs and the related vectorized documents, the manager additionally interacts with the *Vectorized Database* component. For that, it needs to access the *Embedding Function* component (provided by the LangChain library) to retrieve the embeddings of the documents.

Indexing: This component is responsible for creating and updating the vectorized database, which is used for the retrieval operations in the RAG pipeline [LPP+20]. Section 6.7.1 already provides a concept of the indexing process. The *Text Splitter* component loads and splits the PDFs into reasonable chunks using the `PyPDFDirectoryLoader` and `PyPDFTextSplitter` classes from the *LangChain* library. The raw documents are then converted into vector embeddings using the *Embedding Function*, which are stored in the *Vectorized Database* component.

Retriever: To retrieve the top-k most relevant documents from the vectorized database, the *Retriever* components first access the architecture input data provided by the *Temp Storage Components* and store the content into an *Inputs* object for further processing. The *Retriever* then uses the *Query Summarizer* component to create the query for the retrieval model, which is then sent to the *Retrieval Model* component. Using the *Embedding Function* component and the summarized user query, the retrieval model retrieves the top-k chunks from the vectorized database and generates the prompts using the *Prompt Generator* component. The *Prompt Generator* uses a prompt template.

Generator: Using a pre-trained LLM, the model generates the results of the analysis in a JSON template, which was given by the prompts provided by the *Prompt Generator* component. The responses are then stored in the *Response Storage* and forwarded to the *User Interface* for display.

System Component	Mapped Files	Description
User Interface (Frontend)	`frontend/` directory	Angular-based UI, handling, and formatting user input, sending API requests, and displaying results.
Backend API	`backend/app.py`, `backend/inputs/temp` directory `backend/responses` directory	Flask-based API that routes API requests and forwards them to the appropriate processing components and stores user input data and responses.
Database	`backend/data` `backend/database`	Raw Document Storage (data/) and Vectorized Database (database/)
Document Management	`backend/pdf_manager.py` `backend/get_embedding_function.py`	Handles document storage, adding, and deleting of raw and vectorized documents in the database.
Indexing	`backend/get_embedding_function.py`, `backend/create_database.py`, `backend/text_splitter.py`	Chunks and converts raw documents (`backend/data/`) into vector embeddings and stores them in the vectorized database (`backend/database/`) using an embedding function.
Retriever	`backend/get_embedding_function.py`, `backend/query.py`	Implements the retrieval model for retrieving relevant architectural context from the vectorized database using the same embedding model as in the database creation/update process.
Generator Model and Output Generation	`backend/query.py`	The LLM used for qualitative software architecture analysis.

Table 6.1: Mapping of Concepts from Chapter 5 to the Implementation Files.

6.2 Technical Constraints

The development of the prototype was influenced by various technical constraints that needed to be considered during the implementation, actively affecting the shaped architecture, functionality, and overall feasibility of the system. These constraints arise from hardware limitations, software dependencies, and the scope of the project. By identifying these constraints, we can better understand the trade-offs and design decisions made during the development process.

Hardware Limitations

The development and execution of the RAG-based ATAM prototype were carried out on a GPU server made available by the Chair of Software Engineering at the Technical University of Munich. We focused on running the prototype locally on the server to run the LLM models and the database operations, which required substantial computational resources.

The server was equipped with two **NVIDIA RTX 4090 GPUs**, which provided substantial computational power for large inference and retrieval tasks. However, large-scale LLM inference and RAG operations required careful memory management, particularly when dealing with multiple architecture candidates and retrieval operations. The constraints of the GPU server also introduced inference latency, despite the powerful hardware, which needed to be considered during the development of the prototype.

Besides that, the server was equipped with an **AMD Rypzen Threadripper PRO 3955WX** CPU with 16 cores and 32 threads, which was used for retrieval and pre-processing tasks like data chunking, document parsing, and handling queries in the Vector Database. While this CPU provided sufficient computational power for these tasks, it also introduced limitations in terms of time and resource constraints, since the CPU was shared among multiple users and tasks.

The system was also deployed on **Ubuntu**, which provides a stable environment for LLM operation, database operations, and web application hosting. However, the tool is optimized for Linux, limiting compatibility with Windows and MAC-based development environments.

Scalability Constraints

The prototype was primarily designed for small-to-medium-sized tasks, not massive enterprise-scale architecture evaluations. Therefore, scalability constraints were considered during the development process, particularly in terms of database management and retrieval operations.

LLM Performance Constraints

Combining the limitations and constraints of the hardware and software, we had to narrow down the size of the LLM model to a smaller version, since the large models like *LLaMA 3 405b* or *GPT-4* wouldn't fit on the GPU server. Therefore, we considered models with fewer parameters and a smaller context window, like the `llama3:70b` or `deepseek-r1:70b` model, which was used for the analysis.

6.3 Software Stack and Tools

The implementation of the RAG-based ATAM tool relies on a software stack that enables retrieval-augmented generation (RAG) operations and user interaction. The system consists of a Python-based backend, an Angular frontend, and various database and model integration components. This section provides an overview of the key technologies and tools used in the implementation, along with the reasons for their selection.

Backend

The backend is developed in **Python** using the **Flask Framework**[8], which provides a lightweight and efficient API for handling requests and managing data processing. Python was chosen for its rich ecosystem of AI and RAG-related libraries, making it well-suited for handling embedding models, retrieval operations, and API interactions. Flask is used due to its **minimal overhead**, allowing for quick development and seamless integration with other tools, which is perfect for prototyping and smaller systems.

Key backend technologies include:

- **Flask**: A lightweight web framework for handling API requests, selected for its simplicity, flexibility, and ease of deployment.
- **LangChain**: A framework used for embedding operations and retrieval, enabling easy integration with LLMs and supporting modular prompt engineering.
- **ChromaDB**: A vector database for storing and retrieving document embeddings, chosen for its similarity search capabilities and scalability in handling large datasets.
- **Ollama**: A model management tool for locally running most large language models, allowing for efficient execution without relying on external APIs, reducing latency.

Frontend

The frontend is built using **Angular**, a TypeScript-based framework, providing a dynamic and responsive user interface. Angular was selected for its component-based architecture, which promotes maintainability and scalability. It also offers strong state management capabilities, ensuring a smooth user experience when interacting with architectural data.

[8] https://flask.palletsprojects.com/en/stable/api/

6.4 Model Selection

In our RAG-enhanced prototype, we implemented multiple models for different purposes and tasks. The selection of the models was based on the hardware constraints from Section 6.2 and the requirements from Section 4.2. We narrowed down the selection and implementation of the models to the open-source models provided by `Ollama`, which allowed us to run the models locally on the GPU server. Another reason for the selection was the compatibility with the `LangChain` framework (for embedding operations and RAG functionalities), which provides an easy way to integrate the models into the backend. In total, we used the following models for the following tasks:

1. **Model for Embedding/Retrieval Operations:** For the embeddings of the documents and user query embeddings, we want to use a model that is specified for this task. After comparing different models, we chose using the **`nomic-embed-text`** [NMDM25] embedding model. With a context length of 8192 tokens, it provides a good balance between performance and memory usage, which is important for the embedding operations and the retrieval model. According to Nussbaum et al., the model outperforms OpenAI's Ada-002 and text-embedding-3-small in both short and long-context tasks while being open-source, making it a suitable choice for our prototype.
2. **Model for Summarizing the User Query for Similarity Search:** As discussed in Section 5.6.2, we only want to keep the most relevant tokens and keywords from the user query to ensure a better retrieval performance. Since the summarization is a rather simple task that can be done with a smaller model [XCW+25], we used the **`mistral 7B`** [JSM+23] model from the Ollama model collection for this task, which is a smaller model compared to the *llama3:70b* model, but still provides good performance and fast results for summarization tasks. Additionally, using models with fewer parameters for summarization tasks can reduce the computational overhead and latency in the system [XCW+25].
3. **Model for the Generator in the RAG Pipeline:** After considering the hardware constraints and some testing between different models and amount of parameters, we chose using the **`llama3:70b`** model by Meta. With the open-source access and the context length of up to 128K tokens, Llama 3 70B can process extensive architectural documents in a single pass, facilitating comprehensive analysis [DJP+24]. Additionally, Llama 3 70B demonstrates superior language comprehension and reasoning abilities, enabling it to effectively analyze and interpret complex architectural documentation and codebases. We considered the `Llama 3 405B` model as well, but due to the hardware constraints and initial testing in the GPU server, we decided to use the smaller version of the model to keep the latency and memory usage low. `deepseek-r1:70b` [DGY+25] was also considered due to its reasoning capabilities, but the release date was after the implementation phase, so we decided to use the `llama3:70b` model.

6.5 Frontend Design and Communication

The RAG ATAM Tool frontend is a web-based user interface that allows users to manage documents, provide system descriptions, and interact with the RAG-based retrieval and generation pipeline. As mentioned in Section 6.3, it is built using *Angular*, providing a responsive and dynamic interface for users to input data, trigger analysis, and view results. Key features of the frontend include:

- **Sidebar Navigation Panel (Left):** Provides quick access to major system functions and buttons, as shown in Figure 6.2.
- **Input Fields (Center):** The structured input fields for system descriptions, architectural approaches, quality criteria, and scenarios. Each section contains an "Add context" button to structure the inputs into a JSON format, as suggested in Section 5.3.1. For modifying the JSON, the content for each section will be displayed below the "Add context" button, as shown in Figure 6.2.
- **Results Display (Bottom):**The output fields display the final analysis results, structured in collapsible tables for easy readability, as shown in Figure 6.4 andFigure 6.5 Each table documents the current architectural approach, decision, and quality attribute scenario as the header. The risks, sensitivity points, and tradeoffs are displayed as rows, with a category and description for each entry. In addition to that, a Source column is provided, which indicates whether the information was retrieved from the document or generated by the LLM. A list of the documents retrieved for the current analysis is also displayed allowing user to lookup the original document.

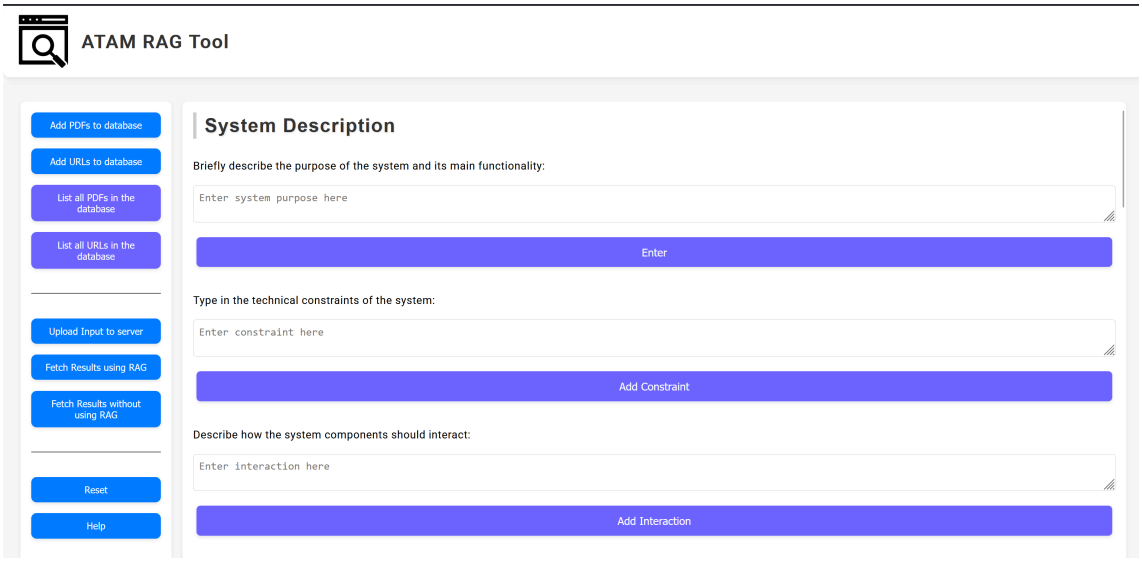

Figure 6.2: Frontend Design of the RAG ATAM Tool.

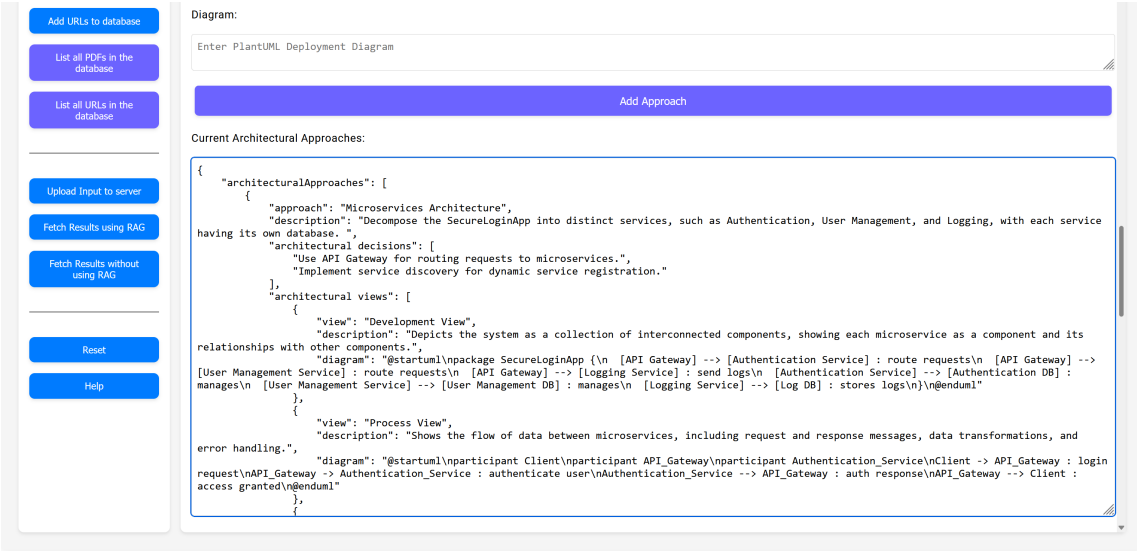

Figure 6.3: Capturing the input data into a structured JSON format

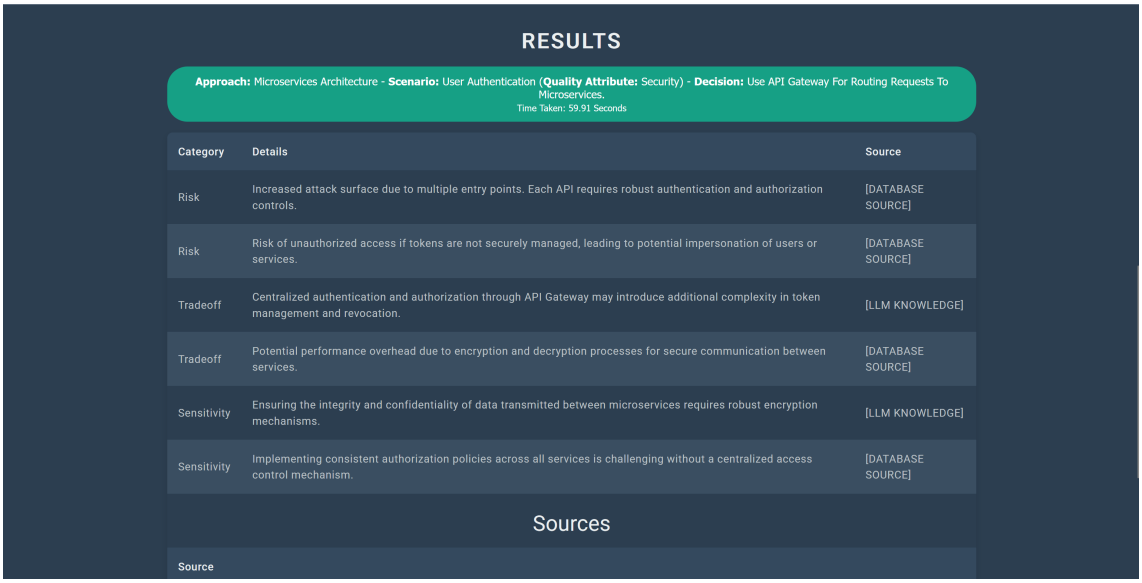

Figure 6.4: Results Reporting in the RAG ATAM Tool.

Figure 6.5: Source Reporting in the RAG ATAM Tool.

Regarding the communication between the frontend and the backend, the frontend sends structured JSON data to the backend via HTTP requests. The main interactions include:

- **Uploading PDFs to the database (/upload-pdf):** When clicking the "Add PDF" button, the user can select a file from their local system, which is then uploaded to the backend for storage.
- **Uploading Input Data (/upload-inputs):** After filling out the structured input fields and clicking the "Add context"-button, the user can click the "Upload Input to server" button to send the input data to the backend for processing. The JSON data from each section is then combined into a single JSON payload and stored in the backend for retrieval and analysis. The exact structure is already defined in Section 5.3.1.
- **Fetch Results using RAG (/get-results):** This button triggers the RAG pipeline in the backend and fetches the final analysis results for display in the frontend.
- **List Stored PDFs (/list-pdfs):** This button retrieves a list of all stored PDFs in the database, displaying them in a popup window for user selection. The popup window allows users to select a document for **deletion (/delete-pdf/<pdf_name>)** or **download (/download/<filename>)**.

Optionally, we also implemented a button that allows the user to add and view web-scraped articles, which are transformed into PDFs in the backend, to the database (`/upload-url` and `/get-urls`), but in the range of this thesis, we focused on the PDF upload and analysis. Also, we implemented a button that triggers the analysis process without the need of RAG, meaning that the user only uses the generator model for the analysis (/get-results-without-retrieval), which can be useful for testing the generator model or for a faster analysis process. But since we focus on the RAG pipeline, we will not go into detail about these functionalities.

6.6 Database Interaction with Frontend

The frontend offers different functionalities for the user to interact with the database, especially for the PDF files in the raw document storage, which lies within the `backend/data/` directory. These functionalities include adding, listing, downloading, and deleting PDFs from the database. We focused on implementing the functionalities from Section 5.5.1.

One difference is that we implemented a PDF Manager (pdf_manager.py file), which handles the document storage, adding, and deleting of raw documents in the database. The manager was created to separate the document management from the main app.py file, making the code more modular and easier to maintain.

6.6.1 Adding, Viewing and Downloading PDFs from the Database

When the user clicks the "Add PDF" button in the frontend, the user can select a PDF file from their local system, which is then packed into a POST request and sent to the `/upload-pdf` endpoint in the backend. Using the Flask `request` object, we can pass on the filename (`request.files['pdf'].filename`) and the content (`request.files['pdf'].`) itself to the `pdf_manager.py` file. The `add_pdf(pdf_name, pdf_content)` function in the `pdf_manager.py` file then stores the PDF in the `backend/data/` directory.

The user can list all stored PDFs in the database by clicking the "List PDFs" button in the frontend, which triggers the `/list-pdfs` endpoint in the backend. It then forwards the request to the `pdf_manager.py` file, which retrieves a list of all stored PDF names using the `get_all_pdfs()` function, which just returns the files in the raw document storage (`backend/data/`) directory. The data gets serialized into a JSON format and sent back to the frontend for display in a popup window.

For the downloading mechanism the user triggers the `/download-pdf/<pdf_name>` endpoint by clicking the "Download" button. By using the `download_pdf(pdf_name)` function of `pdf_manager.py`, the `send_from_directory` method from the Flask library is used to send the file back to the frontend. It takes the file name and the directory path (`backend/data/`) as arguments and packs the found object as an attachment for download.

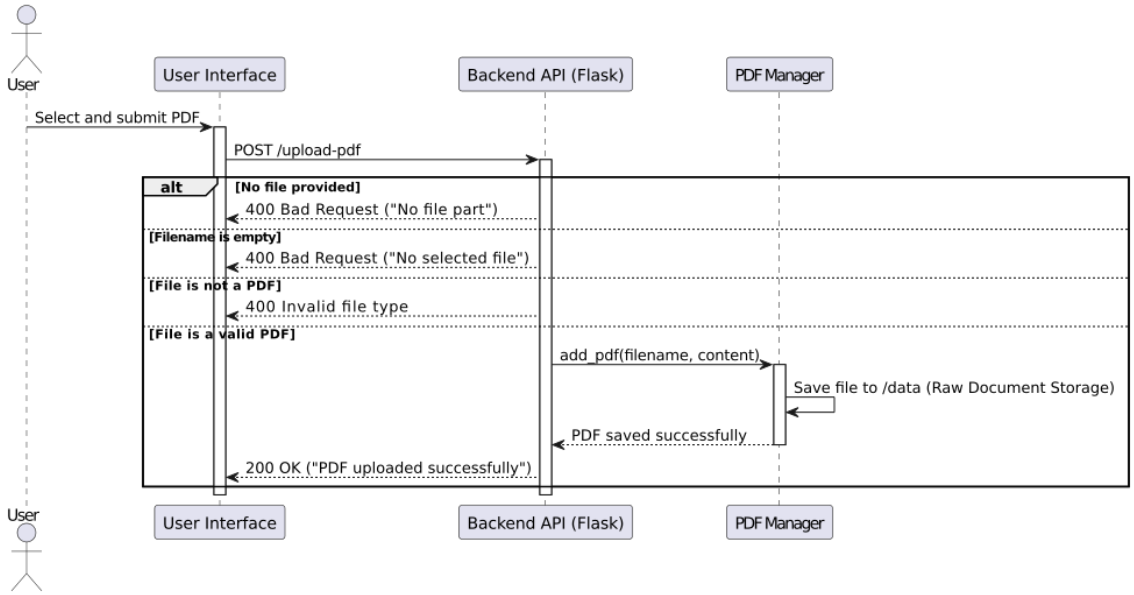

Figure 6.6: Sequence Diagram of Adding PDFs from the Raw Document Storage.

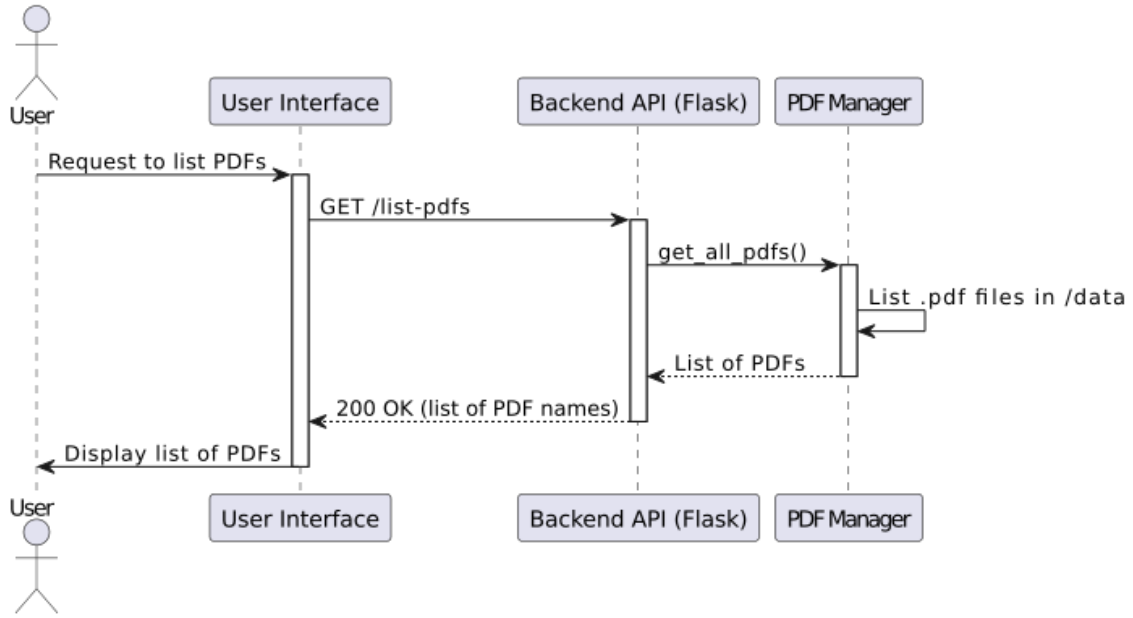

Figure 6.7: Sequence Diagram of Viewing PDFs from the Raw Document Storage.

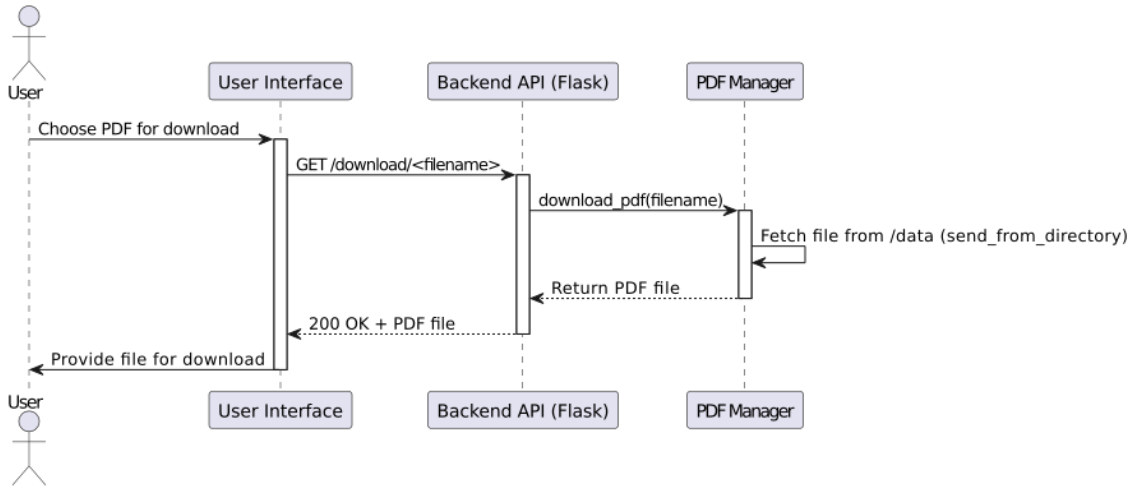

Figure 6.8: Sequence Diagram of Downloading PDFs from the Raw Document Storage.

6.6.2 Deleting PDFs from the Database

The "Delete Button" sends a request to the `/delete-pdf/<pdf_name>` endpoint, which triggers the `delete_pdf(pdf_name)` function of the PDF Manager. It then proceeds with two tasks: deleting the PDF from the raw document storage and deleting the corresponding vectorized document chunks from the database.

The deletion of the PDF from the raw document storage is done by using the `os.remove` method from the Python library, which deletes the file from the `backend/data/` directory. Since the raw document is successfully removed, the function proceeds with removing the corresponding vectorized representations of the document stored in the vector database. The system initializes a connection to the Chroma database using the `Chroma` class, which is configured to persist its data in the `backend/database` directory.

To find and remove the associated vectorized chunks, the function queries the database using `db.get(include=[''metadatas''])`, retrieving all metadata entries stored in the database. Each metadata entry is then checked for a matching `source` field that corresponds to the deleted PDF file. The source field, in this case, is the document name, which suffices to identify the vectorized chunks associated with the document, when referring to Section 5.6.1. The composition of the chunk IDs is done by concatenating the document name with the page number and the chunk number to the format: **`<pdf_name>:<page_number>:<chunk_number>`**. Any vectorized chunks associated with the deleted PDF (where `metadata[''source'']` matches `''data/<pdf_name>''`) are identified, and their unique IDs are extracted for deletion. These chunks are then removed from the vector database using the `db.delete(metadata.get(''id''))` method.

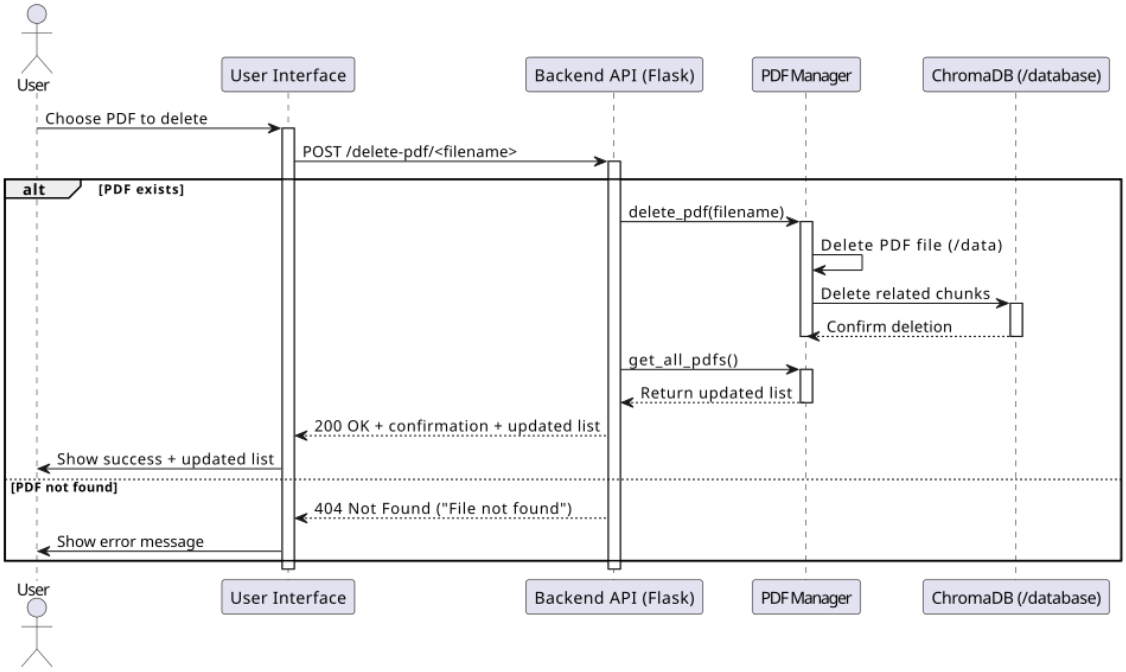

Figure 6.9: Download PDF from the Raw Document Storage.

6.7 RAG Pipeline

The pipeline is responsible for conducting the qualitative analysis of architectural decisions against quality attribute scenarios using the most relevant chunks of documents provided by the external database. The following sections provide an overview of the key components and functionalities of the RAG pipeline, focusing on the retrieval and generation operations.

6.7.1 Creating, Updating, and Indexing the Vectorized Database

When the user uploads the architecture inputs into the system (`backend/inputs/temp`), he can trigger the database creation and updating process with the `/get-results` endpoint. The process is initiated by the `create_database.py` script, which initializes the ChromaDB database with the `nomic-embed-text` embedding model and the `backend/database` directory for persistent storage. The script then calls the `get_chunks()` function from the `text_splitter.py` script, which serves as the entry point to processing raw documents. After chunking and indexing the documents, the script checks for new chunks by comparing the existing chunk IDs with the new ones. If new chunks are found, they will be embedded by the embedding function added to the database and persisted.

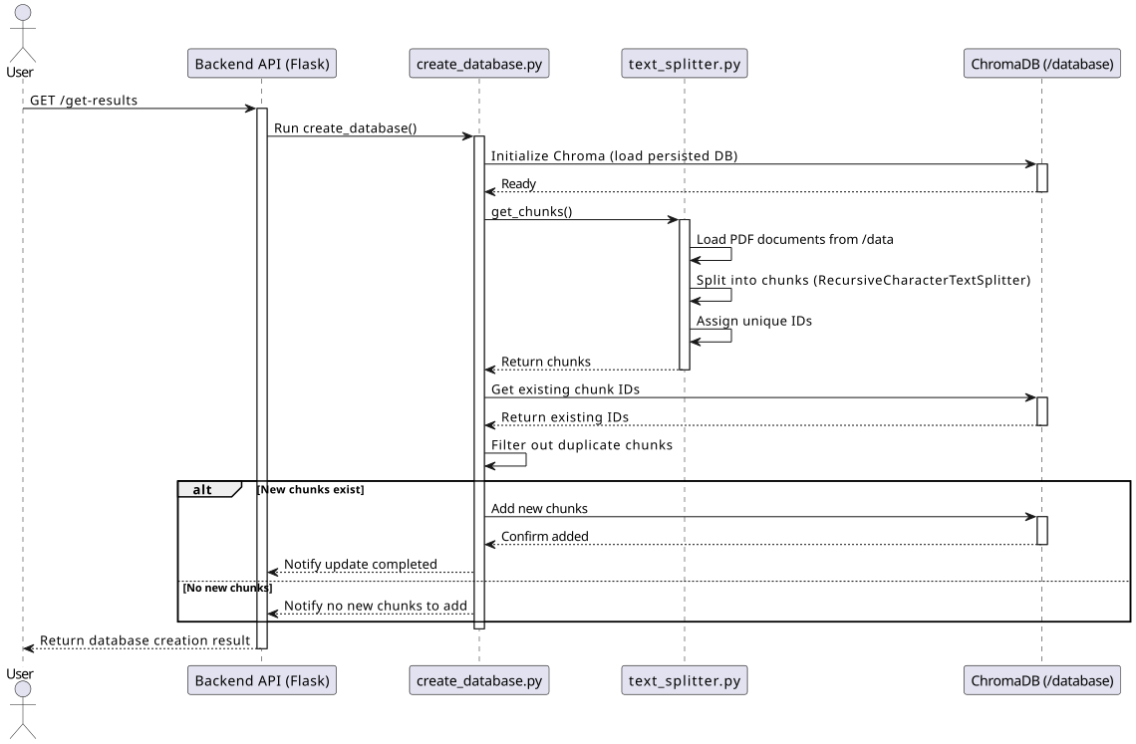

Figure 6.10: UML Sequence Diagram of the Indexing Process for the Vectorized Database.

Initializing the ChromaDB Database

The process of database creation and updating begins with the execution of the `create_database.py` script. This script initializes the ChromaDB database with two key components:

- **Embedding Function:** A function that converts text chunks into vector embeddings for efficient retrieval. As mentioned in Section 6.4, we use the *nomic-embed-text* embedding model for this task. It uses a transformer-based architecture to create vector representations of the text chunks, which are then stored for similarity search [NMDM25].
- **Persistent Directory:** The database is stored in `backend/database`, ensuring that the embeddings persist between different executions.

Once initialized, the script proceeds by calling the `get_chunks()` function from the `text_splitter.py` script, which serves as the entry point to processing raw documents.

Using a TextSplitter to Split and Index Documents

The `get_chunks()` function is responsible for loading, splitting, and indexing the documents. It first loads PDF documents from the `backend/data` directory using the `PyPDFDirectoryLoader`. These documents are then processed using the `RecursiveCharacterTextSplitter`, which splits the text into smaller chunks. The chosen parameters for splitting are:

- **Chunk Size: 800 characters** – This value ensures that each chunk remains manageable in size while capturing enough context for meaningful processing.
- **Chunk Overlap: 80 characters** – Overlapping chunks help maintain the continuity of information, preventing loss of context at split boundaries.
- **Length Function:** `len` – Uses the character length as the basis for splitting, ensuring consistent chunk sizes.

This approach balances chunk size and overlap, optimizing for both retrieval accuracy and storage efficiency. A smaller chunk size would lead to fragmented information, while a larger chunk size would reduce retrieval accuracy.

Assigning Unique Identifiers

Once the documents are split into chunks, each chunk is assigned a unique identifier. This is crucial for tracking individual text segments and avoiding duplication. The ID is structured as:

(6.1) ID = source_file : page_number : chunk_index

where:

- **source_file** - The filename of the original document.
- **page_number** - The page from which the chunk originates.
- **chunk_index** - The position of the chunk within the page.

The chunk index is initialized with 0 and is incremented for each subsequent chunk on the same page. When a new page is encountered, the index is reset to 0. This structure guarantees uniqueness since two chunks cannot have the same combination of source, page, and position. Additionally, it allows for efficient tracking of document segments.

Database Update Mechanism

To prevent unnecessary duplication, the script retrieves all existing chunk IDs from the database before adding new data. The steps are as follows:

1. Retrieve existing chunk IDs stored in ChromaDB.
2. Compare the new chunk IDs against the existing ones.
3. If new chunks exist, add them to the database and persist the update.
4. If all chunks are already stored, terminate the process without modification.

This process ensures that the database only updates when necessary, avoiding redundant storage and unnecessary computation. The final step logs the number of new chunks added or notifies that no new data was found.

6.7.2 Retrieval and Generation Operations

The retrieval and generation pipeline processes architectural data by iterating over each combination of *architectural approach, decision, and quality attribute scenario*. Instead of analyzing the entire user input at once, the system decomposes it into smaller, focused queries, ensuring more focused and contextually relevant results.

Once the database is created, the *Backend API* triggers the *Query Analysis Script*, which loads structured inputs from JSON files. For each combination, a specialized retrieval query is generated and summarized to extract key aspects such as *risks, trade-offs, and sensitivity points*. The summarized query is then used to retrieve the most relevant document chunks from the *ChromaDB* (`backend/database`) vector database. This ensures that only the most contextually relevant information supports the architectural decision-making process.

With the retrieved context, a structured analysis prompt is created and sent to the *Analysis Model (LLM)*, which evaluates the architectural decision based on its implications for the given scenario. The model identifies key insights, which are formatted into a structured response and stored in the *Responses Storage*. Finally, the processed results are returned to the user as a JSON response, completing the retrieval and generation cycle. This iterative approach ensures that each decision is analyzed within its specific context, improving retrieval accuracy and analysis depth.

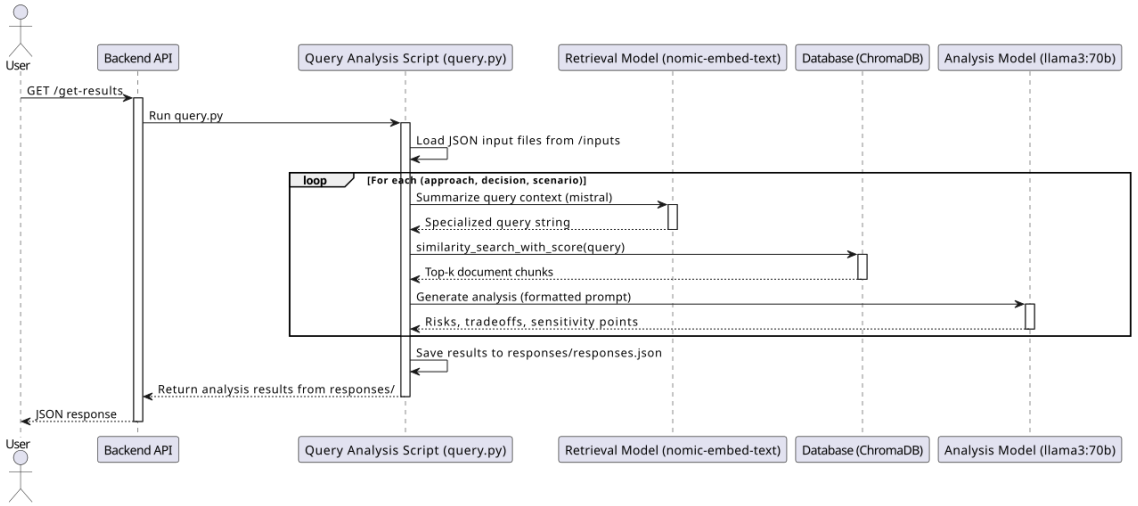

Figure 6.11: Sequence Diagram of the RAG Pipeline.

User Input Formatting

The backend retrieves user input data from the directory `/backend/inputs/temp/`, which contains structured JSON files, that were formatted by the frontend. The retrieval process is managed by the `InputLoader` class, ensuring that all input components—architecture context, architectural approaches, quality criteria, and scenarios—are properly loaded and structured.

Validation and Error Handling: Before proceeding with data processing, the backend performs validation checks:

- Ensures that all required JSON files are present.
- Validates that each file contains properly formatted JSON data.
- Checks for missing fields within the JSON structures.

If any required file is missing or contains incorrect formatting, an appropriate error response is returned to the user.

Retrieving Input Data: The `InputLoader` class initializes with the directory path containing the input JSON files. The following files are expected within `/backend/inputs/temp/`:

- `architecture_context.json` – Defines the overall system architecture description, system interactions, and technical constraints.
- `architectural_approaches.json` – Contains multiple architectural approaches under consideration.
- `quality_criteria.json` – Lists key quality attributes relevant to decision-making.
- `scenarios.json` – Includes different use-case scenarios for architectural evaluation.

Each of these files is parsed using Python's built-in `json` module, ensuring proper data integrity and error handling.

Structuring the Input Data: Once retrieved, the data is structured into an instance of the `Inputs` class. This class encapsulates:

- `architecture_context` – Stores the system's high-level architecture description.
- `architectural_approaches` – Stores different approaches and their respective decisions.
- `quality_criteria` – Stores key evaluation metrics for architectural decisions.
- `scenarios` – Stores different system scenarios relevant to the analysis.

This representation allows access to the input data while ensuring consistency in processing.

Extracting Architectural Decisions and Views: To facilitate analysis, the `InputLoader` class provides methods for extracting architectural decisions and views:

- `get_architectural_decisions_as_list()` – Iterates through the architectural approaches to extract all architectural decisions.
- `get_architectural_views_as_list()` – Extracts all architectural views associated with each approach.

These provided methods enable the backend to process architecture-related decisions and retrieve specific parts of the JSON.

Query Summarizer

The query summarizer encapsulates the most relevant information from the user query, ensuring that the retrieval process focuses on the essential aspects of the input data. In Section 6.4, we discussed the use of the `mistral 7B` model for summarization tasks, which is integrated into the `query.py` script.

To achieve this, we implemented the function `create_specialized_retrieval_query()`, which constructs a structured query string based on the provided inputs, including:

- The *architectural approach* under consideration.
- The *specific architectural decision* being evaluated.
- The *quality attribute scenario* relevant to the decision.

Forming the Retrieval Query: The function first formats the input data into a structured string representation, which can be seen in Listing 6.1.

```
input_data_str = f"""
Scenario: {format_json(scenario)}

Architectural Approach: {approach['approach']}
Architectural Decision: {decision}
"""
```

Listing 6.1: String from the User Input for Summarizing Task.

This representation ensures that the retrieval process is *contextually aware* of the specific scenario and decision under evaluation.

Summarizing Key Information To improve retrieval accuracy, the function (see Listing 6.2) leverages a summarization model to extract essential concepts from the query. This step ensures that the generated query is optimized for the retrieval process, focusing on the most critical aspects of the user input.

```
summarized_query = summarize_model.invoke(
 "Summarize the most important keyword points of the architecture, decision, and scenario. "
 "It should be ideal for Retrieval-Augmented Generation procedures. "
 "Add the keywords risks, tradeoffs, and sensitivity points under the summary:\n"
 + input_data_str
)
```

Listing 6.2: Python Code for Summarization Invocation.

The *summarization model* transforms the input query into a *compact, information-dense query representation*, capturing essential details while removing redundant data. The final summarized query is then returned for use in the retrieval process.

The summarization process enhances the retrieval mechanism by extracting *key concepts* from large architectural descriptions, thereby reducing *query noise* and improving search efficiency. It ensures that database retrieval focuses specifically on *risks, trade-offs, and sensitivity points*, which are critical for decision-making. Additionally, the process increases the *semantic relevance* of retrieved documents, ensuring that only the most relevant and meaningful information is considered for analysis.

Retrieval of Relevant Chunks from the Database

Once the query summarization process is completed, the system proceeds with retrieving relevant architectural context from the database. The retrieval process is based on a *vector similarity search*, ensuring that only the most relevant document segments are considered for further analysis.

Executing the Retrieval Query: The retrieval is performed by the function `retrieval_query()`, which takes the `summarized_query` as input and accesses the ChromaDB database (`backend/database`). This function utilizes a *similarity-based search mechanism*, which ranks stored document chunks based on their semantic closeness to the query.

The function operates as follows:

1. Receives the *summarized query* as input.
2. Performs a *vector similarity search* on the database using the query embedding function (`nomic-embed-text`).
3. Retrieves the *top-k most relevant* document chunks.
4. Returns these results along with their respective similarity scores.

Similarity-Based Ranking: The function executes a *similarity search with score*, which ranks document embeddings based on their proximity to the query embedding. The retrieval function selects the top-k results, where k is set to 4, ensuring that only the most relevant results are considered: We focused on the top 4 results since higher values like would increase the context size a lot, causing the generator model to potentially create a risk of generating irrelevant information and losing focus on the main query.

```
top_k_results = db.similarity_search_with_score(query_text, k=4)
```

Listing 6.3: Python Code for Retrieving Top-k Results from the Database Using Cosine Similarity.

The default configuration uses *Cosine Similarity* [GSB18] as the similarity metric. Each document chunk retrieved from the database contains *metadata* that includes its unique identifier, source file, and other relevant attributes. The function logs these retrieved chunks, providing insight into the relevance of each result:

```
for doc, score in top_k_results:
    print(f"Document ID: {doc.metadata.get('id')} - Score: {score}")
```

Listing 6.4: Python Code for Printing Retrieved Document Chunks.

The retrieval step is a critical component of the *Retrieval-Augmented Generation (RAG) pipeline*, ensuring that decision-making is supported by *relevant architectural context*. It enables *efficient access to relevant context* by retrieving only the most pertinent document chunks, thereby reducing unnecessary information. This process facilitates *context-aware decision-making*, as architectural decisions are grounded in well-structured and contextualized knowledge. Additionally, the retrieval mechanism helps in the *reduction of irrelevant information*, minimizing noise and improving the accuracy of subsequent analysis. By leveraging *vector similarity scoring*, the system enhances *retrieval precision*, ensuring that only the most semantically relevant documents are selected for further processing.

Generating Prompts for the Analysis Model

For this task, we mainly focused on implementing the principles from Section 5.3.1 and Section 2.3, by using the *Template* and *Chunking*, and *Persona* strategies by White et al. [WFH+23]. To analyze architectural decisions in relation to quality attribute scenarios, the system constructs a structured prompt that incorporates relevant architectural context, retrieved database snippets, and structured user input. The function `generate_analysis_prompt()` is responsible for formatting this prompt in a way that allows the language model to generate insightful assessments regarding risks, trade-offs, and sensitivity points.

Structure of the Prompt: The prompt is generated dynamically for each architectural approach, decision, and scenario combination. It consists of the following key sections:

- *Task Definition* – Provides the model with explicit instructions regarding the analysis to be performed.
- *User Input Section* – Embeds structured information about the architectural approach, decisions, and scenario under evaluation.

- *Retrieved Context Section* – Includes relevant document chunks retrieved from the knowledge base.
- *Structured Output Format* – Specifies the required JSON format for consistency in the model's response.

Task Definition and Context: At the beginning of the prompt, the model is explicitly instructed to conduct a qualitative analysis based on the *Architecture Tradeoff Analysis Method (ATAM)*. It is asked to provide an assessment of risks, trade-offs, and sensitivity points regarding the given architectural decision and quality attribute scenario. The following snippet outlines the initial part of the prompt:

```
<TASK>

You are an expert analyst for software architecture. We are conducting a qualitative analysis
of architectural decisions against quality attribute scenarios based on the Architecture
Tradeoff Analysis Method (ATAM).

You are given a user input of the approach: {current_approach}

Task:
Provide the risks, tradeoffs, and sensitivity points regarding the quality attribute scenario
for the architectural decision: {decision}.
</TASK>
```

Listing 6.5: Task Definition for the Generator Model in the RAG Pipeline.

This section ensures that the model understands its role as an architectural analyst and maintains a structured response approach.

User Input Section: The next section of the prompt incorporates structured user input related to the architecture under evaluation. This includes:

- The overall *architecture context*.
- A textual description of the *architectural approach*.
- The list of *architectural views* represented in PlantUML format.
- The *quality criteria* used in the decision-making process.
- The *specific scenario* for which the architectural decision is being evaluated.

The user input section is formatted as follows:

```
<USER INPUT>

User query:
Architecture Context: {architecture_context}
Architectural Approach Description: {approach_description}
Architectural Views (PlantUML): {architectural_views}
Quality Criteria: {quality_criteria}
Scenario: {scenario}

</USER INPUT>
```

Listing 6.6: Structured User Input for the RAG Pipeline.

By maintaining a structured format, the model can better contextualize its response and relate different architectural elements effectively.

Incorporation of Retrieved Context: The *retrieved snippets from external sources* (scientific articles, architectural best practices, or previous case studies) are included in the prompt to enhance the model's understanding and reasoning. This section appears as:

```
<SNIPPETS FROM ARTICLES>
{context}
</SNIPPETS FROM ARTICLES>
```

Listing 6.7: Incorporating Retrieved Context into the Prompt.

This enables the model to ground its response in both the provided user input and additional authoritative sources.

Enforcing a Structured Response Format: To ensure that the generated response remains structured and easily processable, the prompt concludes with explicit instructions on the expected output format. The response is requested in JSON format, ensuring that each decision analysis contains:

- The architectural approach.
- The scenario under evaluation.
- The identified risks, trade-offs, and sensitivity points.
- The sources used (*LLM Knowledge* or *Database Source*).

The enforced response structure is as follows:

```
Format the response in JSON format. Only use the format below and DO NOT ADD ADDITIONAL
CHARACTERS:
{
  "architecturalApproach": "{current_approach}",
  "scenario": {
    "name": "{scenario_name}",
    "qualityAttribute": "{quality_attribute}"
   },
  "architecturalDecision": "{decision}",
  "risks": [
    {
       "source": "[LLM KNOWLEDGE] or [DATABASE SOURCE] or [NO RELATIONSHIP]",
       "details": "(enter risks here)"
    }
  ],
  "tradeoffs": [
    {
      "source": "[LLM KNOWLEDGE] or [DATABASE SOURCE] or [NO RELATIONSHIP]",
      "details": "(enter tradeoffs here)"
```

```
    }
  ],
 "sensitivityPoints": [
   {
     "source": "[LLM KNOWLEDGE] or [DATABASE SOURCE] or [NO RELATIONSHIP]",
     "details": "(enter sensitivity points here)"
   }
  ]
}
```

Listing 6.8: Structured JSON Format for Model Response.

By enforcing this structured format, the system ensures that responses remain *consistent, machine-readable, and suitable for automated evaluation.*

Output Generation and Formatting

For the analysis model to generate meaningful insight, we provided the *LLaMA 3 70B* model with the prompt template generated in Section 6.7.2. The model is configured with a low temperature (`0.2`) to ensure deterministic responses, a `top_p` value of `0.7` to maintain response focus and a `top_k` setting of `50` to enhance structured output quality. Additionally, a `repeat_penalty` of `1.1` is applied to prevent redundant phrasing. Given the complexity of architectural decisions, the model is allowed a large context size of `8192` tokens to maintain coherence across lengthy architectural descriptions.

Processing the Model's Response: Once the model is invoked with the formatted prompt, the raw output is received and undergoes a post-processing phase. To ensure consistency and proper structuring, any unwanted Markdown formatting, that frequently occurred during generation (see Listing 6.9), is removed. The model's response is typically wrapped in a code block with the `json` tag, which needs to be stripped before further parsing:

```
# Get the model's response
response_text = model.invoke(formatted_prompt)

# Remove any Markdown-style code blocks containing "json"
cleaned_response_text = re.sub(r"```json\s*", "", response_text)
cleaned_response_text = re.sub(r"```$", "", cleaned_response_text)

# Ensure no trailing/leading whitespace
cleaned_response_text = cleaned_response_text.strip()
```

Listing 6.9: Post-Processing the Model's Response.

After removing unnecessary formatting, the response is parsed into a structured JSON format. This step ensures that the extracted information can be efficiently utilized for further processing.

```

# Attempt to parse the cleaned response as JSON
response_dict = json.loads(cleaned_response_text)
```

Listing 6.10: Parsing the Model's Response as JSON.

Incorporating Metadata and Tracking Processing Time: To maintain traceability, each generated response is supplemented with metadata that includes the architectural approach, decision, and scenario under evaluation. Additionally, if the retrieval was performed as part of the pipeline, the sources of the retrieved context are appended to the response dictionary:

```
# Add sources to the response dictionary
sources = [] if without_retrieval else [doc.metadata.get("id") for doc, _ in top_k_results]
response_dict['sources'] = sources
```

Listing 6.11: Incorporating Metadata and Tracking Processing Time.

The processing time for each query is also measured for experimentation purpose, ensuring that performance insights can be logged for further optimization:

```
# Track processing time
tuple_time_taken = tuple_end_time - tuple_start_time
log_data.append({
    "approach": approach['approach'],
    "decision": decision,
    "scenario": scenario['scenario'],
    "time_taken_seconds": tuple_time_taken
})

# Append response
response_dict['time_taken_seconds'] = tuple_time_taken
responses.append(response_dict)
```

Listing 6.12: Tracking Processing Time for Each Query.

Ensuring a Structured and Processable Output The final processed output maintains the structure defined in Section 6.7.2, ensuring consistency across all generated insights. Each response adheres to the expected JSON schema, allowing seamless integration with downstream evaluation and visualization tools.

By enforcing strict formatting and applying validation mechanisms, the generated insights remain reliable and interpretable for architectural decision-making. The responses are then sent back to the `backend/responses` directory for further processing and visualization at the frontend.

7 Evaluation

In this chapter, we present the evaluation of our research. The documented task sheets, questionnaires, and results can be viewed in Zenodo[1] We aim to assess the following research questions and hypotheses:

RQ1 How does the use of Retrieval Augmented Generation in ATAM compare to manual analysis in terms of the relevance of the results and time efficiency?

RQ2 How well does the RAG ATAM Tool support the analysis process in terms of usability?

We hypothesize that integrating RAG into the ATAM process improves efficiency by reducing the time required for analysis while maintaining or enhancing the relevance of the results. Additionally, we expect that the structured retrieval and automated assessment provided by RAG will enhance usability, making the analysis process more accessible to users with varying levels of architectural expertise.

7.1 Study Design

In order to answer all the research questions and hypotheses, we designed an experiment with two phases. Our first experiment consists of a comparison between the analysis performance of the traditional approach, using participants who already have a fairly solid knowledge of software engineering, and the RAG enhanced approach, represented by the RAG ATAM tool (see Chapter 6), on a sample model problem. We pay particular attention to the relevance of the results and the time required to perform the analysis using expert rater evaluations.

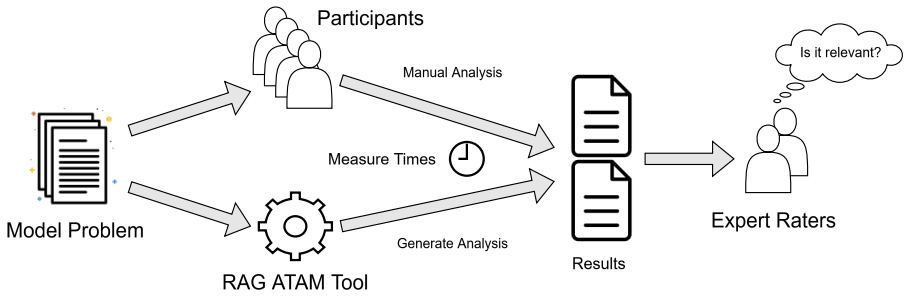

Figure 7.1: Experiment Design for Phase 1

[1] https://zenodo.org/records/15089678?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6ImFlNWFhOTg1LTE1MGQtNDNhNS04NDhmLTdmYTZmYmQ4NWIxNyIsImRh
RcPWub_GUDkKEMzOH2Zh_fJQ02gwibgezuURkWWt0qSteT7ClzMT5KdRBg0B9P3FxDWTUnndSAT-BS0mOhoo7A

The second experiment is a task-based evaluation of the RAG ATAM Tool. For this, we let users who are experienced in the field of software engineering perform an analysis task with the tool, followed by a questionnaire. Our aim is to evaluate the usability and highlight the strengths and weaknesses of the analysis process using RAG.

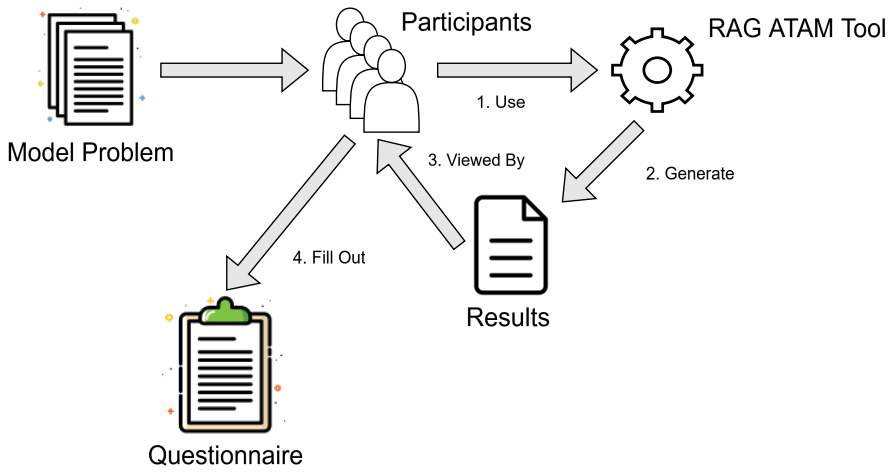

Figure 7.2: Experiment Design for Phase 2

7.1.1 Model Problem used for the Experiment: E-commerce Website

The model problem used in this experiment is an e-commerce website, derived from simplifying and integrating common features and requirements found in real e-commerce platforms. The problem definition is based on existing literature, in particular, Yuan et al. [YF11], which discusses patterns for business-to-consumer e-commerce applications. In addition, we have included insights from the web article "E-Commerce Architecture Design Choices"[2] to ensure relevance to modern system architectures.

Because the focus of this study is on qualitative risk analysis rather than quantitative evaluation, we deliberately did not define strict numerical metrics for the quality attribute criteria. Instead, we used them as descriptive benchmarks to assess the architecture's ability to satisfy key quality attributes.

Given the inherent complexity of the model problem, we opted for a concise presentation that highlights the most relevant aspects, while leaving room for both participants and the RAG-based tool to explore and generate analytical results.

[2] https://medusajs.com/blog/ecommerce-architecture/

7.1.2 System Context

Description: The model problem represents a simple online store that allows users to browse products, add items to a shopping cart, and process orders. The main goals of the system are to support a large number of concurrent users, to ensure a seamless customer experience even during peak traffic events such as Black Friday, and to provide a reliable inventory and order management system to prevent overselling and stock inconsistencies.

System Interactions: The system interacts with various external and internal components to ensure smooth operation. It integrates with third party payment gateways such as Stripe and PayPal to handle secure transactions. It also provides APIs to connect to external inventory management systems, enabling real-time stock synchronisation. Finally, the system facilitates user interaction, allowing customers to browse products, make purchases and submit reviews within the platform.

Technical constraints: The system is required to use specific programming frameworks, with React for the front-end and Spring Boot for the back-end. It must also be hosted on a cloud infrastructure such as AWS or Microsoft Azure to ensure scalability, reliability and deployment flexibility.

7.1.3 Architectural Approach 1

Monolithic Architecture A monolithic architecture consolidates all functionalities—user interface, business logic, and data operations—into a single, tightly integrated application. It is built as one deployable unit, facilitating straightforward development, deployment, and scaling on cloud infrastructure. This architecture is suitable for an online store aiming for operational efficiency and reliable performance under predictable loads.

Architectural Decisions

1. Unified Codebase and Deployment:
    - Develop the application as a single project using React for the front end and Spring Boot for the back end.
    - Enable horizontal scaling with a load balancer (e.g., AWS Elastic Load Balancer) to distribute traffic during high-demand periods such as Black Friday.
2. Database Management:
    - Implement a centralized relational database (e.g., PostgreSQL or MySQL) to manage data for users, products, orders, and payments.
    - Use read replicas to handle heavy read workloads and implement caching (e.g., Redis) for frequently accessed data like product listings or user profiles.
3. External Integrations:
    - Utilize APIs to connect with external inventory management systems for real-time stock synchronization, minimizing overselling or stockouts.
    - Integrate with third-party payment gateways such as Stripe and PayPal using their SDKs or APIs to process secure payments, ensuring full compliance with PCI-DSS standards.

4. Scalability and Security:
    - Configure autoscaling to dynamically adjust resources based on traffic patterns, ensuring consistent performance during surges.
    - Enforce HTTPS for all communication, implement role-based access control (RBAC) for authorization, and secure sensitive data using encryption both in transit and at rest.
    - Use monitoring tools (e.g., AWS CloudWatch, Datadog) to track system performance and detect anomalies early.

Architectural Views The views for the Monolithic Architecture are depicted in Figure 7.3, Figure 7.4, and Figure 7.5.

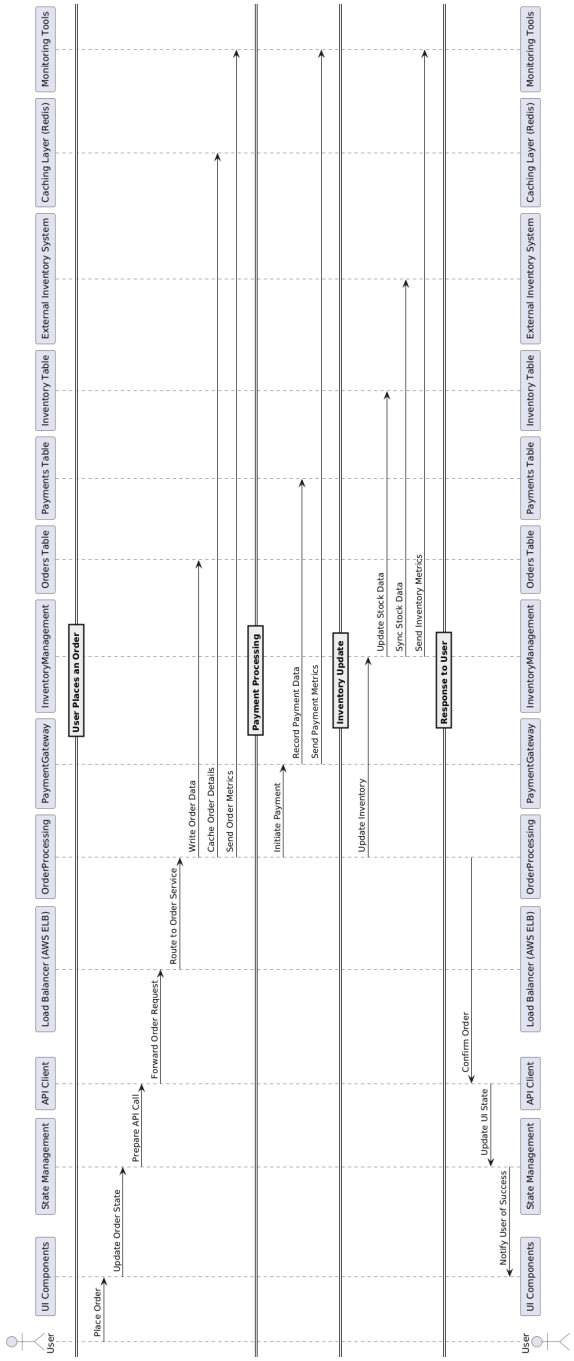

Figure 7.3: Process View for the Monolithic Architecture Approach

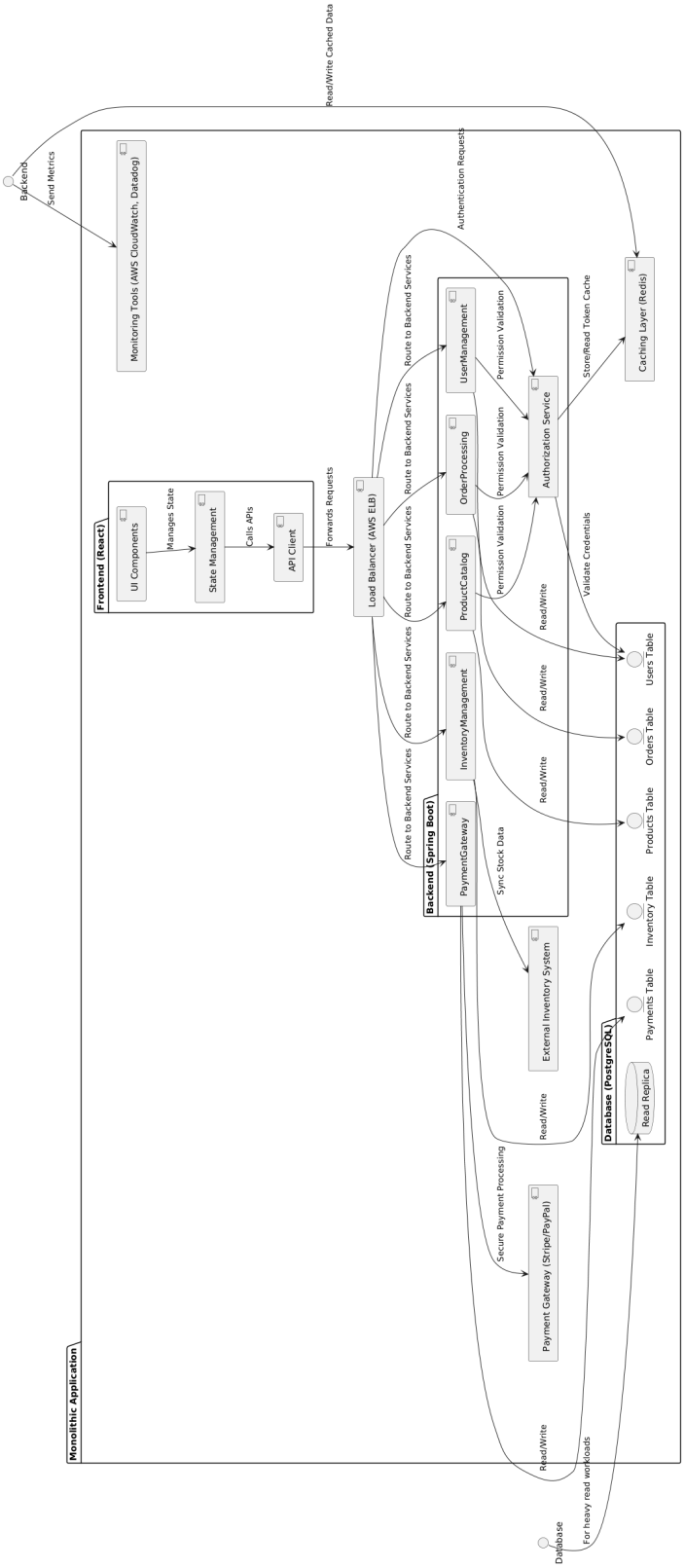

Figure 7.4: Development View for the Monolithic Architecture Approach

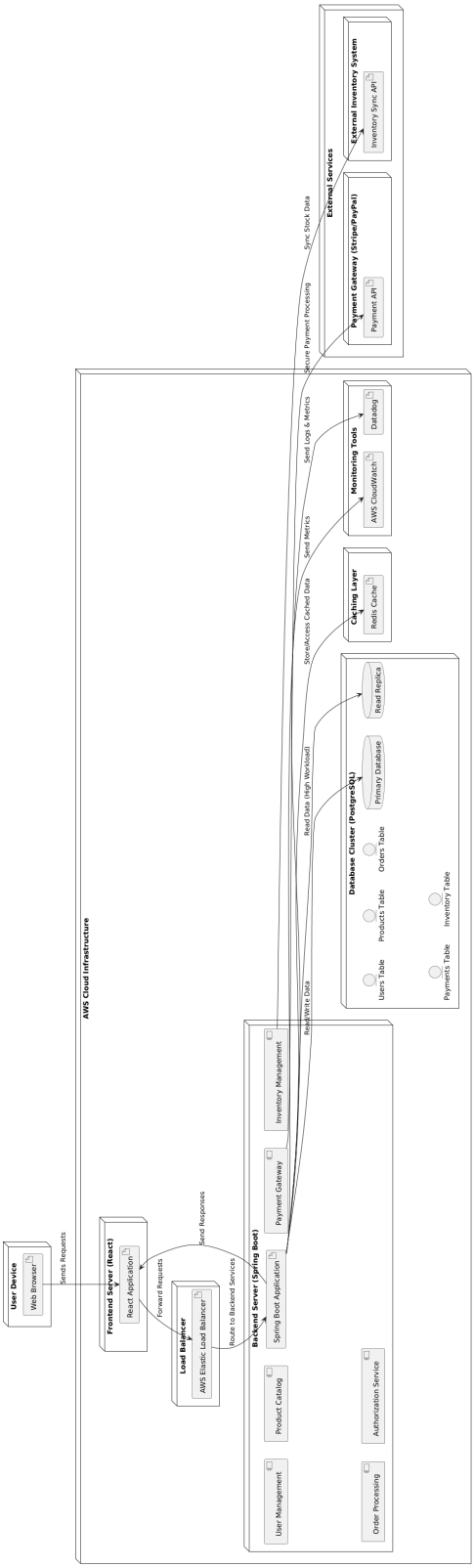

Figure 7.5: Physical View for the Monolithic Architecture Approach

7.1.4 Architectural Approach 2

Microservices Architecture In the microservices architecture, the system is divided into multiple small, independent services, each focusing on a specific functionality, such as user management, product catalog, or order processing. These services communicate with each other through lightweight protocols (e.g., REST or messaging systems like Kafka). This architecture enhances scalability, flexibility, and fault isolation.

Architectural Decisions

1. Service Separation and Responsibilities:
    - User Service: Handles user authentication, authorization, and profile management.
    - Product Service: Manages the product catalog, including browsing, search, and inventory updates.
    - Order Service: Manages shopping cart operations, order placement, and tracking.
    - Payment Service: Handles payment processing, including integration with third-party gateways (e.g., Stripe, PayPal).
    - Inventory service: Track stock levels, Update stock from order events, Notify low stock, Provide stock APIs.
2. Infrastructure and Deployment:
    - Each service is containerized using Docker for consistent deployment environments.
    - Kubernetes (K8s) is used for container orchestration, ensuring scalability, self-healing, and load balancing.
    - API Gateway: A centralized entry point for routing client requests to the appropriate microservices.
3. Communication and Integration:
    - Synchronous Communication: Use REST APIs for real-time client interactions (e.g., retrieving product data).
    - Asynchronous Communication: Use an event-driven architecture with Apache Kafka for inter-service communication (e.g., order events, inventory updates).
    - Database Per Service: Each service has its own database (e.g., PostgreSQL for relational data or MongoDB for NoSQL data) to ensure decoupling and autonomy.
4. Scalability and Security:
    - Implement horizontal scaling at the service level, allowing individual services to scale independently based on their workload.
    - Secure APIs with OAuth 2.0 for authentication and role-based access control (RBAC) for authorization.
    - Use a service mesh (e.g., Istio) for observability, traffic management, and enhanced security.

Architectural Views The views for the Microservices Architecture are depicted in Figure 7.6, Figure 7.7, and Figure 7.8.

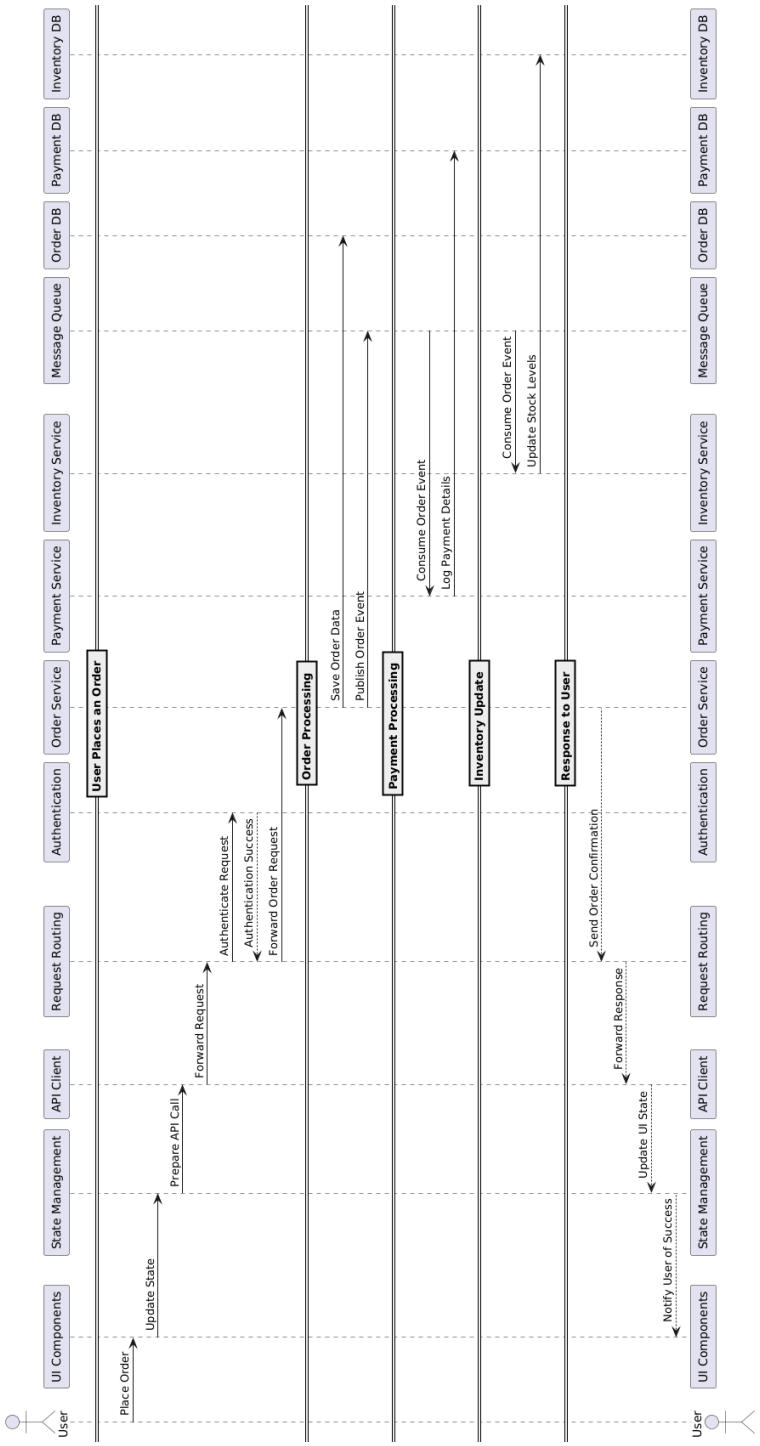

Figure 7.6: Process View for the Microservices Architecture Approach

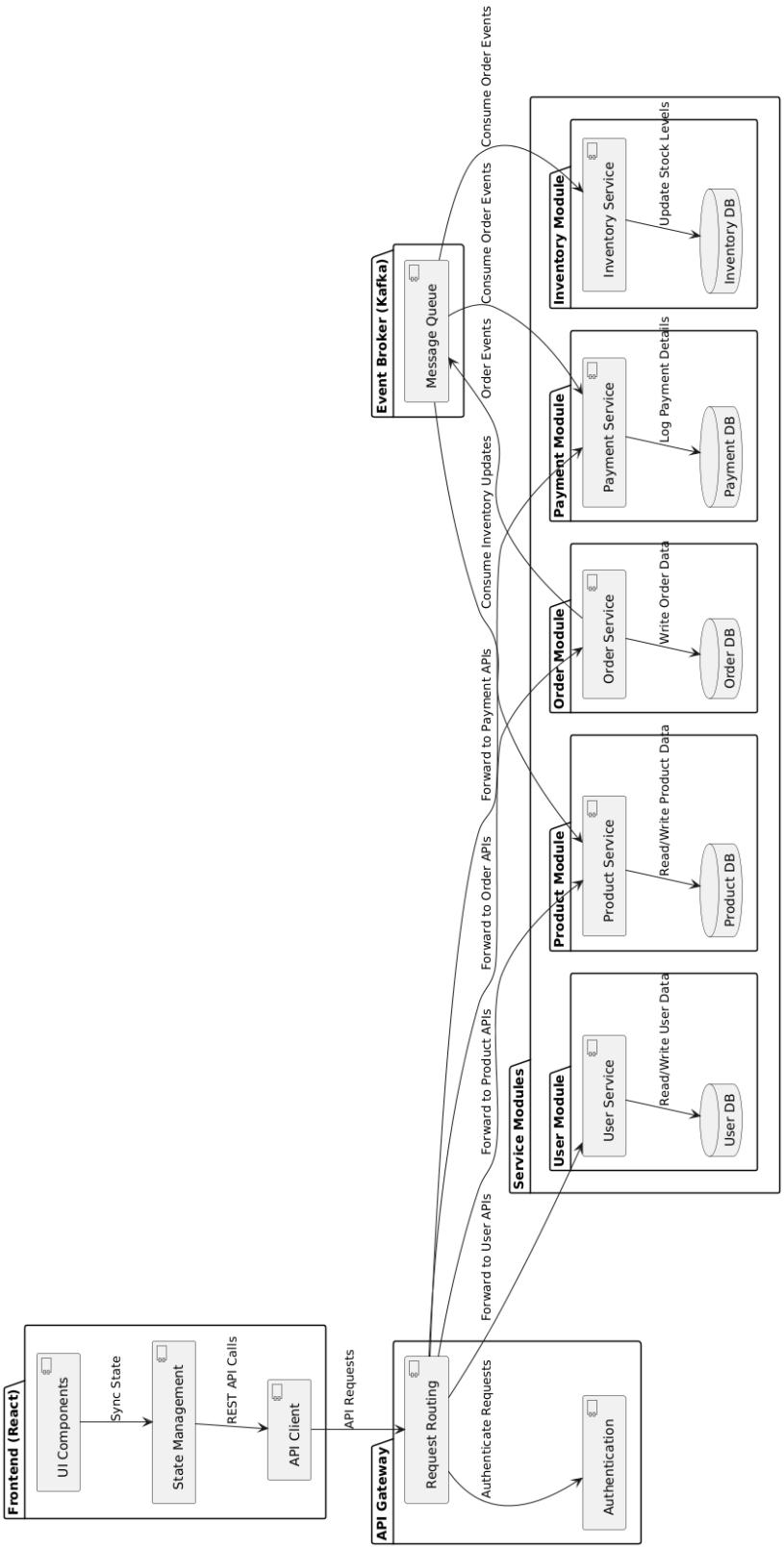

Figure 7.7: Development View for the Microservices Architecture Approach

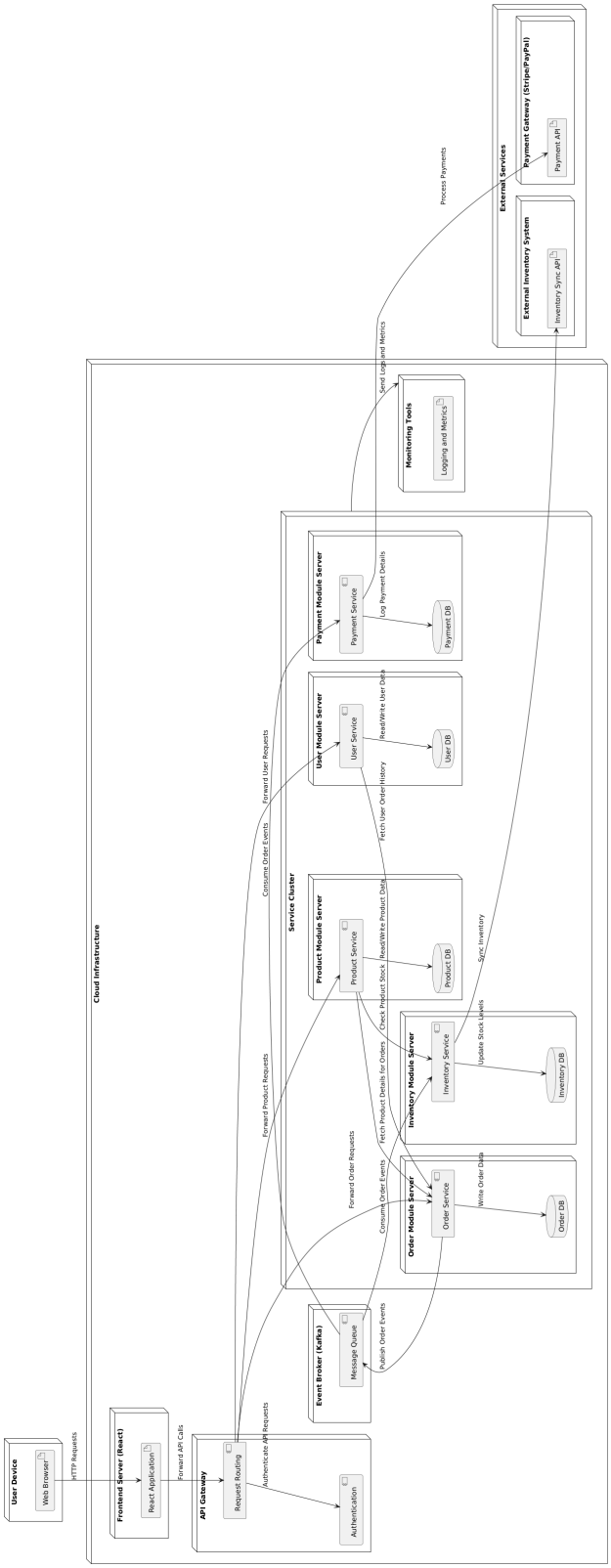

Figure 7.8: Physical View for the Microservices Architecture Approach

7.1.5 Quality Attribute Scenarios

These are the scenarios we are using to analyze the architecture against.

Scenario 1: User Authentication Reliability

Attribute: Reliability
Environment: The system operates in a typical production environment under normal and peak usage conditions.
Stimulus: A sudden spike in user authentication requests due to an external event (e.g., a promotional campaign), coupled with the unavailability of one authentication component or service.
Response: The system ensures uninterrupted authentication services by rerouting requests to backup components or services, maintaining an average response time of under 2 seconds, and ensuring at least 99% of requests are processed successfully.

Scenario 2: Scaling Under Peak Load

Attribute: Scalability
Environment: The system operates in a production environment with variable user traffic, including occasional peak loads triggered by external events.
Stimulus: A sudden surge in concurrent user activity, such as 10x the average traffic within a short time window (e.g., during a flash sale or unplanned event).
Response: The system dynamically scales resources to handle the increased load without degrading response times beyond 3 seconds for 95% of requests, ensuring system stability and maintaining core functionality.

Scenario 3: Order Placement Consistency

Attribute: Consistency
Environment: The system operates in a production environment where multiple users are concurrently placing orders, with some transactions involving high-value items or limited stock.
Stimulus: Simultaneous order placement requests for the same item from multiple users during a high-demand period (e.g., during a limited-time sale or product launch).
Response: The system ensures consistency by processing orders sequentially or using a distributed locking mechanism, ensuring that no duplicate or conflicting orders are processed. All successful orders are confirmed within 5 seconds, and inventory levels are accurately updated in real time across all components.

7.1.6 Quality Attribute Criteria

Quality attribute criteria are descriptive benchmarks used to assess how well a system's architecture supports desired attributes like reliability, scalability, or maintainability based on scenarios and expert judgment rather than precise metrics.

Reliability

- Failure Recovery: The system must detect and handle the unavailability of an authentication component or service within 5 seconds, ensuring continuity by rerouting requests to backup components without impacting user experience.
- Request Success Rate: At least 99% of authentication requests must be successfully processed under both normal and peak conditions, including during partial component failures.
- Response Time Consistency: The system must maintain an average response time of under 2 seconds for 95% of requests during peak traffic and recovery scenarios, ensuring reliability in high-demand situations.

Consistency

- Sequential Order Processing: The system must process simultaneous order placement requests for the same item in a sequential manner, ensuring no conflicting or duplicate orders occur, even under peak load conditions.
- Real-Time Inventory Updates: Inventory levels must be updated consistently and accurately across all components within 2 seconds of an order confirmation to prevent overselling or stock inconsistencies.
- Order Confirmation Accuracy: All successfully processed orders must be confirmed to the user within 5 seconds, with no discrepancies between the confirmation details and the actual processed transaction.

Scalability

- Dynamic Resource Allocation: The system must dynamically allocate additional computational resources within 5 seconds of detecting a traffic surge, ensuring sustained service availability during peak loads.
- Performance Stability: Response times must not exceed 3 seconds for 95% of user requests, even when traffic spikes to 10x the average load.
- Core Functionality Retention: All core functionalities (e.g., login, checkout) must remain operational without degradation or failure during peak load scenarios, maintaining user experience across all services.

7.1.7 Phase 1: Comparing Manual and RAG ATAM Analysis

To compare manual and RAG-enhanced ATAM analyses, we designed a model problem consisting of a set of quality attribute scenarios and architectural approaches (see Section 7.1.1). Participants were tasked with analysing pre-defined architectural approaches, each of which consisted of key architectural decisions evaluated against several relevant scenarios. We used the same model problem for both manual and RAG-based analyses, with the RAG ATAM tool representing a semi-automated qualitative architecture analysis.

Participants in the manual analysis were students in the advanced stages of their bachelor's degree in computer science or software engineering. These students had already completed foundational courses in software engineering, ensuring they possessed the necessary background to perform the analysis. Their solutions were expected to provide a realistic basis for comparison with the automated RAG-based method, even if they lacked the depth and rigor of professional analyses.

After a brief introduction to ATAM and task instructions, the students performed a manual risk analysis based on pre-defined scenarios. Their aim was to identify potential architectural risks, trade-offs, and sensitivity points. Each student was given one hour to complete the task.

Given the large number of possible combinations of architectural approaches, choices, and scenarios, it was impractical to analyze all possibilities within the time limit. Instead, each student focused on two architectural decisions within a given architectural approach.

The model problem consisted of:

- 2 architectural approaches
- 4 major decisions per approach
- 3 scenarios

This resulted in 24 different analyses (2 approaches $\times$ 4 decisions $\times$ 3 scenarios). The results were then recorded in a table format, with each row representing a specific combination of architectural approach, decision, and scenario, and each column representing a specific aspect of the analysis (risk, trade-off, sensitivity point).

For the RAG-based analysis, we used the RAG ATAM tool to automate the risk analysis process based on the same model problem. The tool was provided with the same set of architectural approaches and scenarios. An exception was that we provided the tool with the architectural views of the approaches in PlantUML format so that it could use them for the analysis. To simulate domain knowledge similar to that of a software architect, the tool was provided with a collection of scientific articles (PDF format) on different software architectures, as well as their strengths and weaknesses. The model problem focused specifically on monolithic and microservices architectures.

Once the tool had generated the risks, trade-offs, and sensitivity points, the results were compiled into a report (see Table 7.1) for evaluation. To improve the completeness of the output, we generated the analysis twice, combining both results into a single report. This decision was based on the observation that the results did not differ significantly beyond the second iteration.

Scenario Approach Decision	**Risks**	**Tradeoffs**	**Sensitivity Points**
Scenario 1 Approach 1 Decision 1 Time RAG: Time Manual:	...	...	...
Scenario 1 Approach 1 Decision 2 Time RAG: Time Manual:	...	...	...
...	...	...	...
Scenario X Approach Y Decision Z Time RAG: Time Manual:	...	...	...

Table 7.1: Template table for the analysis

After the students of the manual approach and the RAG ATAM tool have completed their manual analyses, we invite experienced people with moderate architectural knowledge and have them independently rate each aspect of the report using a predefined rubric based on a Likert scale to assess the relevance of the identified risks, trade-offs, and sensitivity points. For relevance, we provided a 5-point Likert scale ranging from 1 (not relevant) to 5 (highly relevant). To remove evaluator bias, we anonymized the submissions and randomized the order in which they were presented. In addition, to gather qualitative feedback, we asked the raters to comment on the relevance of each aspect.

In order to truly assess relevance, the following criteria and questions were used:

- Can the identified aspect realistically occur in a real scenario? If not, it is not relevant.
- Can the identified aspect be mitigated by practical architectural adjustments, or is it an inherent limitation of the approach? If it can be mitigated, then it is relevant.
- Does the identified aspect make sense contextually?
- Does the identified aspect correspond to the given (scenario, decision) tuple?

This expert evaluation should serve as a reliable standard for benchmarking manual analyses against those generated by RAG-enhanced LLM models.

At the end of the exercise, we will have a report with the results (time spent per analysis and aspects mentioned) of the manual and RAG-enhanced analyses, as well as the expert ratings for each of the aspects mentioned.

7.1.8 Phase 2: Task-based Evaluation

This phase will focus on evaluating the usability of using RAG for ATAM analysis. In particular, we want to know how well the idea of using a RAG-enhanced tool for ATAM is understood and how the quality of the results is perceived by the participants.

For this purpose, we had the participants perform a task-based evaluation of the RAG ATAM tool, which was developed to represent the RAG model and the ATAM method. They were given a brief introduction to the tool and then asked to carry out a series of analyses using the tool by entering user input into the text fields provided by the tool. To do this, the supervisor provided the participants with a subset of the model problem, consisting of a few rows of the table from Table 7.1.

After the task, participants were asked to complete a questionnaire to evaluate the usability of the tool. The questionnaire was a combination of quantitative and qualitative data collection. For quantitative data, we used a 5-point Likert [Rob23] scale ranging from 1 (strongly disagree) to 5 (strongly agree). For qualitative data, we used open-ended questions to capture participants' thoughts and suggestions for improvement.

Regarding the participants, we used the same students as in the first phase, i.e. they already have basic knowledge about software engineering and software architecture. After the task-based evaluation, we collected the results from the questionnaire for further evaluation.

For the quantitative evaluation (Likert scale) of usability we used the following questions for the questionnaire:

- The tool was easy to use.
- The tool was intuitive.
- I was able to complete my tasks efficiently using the tool.
- I thought there was too much inconsistency in this tools' results (reversed scoring).
- The tool effectively identified trade-offs in the architecture.
- The tool provided actionable insights on risks in the architecture.
- The tool was helpful in identifying sensitivity points.
- The tool supported a comprehensive analysis of the architecture.
- The tool's outputs aligned with my expectations as an software engineer.

For the qualitative evaluation, we used the following open ended questions:

- What aspects of the tool's usability did you find most effective?
- What challenges or frustrations did you encounter while using the tool?

- What do you consider the greatest strength of using the RAG ATAM Tool for architecture analysis?
- What do you consider the greatest weakness of the tool?
- What would you suggest to improve the tool?

7.2 Results

In this section we present the results of the evaluation, focusing on the comparison between manual and RAG enhanced analysis and the usability assessment of the RAG ATAM tool.

7.2.1 Phase 1: Comparing Manual and RAG ATAM Analysis

In the first phase of the evaluation, we compared the manual and RAG enhanced analyses of the model problem. After performing both manual and RAG enhanced analyses, we collected results and expert ratings for each architectural aspect evaluated. Due to the large amount of data and the limited number of expert raters, we narrowed the scope of the model problem to facilitate a clear comparison, resulting in an evaluation that included one scenario, two architectural approaches, and four decisions, for a total of 8 different analyses.

In total, both the manual and RAG-enhanced analyses identified 22 risks, 21 sensitivities, and 20 trade-offs (63 issues identified in total). After the expert evaluation, we received a relevance rating for each issue, which we used to compare the manual and RAG-enhanced analyses. We were able to recruit two expert raters with moderate to advanced knowledge of software architecture and ATAM for the evaluation, giving us a total of 126 ratings (2 raters $\times$ 63 aspects).

Risk points	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Expert rater 1 / RAG	5	5	5	5	–	5	5	–	5	5	–	5	5	–	5	1	2	1	–	2	3	1
Expert rater 2 / RAG	5	4	4	4	–	1	5	–	3	5	–	5	4	–	5	1	4	3	–	3	2	1
Expert rater 1 / Manual	5	–	–	–	5	–	–	5	–	–	2	–	–	5	–	–	–	–	2	–	–	–
Expert rater 2 / Manual	5	–	–	–	3	–	–	1	–	–	5	–	–	1	–	–	–	–	1	–	–	–

Table 7.2: Relevance Ratings for RAG-Based and Manual Analyses regarding Risks

Sensitivity Points	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21
Expert rater 1 / RAG	2	5	3	3	5	–	1	5	–	2	2	2	–	2	5	–	3	–	2	–	5
Expert rater 2 / RAG	4	5	5	1	3	–	1	2	–	3	1	3	–	3	4	–	4	–	4	–	2
Expert rater 1 / Manual	2	–	–	3	–	1	–	–	1	–	–	–	1	–	–	2	–	2	–	2	–
Expert rater 2 / Manual	4	–	–	1	–	1	–	–	1	–	–	–	1	–	–	1	–	4	–	1	–

Table 7.3: Relevance Ratings for RAG-Based and Manual Analyses regarding Sensitivity Points

Tradeoffs	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20
Expert rater 1 / RAG	–	4	1	–	2	5	–	5	1	1	5	5	2	–	5	–	5	5	5	5
Expert rater 2 / RAG	–	5	4	–	5	4	–	2	3	5	4	5	3	–	3	–	5	4	4	4
Expert rater 1 / Manual	2	–	–	2	–	–	2	–	1	–	5	–	–	1	–	2	–	–	–	–
Expert rater 2 / Manual	1	–	–	1	–	–	1	–	3	–	4	–	–	5	–	2	–	–	–	–

Table 7.4: Relevance Ratings for RAG-Based and Manual Analyses regarding Tradeoffs

Architectural Approach Key Decision	Manual approach	RAG iteration 1	RAG iteration 2
Monolithic Architecture Unified Codebase and Deployment	19:23 min	1:19 min	1:11 min
Monolithic Architecture Database Management	8:12 min	1:24 min	1:14 min
Monolithic Architecture External Integrations	6:32 min	1:42 min	1:44 min
Monolithic Architecture Scalability and Security	20:45 min	1:11 min	1:36 min
Microservices Architecture Service Separation and Responsibilities	13:51 min	1:15 min	1:07 min
Microservices Architecture Infrastructure and Deployment	15:39 min	1:12 min	1:25 min
Microservices Architecture Communication and Integration	15:11 min	1:06 min	1:07 min
Microservices Architecture Scalability and Security	3:58 min	1:20 min	1:10 min

Table 7.5: Measured times to Complete a Full Decision Analysis for the User Authentication Reliability Scenario.

Evaluating the Measured Times

To provide a comprehensive overview of the time efficiency of the different approaches, we calculated various statistical measures, including the **count**, **mean**, **standard deviation**, **minimum**, **maximum** and **interquartile range (IQR)** for the manual and RAG-enhanced analyses. We can generate a boxplot to visualize the distribution of time taken for the manual and RAG-enhanced analyses, as shown in Figure 7.9.

Approach	Count	Mean	Std Dev	Min	Q1 (25%)	Median (50%)	Q3 (75%)	Max	IQR
Manual Approach	8	**776**	365	238	467	871	995	1245	528
RAG Iteration 1	8	**78**	11	66	71	77	81.	102	9
RAG Iteration 2	8	**79**	14	67	69	72	87.	104	18

Table 7.6: Statistical Summary of Time Efficiency Across Approaches.

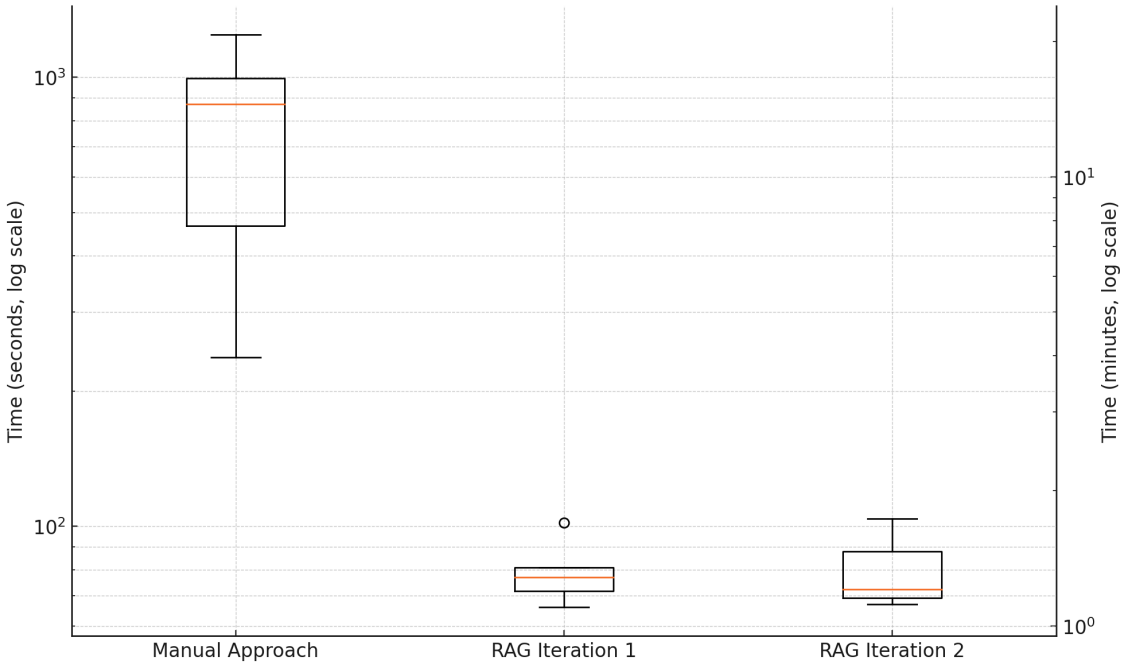

Figure 7.9: Boxplot of the Time needed for the Manual and RAG-enhanced Analyses.

Looking at Figure 7.9, the results show a significant improvement in time efficiency with the RAG-enhanced approach compared to the manual method, given the given hardware configuration and code implementation of the RAG. ATAM tool. As shown in Table 7.6, the manual approach took an average of approximately 13 minutes (776 seconds), while RAG Iteration 1 and 2 completed the same tasks in 1:19 min (78 seconds) and 1:20 min (79 seconds) respectively, an approximate *90% reduction in processing time*.

The manual approach also showed high variability, with a standard deviation of over six minutes (365 seconds), possibly reflecting differences in decision complexity and participant skill. In contrast, the RAG iterations were much more stable, with standard deviations of around 11 seconds (iteration 1) and 14 seconds (iteration 2), indicating *predictable execution times under normal circumstances*.

Execution times for the manual approach ranged from 4 to 21 minutes (238 to 1245 seconds), while both RAG iterations remained within a narrow band of just over 1 to under 2 minutes (66 to 102 seconds), demonstrating *greater consistency in execution times*. The interquartile range (IQR) supports this further, resulting in 9 minutes (528 seconds) for the manual approach compared to 9

seconds (iteration 1) and 18.50 seconds (iteration 2) for the RAG approach, highlighting a general *reduction in variability in processing time*. Furthermore, the minimal difference between the two RAG iterations suggests the time consistency and reliability of the tool over multiple runs, with *iteration 1 being slightly faster on average*.

Focusing on the model problem, the results generally indicate a solid improvement in time efficiency, with consistent and predictable performance compared to the manual approach. The manual approach generally relies heavily on the complexity of the decisions and the skill of the participants, which can lead to high variability in execution times, whereas the RAG doesn't have this subjective factor. However, it is important to consider not only the time efficiency but also the quality of the results generated by the RAG approach in order to provide a comprehensive evaluation of the tool.

Evaluating Relevance Scores

Two expert raters evaluated the relevance of identified risks, sensitivity points, and tradeoffs produced by both the manual and RAG-enhanced ATAM approaches using a 5-point Likert scale (1 = not relevant, 5 = highly relevant). We analyzed the results by using descriptive statistics, including the mean, median, and standard derivation, to compare the relevance of the results generated by the two approaches.

Due to the small number of data points, the findings are rather descriptive than statistically conclusive. Therefore, we interpret these results as indicative trends rather than definitive proof.

The mean relevance score provides an overview of expert ratings for the results from each approach. The median relevance score indicates the central tendency of the ratings, while the standard deviation measures the variability in the ratings.

The results are shown in Table 7.7. Across all categories, the RAG enhanced approach tended to receive higher mean and median scores than the manual method, especially for sensitivity points and tradeoffs (as shown in Figure 7.10). The variability in ratings (as measured by standard deviation) differed between raters and categories reflecting different levels of consistency in expert judgment.

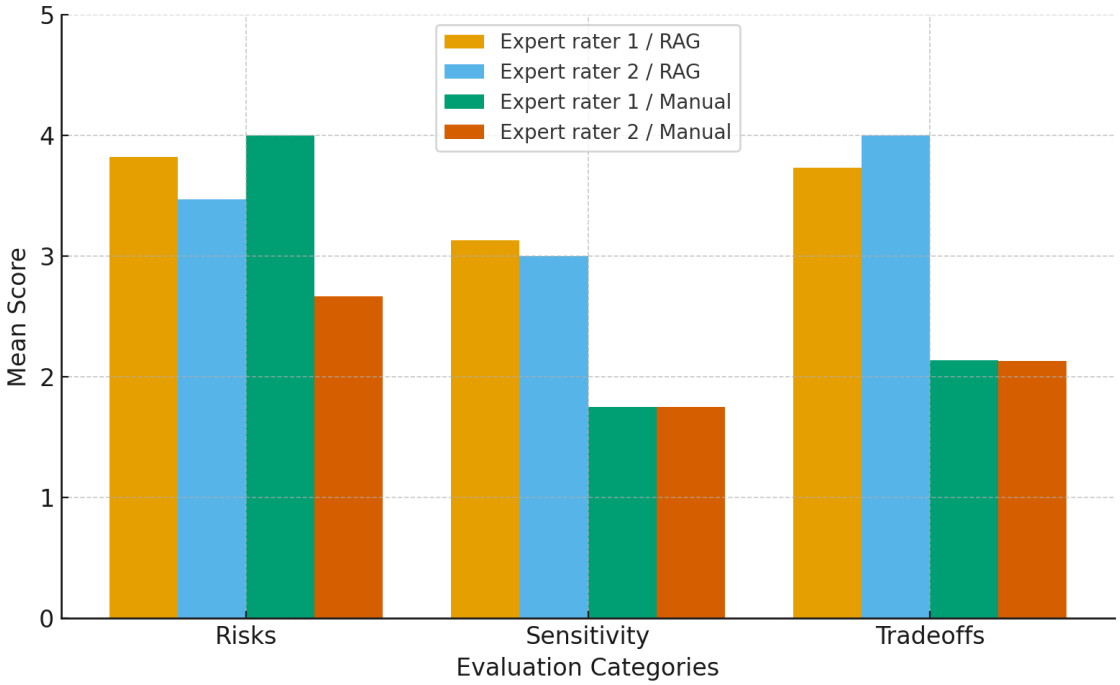

Figure 7.10: Mean relevance scores for Risks, Tradeoffs, and Sensitivity Points

Risks	Count	Mean	Std Dev	Min	Q1	Median	Q3	Max	IQR
Expert rater 1 / RAG	17	3.82	1.70	1.00	2.00	5	5.00	5.00	3.00
Expert rater 2 / RAG	17	3.47	1.46	1.00	3.00	4	5.00	5.00	2.00
Expert rater 1 / Manual	6	4.00	1.55	2.00	2.75	5	5.00	5.00	2.25
Expert rater 2 / Manual	6	2.67	1.97	1.00	1.00	2	4.50	5.00	3.50
Sensitivity Points	Count	Mean	Std Dev	Min	Q1	Median	Q3	Max	IQR
Expert rater 1 / RAG	15	3.13	1.46	1.00	2.00	3	5.00	5.00	3.00
Expert rater 2 / RAG	15	3.00	1.36	1.00	2.00	3	4.00	5.00	2.00
Expert rater 1 / Manual	8	1.75	0.71	1.00	1.00	2	2.00	3.00	1.00
Expert rater 2 / Manual	8	1.75	1.39	1.00	1.00	1	1.75	4.00	0.75
Tradeoffs	Count	Mean	Std Dev	Min	Q1	Median	Q3	Max	IQR
Expert rater 1 / RAG	15	3.73	1.75	1.00	2.00	5	5.00	5.00	3.00
Expert rater 2 / RAG	15	4.00	0.93	2.00	3.50	4	5.00	5.00	1.50
Expert rater 1 / Manual	7	2.14	1.35	1.00	1.50	2	2.00	5.00	0.50
Expert rater 2 / Manual	7	2.43	1.62	1.00	1.00	2	3.50	5.00	2.50

Table 7.7: Descriptive Statistics for the Relevance Scores

Risk Ratings: The risk ratings show mixed results, with the manual ATAM approach receiving slightly higher average scores (Mean = 4.00) than the RAG approach (Mean = 3.82) from Expert Rater 1, with both having a median relevance score of 5. Expert Rater 2 showed the opposite trend, rating RAG generated risks higher (Mean = 3.47) than manually generated risks (Mean = 2.67). The median of the scores shows that the second rater also tended to give a much lower relevance score for the manual approach (median = 2) compared to the RAG approach (median = 4). In terms of standard deviation, all scores show moderate variability, with Expert Rater 1 being slightly more consistent in scoring manual results (1.55) than RAG results (1.70), while Expert Rater 2 showed much greater variability in scoring manual results (1.97) compared to RAG (1.46). This indicates that both approaches consist of results with quite a large variability in relevance scores, meaning that both RAG and the manual approach have a mix of relevant and irrelevant results according to the subjective opinion of the expert raters.

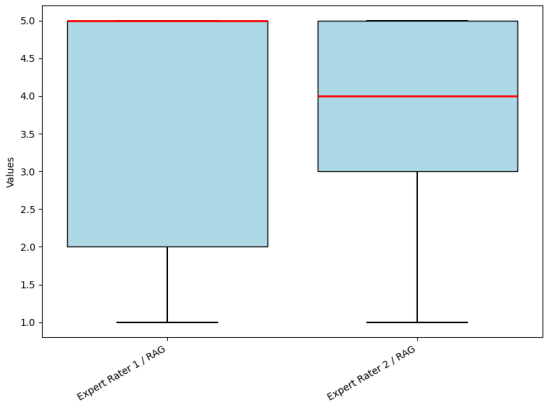

Figure 7.11: Relevance Score Boxplot for Risks (RAG approach)

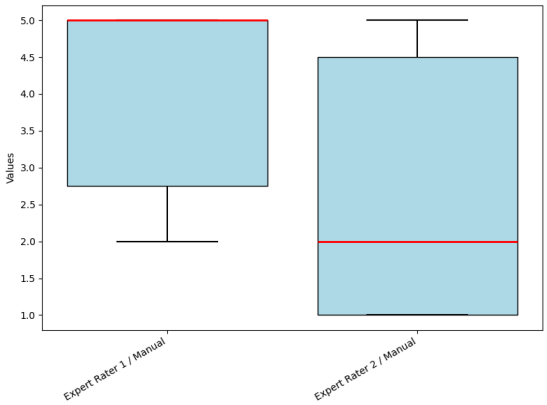

Figure 7.12: Relevance Score Boxplot for Risks (Manual approach)

Sensitivity Point Ratings: The greatest divergence between RAG-enhanced and manual ATAM results is observed in the sensitivity point ratings. Looking at the means, both expert raters rated the manual ATAM approach much lower (mean = 1.75) than the RAG enhanced approach (mean = 3.13 for Expert Rater 1 and 3.00 for Expert Rater 2). The medians confirm this pattern, with manual results having medians of 2 (Expert Rater 1) and 1.00 (Expert Rater 2), while RAG-enhanced results had higher medians of 3 (both raters). Interestingly, Expert 1's ratings for the manual approach were more consistent (standard deviation = 0.71) than for the RAG approach (standard deviation = 1.46), suggesting that the manual approach tends to consistently give low relevance scores. Expert Rater 2, on the other hand, showed similar variability for both approaches (manual = 1.39, RAG = 1.36), suggesting approximately the same level of variability in relevance scores for both approaches, although the relevance scores for the manual approach were lower on average.

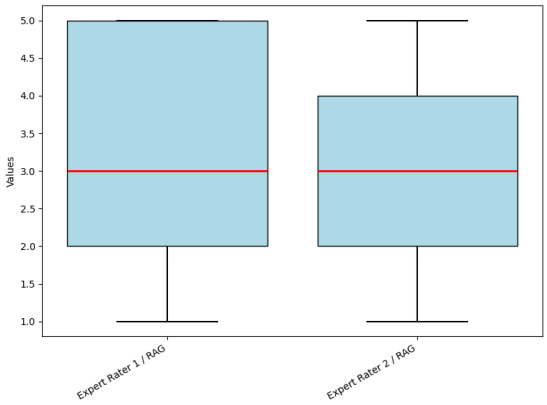

Figure 7.13: Relevance Score Boxplot for Sensitivity Points (RAG approach)

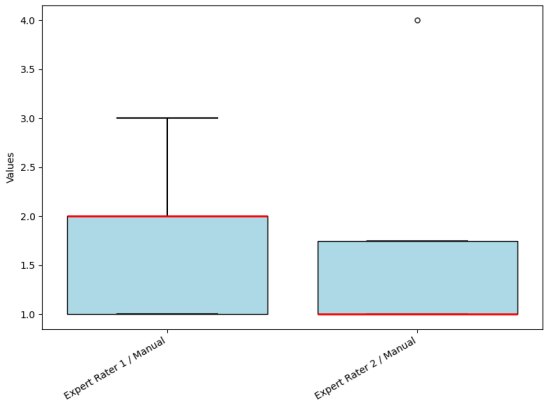

Figure 7.14: Relevance Score Boxplot for Sensitivity Points (Manual approach)

Tradeoff Ratings: A similar pattern to the sensitivity score relevance scores is observed for the tradeoff scores. Again, the RAG enhanced results received higher mean scores, with Expert Rater 1 assigning a mean score of 3.73 to the RAG results and 2.14 to the manual results, while Expert Rater 2 rated the RAG results higher (mean = 4.00) than the manual results (mean = 2.43). The median scores combined with the mean also show a tendency towards higher ratings, with the RAG results having higher medians of 5 (Expert Rater 1) and 4 (Expert Rater 2) compared to the manual results, which had medians of 2 (Expert Rater 1) and 2 (Expert Rater 2). Notably, Expert Rater 2's RAG ratings had a very low variability (standard deviation = 0.93), indicating a rather consistent assessment of the relevance of the generated results according to Expert Rater 2. Expert Rater 1, on the other hand, had a slightly higher standard deviation for the RAG ratings (1.75) compared to the manual ratings (1.35), suggesting more variation in the relevance of the different trade-offs considered in the RAG enhanced results. However, the overall trend is that the RAG-enhanced results were rated as more relevant when considering the mean and median scores.

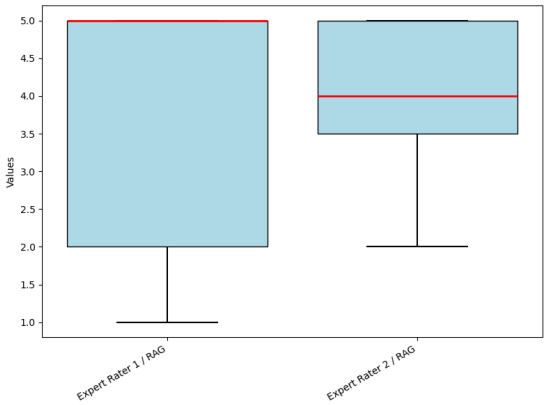

Figure 7.15: Boxplot of the Relevance Ratings for Tradeoffs (RAG approach)

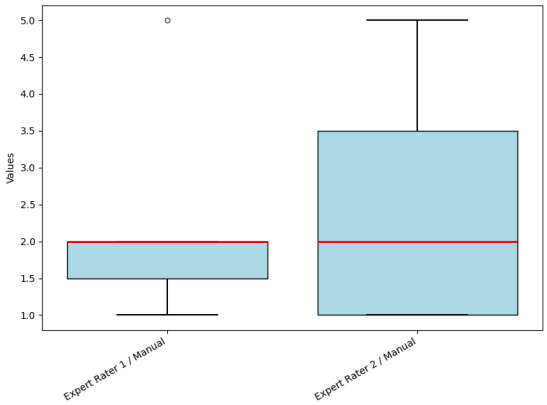

Figure 7.16: Boxplot of the Relevance Ratings for Tradeoffs (Manual approach)

Summary of Findings:

- RAG-enhanced ATAM results generally received higher mean scores than manual ATAM results, particularly for sensitivity and trade-off ratings.
- The mean and median scores for both expert raters support this trend in these two categories, with RAG consistently receiving higher central relevance scores than the manual method.
- The risk ratings showed mixed results, with Expert Rater 1 rating the manual results higher than the RAG results, while Expert Rater 2 rated the RAG results higher than the manual results. However, even in this category, the mean scores of the RAG approach consistently showed rather high relevance (3.47 and 3.82).
- In terms of standard deviation, both approaches showed a rather high variability in relevance scores, indicating that both approaches consist of a mixture of relevant and irrelevant results according to the subjective opinion of the expert raters.

Overall, RAG enhanced results were generally rated as more relevant than manual results. However, due to the limited sample size, these results should be interpreted with caution and further research is needed to confirm these trends.

Comments of Expert Raters

To gain a deeper understanding of the experts' assessments, we collected qualitative feedback on the relevance of the risks, trade-offs and sensitivities identified in the architecture analysis. From this feedback, we identified key themes that highlighted strengths and weaknesses in both the manual and RAG-enhanced analyses.

Clarity The manual ATAM approach often presented risks and tradeoffs in a short and terse manner, which sometimes led to vague phrasing. For example, a manually detected tradeoff **(T1)** was listed as:

> "Reliable authentication (quick response time for user) vs. Secure authentication (deeply integrated, little flexibility for new security standards)."

Expert 1 found this wording confusing, one questioned why "reliable" was equivalent to "quick", noting that authentication mechanisms should focus on verifying user identity correctly and securely rather than speed alone, making the tradeoff unclear. Such brief X vs. Y phrasing lacks clarity about the nature of the conflict since further explanation is needed to understand the tradeoff. Therefore, both experts rated this tradeoff as irrelevant (2 and 1).

Another manual detected tradeoff **(T7)** and risk **(R11)** confirming the tendency for terse phrasing was:

> "Secure, performant API vs. Pricing"

In this case, Expert Rater 2 mentioned that the tradeoff was kinda relevant, but the bullet point was not specified for the architectural context, leading to a low relevance rating of 2 and 1.

> "No proper scaling technology mentioned, auth service could fail if not properly scaled."

The risk was rated low by the first rater, arguing that mentioning scaling without specifying a particular scaling mechanism is not meaningful.

Regarding some sensitivity points in the manual approach, the expert raters identified some relevant areas of concern but failed to clearly articulate how or why that aspect was sensitive. This resulted in feedback about sensitivity points being too general or lacking specificity, making them less actionable. For example, the following sensitivity point **(S18)** was identified:

> "Communication between API client and Request Routing, since response time depends on it"

This issue is valid for both raters, especially for Expert Rater 2, who rated this point with a relevance score of 4. However, Expert Rater 1 found this sensitivity point unclear, as it doesn't specify what property of the communication is sensitive and affects response time, leading to a relevance score of 2.

But some participants still provided clear results, for example:

> **(R5)** "Security of authentication data stored in same DB with other data load of other functionalities using DB impacts accessibility."

> **(R1)** "Unavailability of single component may result in unavailability of whole system." (also provided by the RAG generated results)

This kind of clear and structured explanation is probably strongly dependent on the skill level and writing style of the participants, as the manual approach showed a lot of variation in the quality of the results.

Generally, the results of the manual approach often assumed that the reader could infer the meaning, leading to feedback about unclear intentions or missing context. This lack of clarity was a recurring theme in the manual approach, as the expert raters found many points to be too vague or lacking specificity to be actionable.

In contrast, the RAG enhanced approach generally generated more explanatory results. They were mostly written in full sentences that explicitly linked cause and effect. For instance, one RAG-generated tradeoff **(T2)** explains:

> "While monolithic architecture simplifies development and deployment, it trades off scalability and fault tolerance. The tight coupling of components makes it harder to isolate failures or scale individual services independently."

This construction clearly communicates the benfit of the decision against its cost, making the tradeoff immediately understandable, therefore receiving a relevance score of 4 and 5. Similarily, in the microservices scenario **(T20)** this reasoning structure was also mentioned:

> "While microservices provide scalability and fault isolation, they introduce complexity in communication and monitoring, which can impact reliability if not properly managed"

For the risks, the RAG approach also provided more detailed and structured results, which were easier to understand. For example, **R15** was explained as:

> "Data Consistency Risks: Each microservice has its own database, which can lead to data inconsistencies if not properly synchronized, especiallyduring peak traffic (RAG)"

This kind of complete explanation was easier for evaluators to follow and showed strong coherence in linking an architectural choice to its ramifications. Overall, the RAG approach's language was more precise and structured, making it easier for the expert raters to understand the relevance of the generated results.

However. there were instances where the RAG approach introduced redundancy by repeating similar points multiple times **(R2-4)**, for example:

- "Lacks fault isolation. Occurrence of bottlenecks or cascading failures throughout the system, affecting overall reliability during peak traffic scenarios."
- "The monolithic approach may lead to reduced system resilience compared to microservices architectures, where failing authentication services can be isolated without affecting the entire system."
- "The interconnection is strong, making it hard to achieve resilience. A failure in one service can influence others because they are part of a single codebase."

According to the Expert 1, all three points describe the same risk and could be condensed into a single statement to enhance clarity. Besides that redundancy, the RAG approach still generates unclear details or tends to lapse into naming a general concern without, even though the explaination provides detail and clarity, for instance:

> **(S19)** "Ensuring reliable event processing and consumption in Apache Kafka is critical for maintaining correct order processing and inventory updates."
>
> **(T3)** "The monolithic approach may lead to reduced system resilience compared to microservices architectures, where failing authentication services can be isolated without affecting the entire system."
>
> **(S3)** "Dependency on the primary database for write operations could become a bottleneck during peak traffic, potentially affecting response times and request success rates"

Both Expert Raters found them acknowledgeable, but without pinpointing the variable, similar to the issues of the manual approach, the mentioned aspect was considered unclear.

But despite that, the RAG approach consistently produced clearer, more structured, and context-rich results that made the implications easier to understand. This improved clarity helped expert raters more reliably assess the relevance of the identified risks, sensitivity points, and tradeoffs. Despite that, further evaluation of the quality and coherence of the results must be considered to generally draw first comparisons about the relevance of the results.

Overall, the RAG-enhanced ATAM analysis tended to produce more uniform, structured content that explicitly linked architectural properties to their impact on system qualities. By contrast, the manual analysis showed more variation in detail and style. Some points were well-explained, while others were mere phrases, causing a lack of specificity and ambiguous statements. This inconsistency sometimes made the manual output feel disjointed or incomplete, probably caused by the different skill levels and writing styles of the participants. But the RAG approach had one flipside, which is redundancy. This can be explained since it likely generates many points rapidly and probably without holistic filtering a human would do.

Quality of the Outputs The expert raters' qualitative feedback and scoring provide a measure of content quality for both approaches. We mainly focused on the results where both raters rated the results similarily, with only one to two relevance scores difference, to identify the most relevant and irrelevant results.

For risks, both manual and RAG enhanced approaches provided reasonable responses according to both raters, for instance:

> **(R1)** "Unavailability of single component may result in unavailability of whole system." (Manual and RAG approach)
>
> **(R2)** "Lacks fault isolation. Occurrence of bottlenecks or cascading failures throughout the system, affecting the overall reliability during peak traffic scenarios." (RAG approach)
>
> **(R5)** "Security of authentication data stored in same DB with other data load of other functionalitiesusing DB impacts accessibility." (Manual approach)
>
> **(R13)** "Security-wise, while HTTPS and RBAC are enforced, a monolithic architecture has a larger attack surface due to its tightly coupled components. A breach in one part can potentially compromise the entire system"

Especially, the expert raters found that, for monolithic architecture, both manual and RAG-enhanced assessments were generally coherent, providing relevance scores between 4 and 5. An important note is, that the RAG enhanced approach provided more risks for the monolithic architecture compared to the manual approach, while still maintaining a high relevance scores, providing more details and context to the risks when using a monolithic architecture. Some examples, where the RAG approach provided more risks are:

> **(R12)** "In a monolithic architecture, scalability is achieved by scaling the entire application, which can lead to over-provisioning and increased costs. This approach may not efficiently handle high traffic spikes as required for an online store during peak events like Black Friday."
>
> **(R7)** "Single point of failure risk due to the centralized nature of the database. If the primary database fails, it could impact all dependent services, including authentication"

Those examples were both rated with relevance scores of 5 by both raters, indicating that the RAG enhanced approach found additional risk points that participants might have missed.

However, for the risk analysis of microservices architecture (R14-22 in Table Table 7.2), both manual and RAG enhanced approaches exhibited coherence issues. Especially, participants for the manual approach struggled to identify relevant risks, with only two risk marked as relevant across both expert raters and 3 risks in total.

> **(R14)** "Single Point of Failure at User Service" (Relevance Scores of 5 and 1)
>
> **(R17)** "Increased attack surface due to multiple microservices, each potentially exposing different endpoints and requiring individual security measures." (Relevance Scores of 2 and 4)
>
> **(R19)** "Synchronous communication, since immediate and average response time cannot be guaranteed" (Relevance Scores of 2 and 1)

R19 and R17 were rated as irrelevant by both raters since doesn't explain which part of the architecture is affected. R14 experienced some degree of disagreement between the raters, which will be discussed in Section 7.2.1, making the relevance of the risk questionable.

Quantity-wise, the RAG approach provided more risks for the microservices architecture compared to the participants from the manual approach, but the relevance of the risks was rated lower on average. In total, only one risk was rated as relevant by both raters:

> **(R15)** "Data Consistency Risks: Each microservice has its own database, which can lead to data inconsistencies if not properly synchronized, especially during peak traffic." (Relevance Scores of 5 and 5)

Meanwhile, the rest were rated as irrelevant or only relevant by one rater, for example:

> **(R16)** "Authentication Overhead: Implementing authentication in each microservice can introduce overhead and complexity, potentially leading to increased latency and resource consumption"

> **(R22)** "Increased complexity in ensuring consistent authentication and authorization across multiple services, potentially leading to unauthorizedaccess or inconsistent policy enforcement"

R16 was marked as 1 by both raters since it doesn't fit the context of the architecture, since the User Service is responsible for authentication and authorization and not each microservice. R22 warns of potential issues but doesn't say how or why they might occur in the given scenario. Without a specific configuration that introduces this complexity, it would seem unfounded.

Compared to risks, the raters found more issues in the sensitivity point analysis for both approaches. Especially the results of the manual approach showed low relevance scores by both raters, with a maximum relevance score of 4 for two sensitivity points by only one rater. As the mean relevance score of 1.75 indicates, the mentioned points by the participants encountered coherence issues. Examples include:

> **(S4)** "Cache invalidation between Redis and DB." (Relevance scores of 3 and 1)

> **(S6)** "Interface between API and backend." (Relevance scores of 1 and 1)

> **(S9)** "Load balancing config" (Relevance scores of 1 and 1)

> **(S18)** "Communication between API client and Request Routing, since response time depends on it." (Relevance scores of 2 and 4)

Expert Rater 1 commented that S4 is unclear, since it doesn't specify which property of the cache invalidation impacts system qualities, but then proposes a the frequency of cache invalidation as a potential sensitivity point, leading to a relevance score of 3. S6 also received low relevance scores by both raters, since it doesn't specify what part of the architecture is affected by the interface between API and backend. S9 was rated as irrelevant by both raters, since it doesn't specify what part of the architecture is affected by the load balancing configuration. S18 was rated as with 2 by Expert Rater 1, even though he comments that the interaction between the API client and the request routing is a relevant point, however, the sensitivity point is not clearly defined, leading to a relevance score of 2.

The RAG-enhanced analysis exhibited fewer coherence issues compared to the manual approach. In fact, the RAG approach even generated results which were rated with high relevance scores by both raters, for example:

> **(S2)** "The lack of modularization in monolithic architectures means that scaling or updating the authentication service requires scaling or redeploying the entire application, which can be resourceintensive and risky during peak traffic." (Relevance Scores of 5 and 5)
>
> **(S5)** "Database scalability limits could affect reliability during traffic surges if the centralized system can't handle increased loads despite read replicas." (Relevance Scores of 5 and 3)
>
> **(S15)** "Token Management: The use of short-lived tokens for authorization requires careful management of token expiration and revocation to maintain security without impacting user experience." (Relevance Scores of 5 and 4)

What makes these sensitivity points more relevant is the explicit connection between architectural properties and their impact on system qualities derived from the model problem, making them more actionable and contextually relevant. Compared to the sensitivity points of the participants from the manual approach, the RAG approach provided more detailed and structured results, which were easier to understand. However, some RAG-generated points still lacked clarity and contextual depth. Some examples are:

> **(S3)** "Dependency on the primary database for write operations could become a bottleneck during peak traffic, potentially affecting response times and request success rates." (Relevance Scores of 3 and 5)
>
> **(S10)** "Real-time inventory updates and order confirmation accuracy are sensitive to the monolithic architecture's ability to handle concurrent requests without introducing delays or inconsistencies, especially during peak traffic scenarios" (Relevance Scores of 2 and 3)
>
> **(S11)** "The system's ability to handle simultaneous order placements consistently relies heavily on the effectiveness of autoscaling and load balancing. Inadequate configuration may lead to performance degradation under high traffic." (Relevance Scores of 2 and 1)
>
> **(S12)** "Sequential Processing in Process View: Handling simultaneous requests sequentially can cause delays, especially during traffic spike." (Relevance Scores of 2 and 3)

Similar to the manual approach, the RAG approach also generated ambiguous details like in S3, where Expert Rater 2 noted that the sensitivity point itself is not clearly defined, even though the text is relevant for discussion. S10, S11, and S12 also suffer from the same issue, where the expert rater noted that the sensitivity point itself is not clearly defined, leading to low relevance scores. Additionally, the lack of clarity was emphasized by Expert Rater 1, who found that the sensitivity points were too general or lacked sufficient detail to be actionable.

But even though the RAG approach generated unclear results, it still provided some clear and structured results compared to the manual approach, making the implications easier to understand. Regarding the quantity of the sensitivity points, the RAG approach provided more sensitivity points than the manual approach in a much shorter time.

The expert raters provided mixed feedback on the tradeoff assessments, with notable differences in how they perceived the tradeoffs identified in the manual and RAG-enhanced ATAM analyses. In general, the RAG enhanced approach received higher ratings for tradeoff relevance compared to the manual approach. Many tradeoffs generated by the RAG system were explicitly framed, clearly describing the opposing factors in a way that made the decision tradeoff evident according to the expert raters. For example, the following tradeoffs clearly articulated the conflicting concerns:

> **(T2)** "While monolithic architecture simplifies development and deployment, it trades off scalability and fault tolerance. The tight coupling of components makes it harder to isolate failures or scale individual services independently." (Relevance Scores of 4 and 5)
>
> **(T6)** "The centralized database approach ensures consistency and ACID compliance, which is beneficial for reliable transaction processing, but may introduce bottlenecks in high-concurrency scenarios" (Relevance Scores of 5 and 4)
>
> **(T12)** "Fault Isolation vs. Complexity: While microservices provide fault isolation, ensuring that a failure in one service does not affect others, this comes at the cost of increased complexity in managing and monitoring multiple services" (Relevance Scores of 5 and 5)
>
> **(T19)** "The use of Istio service mesh introduces additional overhead in setup and management, but provides enhanced security and observability" (Relevance Scores of 5 and 4)

The context specific formulations made the tradeoffs more understandable and actionable, leading to higher ratings from the expert raters. In total, eight out of 15 tradeoffs generated by the RAG approach were rated as relevant (at least a relevance score of 4) by both raters, indicating that the RAG approach was able to identify tradeoffs that were contextually relevant.

In contrast, the manual tradeoff results were often too vague or lacked sufficient detail. Some tradeoffs merely listed two competing concepts without explicitly articulating the tradeoff itself. For instance:

> **(T1)** "Reliable authentication (quick response time for user) vs. Secure authentication (deeply integrated, little flexibility for new security standards)." (Relevance Scores of 2 and 1)
>
> **(T4)** "Security vs. Scalability" (Relevance Scores of 2 and 1)
>
> **(T7)** "Secure, performant API vs. Pricing" (Relevance Scores of 2 and 1)

For T1, the expert found this unclear, questioning why reliability was equated with quick response time, noting that an authentication mechanism should focus on overall correctness rather than speed alone. T4 was rated as irrelevant by both raters due to the lack of specificity, while T7 was rated

as irrelevant by both raters since the bullet point did not specify the architectural property, even though the tradeoff was relevant according to Expert Rater 2. This resulted in lower ratings for tradeoff clarity in the manual approach.

The RAG approach however lacked some clarity and architectural relevance in the tradeoff results, for example:

> **(T13)** "Scalability vs. Resource Management: The ability to scale individual services independently requires careful resource management to avoid over-provisioning or under-provisioning, which can impact reliability." (Relevance Scores of 2 and 3)
>
> **(T9)** "While implementing HTTPS and RBAC provides security benefits, the monolithic architecture may trade off scalability. Autoscaling at the application level can lead to inefficiencies compared to microservices where each service can scale independently." (Relevance Scores of 1 and 3)

For T13, Expert Rater 1 questioned whether scalability and resource management are inherently in conflict, noting that resource management often pertains to cost optimization rather than architectural scalability. T9 was rated as rather irrelevant on average by both raters, because it combines general security practices with a loosely connected scalability concern, making the tradeoff unclear.

Nevertheless, the RAG approach provided more structured and context-rich results that made the implications easier to understand, even though some tradeoffs were redundant or lacked clear links to sensitivity points.

Summary of Findings The qualitative evaluation of the expert feedback revealed several key differences in the quality, quantity, structure, and clarity of outputs between the manual and RAG-enhanced approaches. The following are the key findings from the expert feedback:

- Both methods exhibited the presence of ambiguity and lack of specificity in several responses. Many points, especially in the manual approach, were phrased vaguely, failing to clearly identify the relevant architectural property or to establish its impact on system qualities.
- The RAG enhanced approach consistently produced more structured and contextually rich content compared to the manual approach, which can be reflected in the higher relevance scores. Risks associated with monolithic architectures were articulated with clarity and scored highly. However, the RAG approach also introduced redundant or overlapping points, especially in risk analyses.
- The manual approach often suffered from terse and incomplete phrasing, resulting in lower clarity and interpretability. Sensitivity points were frequently vague, and many tradeoffs were expressed simply as "X vs. Y" without justification or context. The manual analysis also tended to miss several high-relevance points that the RAG approach identified, especially in complex scenarios such as monolithic scalability and data consistency in microservices.

In conclusion, the RAG-enhanced ATAM approach outperformed the manual method in terms of clarity, coherence, and breadth of analysis. The manual approach often lacked consistency and often failed to communicate ideas clearly, even though some participants provided clear and structured results. However, due to the lack of probes and limited time, we also must consider that the manual approach might have been more detailed and structured if the participants had more time to prepare

and write the results. Also, the results of the manual approach were more dependent on the skill level and writing style of the participants, leading to a lot of variation in the quality of the results. Therefore, the evaluation of the quality of the results between the two should only serve as first observations and tendencies. More investigation and research are needed to draw real conclusions about the relevance of the results.

Inter-Rater Agreement

The mean relevance score provides an overall assessment of how relevant the identified risks, tradeoffs, and sensitivity points are according to the expert raters. However it does not account for the level of agreement between raters, potentially masking inconsistencies in judgment. For example, two raters might assign similar average scores to a category, but if their individual ratings vary significantly, it could indicate a lack of consistency in their assessments. Inter-rater agreement assesses the consistency between evaluators in identifying relevant risks, tradeoffs, and sensitivity points within the architectural analysis. To quantify this agreement, we calculated **Cohen's Kappa** [RS21], which measures the level of consensus beyond what would be expected by chance.

The formula for Cohen's Kappa is as follows:

$$\kappa = \frac{P_o - P_e}{1 - P_e} \tag{7.1}$$

where P_o is the observed agreement, P_e is the expected agreement, and κ is the Cohen's Kappa value. We first create a confusion matrix using the values from Table 7.2, Table 7.3, and Table 7.4 to calculate the observed agreement:

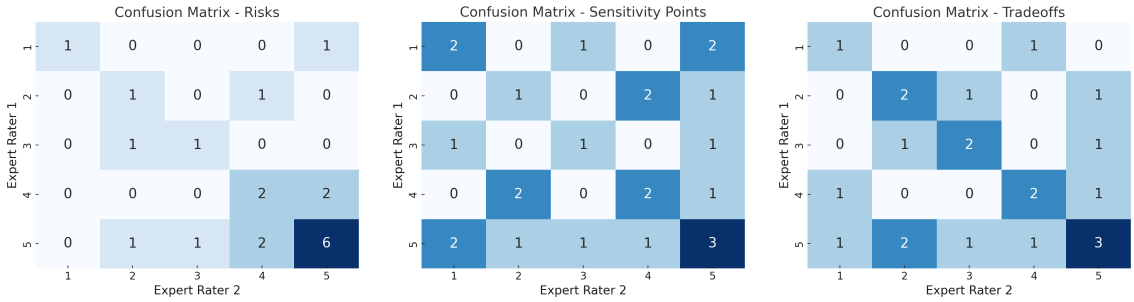

Figure 7.17: Confusion Matrices for Risks, Sensitivity Points, and Tradeoffs

Using the confusion matrices in Figure 7.17, we can calculate the observed agreement for risks, sensitivity points, and tradeoffs.

$$P_o = \frac{\sum_{i=1}^{k} M_{ii}}{N} \tag{7.2}$$

where M_{ii} is the number of agreements on the i-th diagonal element, and N is the total number of ratings. The expected agreement is calculated as follows:

$$P_e = \sum_{i=1}^{k} \left(\frac{\text{Row Total}_i}{N} \times \frac{\text{Column Total}_i}{N} \right) \tag{7.3}$$

where Row Total$_i$ and Column Total$_i$ are the sum of the i-th row and column, respectively. Finally, we can calculate Cohen's Kappa using the observed and expected agreement values. The results are presented in Table 7.8.

Analysis Category	Cohen's Kappa	Interpretation
Risks	0.23	Fair agreement
Sensitivity Points	-0.04	Poor agreement
Tradeoffs	-0.13	Poor agreement

Table 7.8: Cohen's Kappa Values for Inter-Rater Agreement on Risks, Sensitivity points, and Tradeoffs.

Among the three categories, risks had the highest agreement, though it remained below moderate levels. The low agreement for sensitivity points and tradeoffs suggests greater variability in expert assessments. Particularly, the negative Kappa value for sensitivity points indicates that agreement was no better than chance. These variations in agreement may be due to differences in how experts interpret these aspects of architectural analysis, but further investigation is required.

The confusion matrices (see Figure 7.17) provide a detailed view of how often raters agreed or disagreed on specific ratings. The matrices show that while raters frequently agreed on risk assessments, their ratings of sensitivity points and tradeoffs were more inconsistent.

Analysis of Discrepancies in Expert Ratings

As discussed in Section 7.2.1, notable discrepancies were observed in the ratings given by expert raters for certain risks, tradeoffs, and sensitivity points. In several cases, the difference in scores between the two raters exceeded two points on the 5-point Likert scale. To better understand these discrepancies, we analyzed the expert raters' qualitative feedback.

The most commonly cited factors contributing to rating discrepancies include:

- **Differing interpretations of architectural relevance:** One rater considered certain issues (e.g., replication lag, reliance on external services) as primarily operational concerns rather than architectural risks. As a result, these aspects were assigned lower relevance scores, whereas the other rater rated them highly due to their impact on architectural decisions.
- **Uncertainty regarding architectural tradeoffs:** In some cases, one rater questioned whether a tradeoff was properly articulated in the provided descriptions. For example, the tradeoff between *Autoscaling vs. Complexity* (see Table 7.9) was rated as highly relevant by one rater, while the other assigned it a low score, arguing that autoscaling itself is a mechanism rather than an architectural tradeoff.
- **Divergent assumptions about architectural flexibility:** Some disagreements stemmed from differing assumptions about how flexible an architecture can be. One rater argued that a monolithic system could still achieve resilience through design choices, while the other considered reduced resilience an inherent limitation of monolithic architectures.

- **Varying levels of software architecture experience:** Expert Rater 2 explicitly mentioned that he considered himself a novice in software architecture, whereas Expert Rater 1 had prior experience in ATAM and had conducted multiple architectural evaluations. This difference in expertise may have influenced how each rater assessed architectural risks and tradeoffs.

Comparison of Expert Ratings and Justifications Table 7.9 provides specific examples where expert ratings diverged significantly. Below, we analyze key discrepancies and the reasoning behind the differing evaluations.

Scenario, Approach, Decision	Mentioned Risk/Sensitivity Point/-Tradeoff	Rater 1 Score	Rater 2 Score	Difference
Scenario: User Authentication Reliability **Approach:** Monolithic Architecture **Decision:** Unified Codebase and Deployment	**T3** Monolithic approach may lead to reduced resilience	1	4	3
Scenario: User Authentication Reliability **Approach:** Monolithic Architecture **Decision:** Database Management	**R6** Potential for increased latency due to replication lag	5	1	4
Scenario: User Authentication Reliability **Approach:** Monolithic Architecture **Decision:** External Integrations	**R8** High reliability of external services	5	1	4
	S8 Tight coupling leading to bottlenecks	5	2	3
	T8 Monolithic architecture simple initial setup vs. flexibility	5	2	3
Scenario: User Authentication Reliability **Approach:** Monolithic Architecture **Decision:** Scalability and Security	**R11** No proper scaling technology for authentication service	2	5	3
	T10 Use of monitoring tools for scalability	1	5	4
Scenario: User Authentication Reliability **Approach:** Microservices Architecture **Decision:** Service Separation and Responsibilities	**R14** Single point of failure at user service	5	1	4
Scenario: User Authentication Reliability **Approach:** Microservices Architecture **Decision:** Scalability and Security	**S21** Proper management of OAuth 2.0 and RBAC	5	2	3
Scenario: User Authentication Reliability **Approach:** Monolithic Architecture **Decision:** Database Management	**T5** Simplicity vs. Scalability tradeoff	2	5	3
Scenario: User Authentication Reliability **Approach:** Microservices Architecture **Decision:** Infrastructure and Deployment	**T14** Autoscaling vs. Complexity tradeoff	1	5	4

Table 7.9: Architectural aspects with significant discrepancies in expert ratings.

Key Areas of Disagreement

- **Replication Lag and External Dependencies:** Expert Rater 1 rated the risk of replication lag as highly relevant (score: 5), arguing that poor replication configuration can lead to stale reads, retries, and increased response times. Conversely, Expert Rater 2 assigned a low score (1), stating that configuration errors are operational rather than architectural concerns. A

similar disagreement was observed for reliance on external services, where Rater 1 viewed dependency on third-party services as a key architectural risk, whereas Rater 2 did not consider it architecturally relevant.

- **Monolithic System Resilience:** Expert Rater 1 assigned a low score (1) to the claim that monolithic architectures inherently lack resilience, arguing that resilience depends on specific design decisions rather than the architectural style itself. Expert Rater 2, however, rated this risk much higher (4), believing that microservices architectures naturally provide better resilience through service isolation.
- **Scaling and Monitoring:** Expert Rater 2 rated the lack of proper scaling technologies for authentication services as a major risk (score: 5), while Expert Rater 1 gave it a low relevance score (2), arguing that discussing scaling without specifying how it is achieved is not meaningful. Additionally, Rater 2 rated the use of monitoring tools for scalability as highly relevant (5), while Rater 1 assigned a very low score (1), questioning whether monitoring is directly tied to architectural scalability decisions.
- **Tradeoffs Between Simplicity and Scalability:** A key discrepancy emerged in how tradeoffs were interpreted. For example, the tradeoff between simplicity and scalability was rated as highly relevant by Rater 2 (score: 5), while Rater 1 rated it much lower (2), noting that the text did not specify what exactly was being increased or decreased to achieve simplicity or scalability.
- **Autoscaling vs. Complexity:** The largest discrepancy (difference of 4 points) was observed in the rating of the tradeoff between autoscaling and complexity. Expert Rater 1 rated this as irrelevant (1), arguing that autoscaling is a mechanism, not a fundamental architectural quality. Expert Rater 2, however, rated it as highly relevant (5), possibly perceiving autoscaling as an important architectural tradeoff requiring design considerations.

Summary of Findings

- The most significant discrepancies were observed in risks related to replication lag, external service dependencies, and system resilience.
- One rater tended to focus on architectural flexibility, while the other viewed some aspects as inherent limitations of certain architectural styles.
- Differences in experience levels played a role, with Expert Rater 2 occasionally expressing uncertainty about specific architectural concerns.
- Disagreements on tradeoffs often stemmed from differences in how explicit the tradeoff was in the provided descriptions.
- Some discrepancies were due to differing views on whether certain concerns (e.g., monitoring, autoscaling) are architectural decisions or operational considerations.

Overall, the analysis highlights the subjective nature of expert ratings and the importance of clarifying architectural relevance, explicitly articulating tradeoffs, and providing more precise descriptions to reduce interpretational differences. Therefore, the discrepancies in expert ratings underscore the need for clearer guidelines and criteria to ensure consistent and reliable architectural assessments.

7.2.2 Phase 2: Usability Evaluation

In the second phase of the evaluation, we focused on the usability of the RAG ATAM Tool. After the participant tried to solve the model problem using the tool, we gathered feedback through questionnaires and open-ended questions to assess the usability and identify the tool's strengths and weaknesses. We received responses from 10 participants, all students with a background in software engineering and computer science.

The usability of the RAG ATAM Tool was evaluated using a Likert scale questionnaire and open-ended questions. After finishing, the responses were analyzed to assess the ease of use, effectiveness, and perceived strengths and weaknesses of the tool.

Table 7.10 presents the results of the Likert scale questionnaire, where participants rated various aspects of the tool's usability on a scale from 1 to 5. The results indicate that the tool was generally perceived as easy to use and intuitive, with the highest-rated statements being "The tool's interface was intuitive and well-organized" (mean = 4.3, median = 5) and "I thought there was too much inconsistency in this tool's results (reversed)" (mean = 4.6, median = 5), suggesting that users found the results consistent. The question "I was able to complete my tasks efficiently using the tool", received a slightly lower mean score (3.6), indicating that some participants may have encountered challenges in task execution. The lowest-rated item was "The tool supported a comprehensive analysis of the architecture" (mean = 3.5), suggesting potential limitations in the tool's depth of analysis.

The standard deviation values in Table 7.10 indicate the variability in participants' responses. A low standard deviation suggests stronger consensus among respondents, whereas a higher standard deviation indicates more diverse opinions. For example, the statement "The tool effectively identified trade-offs in the architecture" had a standard deviation of 0.72, reflecting relatively consistent responses. Similarly, "The tool was helpful in identifying sensitivity points" also exhibited a low standard deviation (0.72), signifying agreement among participants. Conversely, items such as "The tool was easy to use" (0.99) and "The tool's outputs aligned with my expectations as a software engineer" (0.83) displayed a higher standard deviation, suggesting varying perspectives regarding ease of use and alignment with expectations.

The median values provide an additional robust measure of central tendency, reducing the impact of extreme values. Across most questions, the median value was 4 or 5, indicating a general consensus that the tool was intuitive, provided meaningful insights, and was effective in identifying architectural aspects. The highest median (5) was observed in the statement *Ï thought there was too much inconsistency in this tool's results (reversed)*, suggesting strong agreement that the tool maintained a level of consistency. Regarding both the combination of high median values and relatively low standard deviations in key usability questions suggests that most participants found the RAG ATAM Tool both effective and user-friendly, though certain aspects (e.g., tool usability) had slightly more diverse opinions.

Question	1	2	3	4	5	Mean	Standard Derivation	Median
The tool was easy to use	0	1	2	4	3	3.9	0.99	4
The tool's interface was intuitive and well-organized	0	0	2	3	5	4.3	0.82	5
I was able to complete my tasks efficiently using the tool	0	0	6	2	2	3.6	0.84	3
I thought there was too much inconsistency in this tools' results (reversed)	0	0	2	0	8	4.6	0.84	5
The tool effectively identified trade-offs in the architecture	0	0	3	6	1	3.8	0.72	4
The tool provided actionable insights on risks in the architecture	0	0	3	5	2	3.9	0.73	4
The tool was helpful in identifying sensitivity points	0	0	3	6	1	3.8	0.72	4
The tool supported a comprehensive analysis of the architecture	0	1	4	5	0	3.5	0.77	3
The tool's outputs aligned with my expectations as a software engineer	0	0	3	3	4	4.1	0.83	4

Table 7.10: Results of the Likert Scale Questionnaire

The open-ended questions provided additional insights into the perceived strengths and weaknesses of the tool, as well as areas for improvement. We gathered the feedbacks and categorized them based on the most frequently mentioned aspects.

Most Effective Aspects Of the Tool: Participants appreciated the ability to add or delete PDFs/URLs to enhance knowledge (4 votes), the tool's visual intuitiveness (3 votes), and the provided domain-specific knowledge base (2 votes). The tool was also described as not overwhelming to use after an initial introduction (2 votes).

Challenges and Frustrations: The most frequently mentioned challenge was the lack of a backlog feature (5 votes), followed by difficulties in understanding the purpose of each input field (3 votes). Additionally, some users found the number of required input fields excessive (2 votes), and others noted a significant amount of copy-pasting from external documents (2 votes). One participant mentioned the challenge of providing architectural views as textual input due to the tool's limitations.

Perceived Strengths: The tool's greatest strengths were its ability to provide fast feedback (7 votes), ease of use (3 votes), and discovery of new insights (3 votes). Additionally, participants found that the tool provided a structured overview (2 votes) and that its outputs were often more specific than manual analysis (2 votes). One participant highlighted that the tool could be very beneficial for users with less experience in software architecture.

Perceived Weaknesses: The strongest criticism was the tool's dependence on user input (5 votes), which could lead to inconsistent quality. Other concerns included insufficient domain knowledge (4 votes), occasional superficial outputs (4 votes), and limitations in analyzing architectural layouts beyond textual descriptions. Some participants also mentioned an over-reliance on the LLM (2 votes) and a lack of interactive features (2 votes).

Suggested Improvements: To enhance the tool's capability in analyzing trade-offs, risks, and sensitivity points, participants suggested using larger or more advanced LLM models (5 votes), more specific and granular prompts (4 votes), and additional documents for analysis (2 votes). Other suggestions included improved fine-tuning, the ability to filter PDFs , and better handling of graph-related inputs.

7.3 Discussion

The hypothesis of this thesis states that a Retrieval-Augmented Generation (RAG) enhanced ATAM approach improves the time efficiency of architectural evaluations while maintaining or even enhancing the quality of the analysis results. Furthermore, it was expected that the tool would improve usability for practitioners with varying levels of software architecture expertise.

During the experiment, we encountered several challenges and limitations that may affect the validity of the results (see Section 7.4). For example, the use of students as participants in the manual ATAM analysis may have introduced bias due to their lack of experience in architectural evaluation. Additionally, the small sample size and the application of a model problem for evaluation may limit the generalizability of the results.

Nevertheless, the evaluation results provide first insights and measurements of the tool's performance and usability. The most notable finding was the significant improvement in time efficiency. The RAG enhanced ATAM tool reduced the average analysis time from approximately 13 minutes to only 1.3 minutes, savings roughly 90 percent. This performance was consistent across both runs, demonstrating not only speed but also stability. In contrast, manual analysis times varied significantly depending on the individual analyst's approach and experience level. Therefore, we can conclude that RAG has the potential to significantly accelerate qualitative architectural analysis, especially for less experienced practitioners like students.

Alongside this time improvement, the RAG enhanced approch demonstrated increased coverage by identifying more risks, sensitivity points, and tradeoffs compared to the manual analysis. The quantitative relevance ratings provided by expert raters provides first insights into the relevance of the generated results. In particular, the RAG generated sensitivity points and tradeoffs received notably higher average and median scores than their manually derived counterparts. Particularly for sensitivity points, RAG achieved a mean relevance score of 3.1, compared to 1.75 for the manual approach, indicating a substantial improvement in how clearly and accurately architectural sensitivities were captured. For tradeoffs, RAG's mean ratings ranged from 3.7 to 4.0, whereas the manual results averaged between 2.1 and 2.4. As a conclusion, the RAG-enhanced approach tends to produce more relevant and actionable insights than manual analysis, as perceived by expert raters for this experiment. But it is important that the relevance scores are subjective and may vary between different architecture experts. Furthermore, the small sample size of expert raters

and limitations regarding participants may limit the generalizability of these results, and further evaluations with a larger and more diverse and experienced group of experts and participants are needed to confirm these findings.

From a qualitative perspective, expert rater evaluations revealed that the RAG-enhanced approach yielded outputs with higher clarity, completeness, and contextual relevance, particulary for sensitivity points and tradeoffs. The RAG generated results were often formulated in full sentences with clearly articulated cause-effect relationships and implications, making them more context aware and easier to understand. In addition to that, the RAG tool provided more results in much shorter time, while still keeping the output clear and structured. This contrasts with the manual results, which were often phrased as short fragments or listed as opposing terms without deeper elaboration. Experts noted that such brevity required additional inference, reducing clarity and also relevance in most cases.

However, the RAG-enhanced approach is not without limitations. While it produced produced more detailed and structured outputs, it also introduced some redundancy and overlapping statements (e.g., multiple outputs describing the same issue from different perspectives). Even if it did not significantly detract from overall quality, it added noise to the output and may have negatively affected clarity in a few cases. Furthermore, both approaches occasionally produced results that were not entirely aligned with the architectural context, with the manual approach often missing important details and the RAG tool sometimes generating irrelevant or overly general insights. But in total, the RAG enhanced approach provided more comprehensive and context-aware results, which were generally perceived as more relevant and actionable by expert raters.

Beyond time efficiency and relevance, the RAG ATAM Tool was also evaluated in terms of usability. The Likert scale questionnaire results suggest that the tool was generally well-received by participants, with high ratings for ease of use, interface intuitiveness, and consistency of results.

Regarding the open-ended questions, participants in the second evaluation phase found the tool intuitive to use and helpful in supporting architectural decision-making. The fast and structured feedback provided by the tool was particularly appreciated, as it enabled users to quickly identify architectural risks, sensitivity points, and tradeoffs. Additionally, the provided database of domain-specific knowledge and the visual intuitiveness of the tool were highlighted as key strengths. Some participants also noted that the tool also helped them discover new insights compared to manual analysis, suggesting that the RAG-enhanced approach can enhance architectural exploration and creativity. One participant even suggested that the tool could be very beneficial for users with less experience in software architecture, indicating that the tool's usability and effectiveness could extend to a broader range of practitioners. Last but not least, the consistent structure and readable outputs further enhance its usability in practical settings.

But despite these strengths, the tool also faced several challenges and limitations. The most frequently mentioned issue was the tool's strong dependence on user input, which could lead to inconsistent quality and relevance of the generated results. Other participants noted that the tool sometimes provided superficial outputs or lacked sufficient domain knowledge, limiting its ability to analyze complex architectural layouts or provide detailed insights. One participant also criticized the tool's over-reliance on the LLM and the lack of interactive features, suggesting a more balanced integration of LLM and human input as a potential improvement. And also, missing UX features like a backlog, filtering options, or interactive elements were mentioned as potential areas for improvement, but due to the focus on the core functionality of the tool, these features were not

implemented in the current prototype. Suggested improvements included using larger or more advanced LLMs, potential prompt improvements and structure, fine-tuning the generator LLM, and better handling of graph-related inputs.

But overall, the usability evaluation results suggest that the RAG-enhanced ATAM tool is generally well-received by users and can effectively support architectural analysis tasks. Future work should focus on addressing the identified limitations and challenges, such as improving the tool's consistency and relevance, enhancing its domain knowledge, and refining the user interface to provide a more interactive and user-friendly experience.

Taken together, these findings strongly support the hypothesis, even considering the limitations and challenges encountered during the evaluation. The RAG enhanced approach significantly accelerates architectural analysis, provides more comprehensive and context-aware results, and is generally well-received by users. However, limitations and challenges remain, such as redundancy in the generated outputs, dependence on user input, and occasional superficial results. Additionally, the tool's domain knowledge and ability to analyze complex architectural layouts could be improved. Future work should focus on addressing these limitations, ranging from fine-tuning the LLM to enhancing the user interface and interactive features. To validate the results and further explore the tool's potential, future evaluations should involve a larger and more diverse group of participants and experts, as well as more complex and varied architectural case studies.

7.4 Threats to Validity

While the evaluation shows that the RAG-enhanced ATAM approach can be effective, several factors may affect how reliable or generalizable the results are. This section discusses possible limitations and validity types [YO10] of the study and how we addressed them where possible.

Internal validity is about whether the results we observed really came from the tested approach, or if other factors influenced the outcome. One possible issue is that the manual ATAM analysis was done by students rather than experienced software architects. Since students may lack practical experience, their analyses might have been less detailed or accurate. This could have made the RAG-enhanced approach look better by comparison.

External validity concerns whether the results can be applied to other situations outside of this study. Our evaluation involved a small group of participants and expert raters, which means the results may not reflect how professionals in industry would use or assess the tool. Also, the evaluation was based on just one example project: a fictional e-commerce system. While this helped keep the setup simple and focused, it limits how well the findings can apply to other types of systems, such as embedded or safety-critical architectures. To improve generalizability, future studies should involve more diverse participants and test the tool on different types of real-world systems.

Construct validity looks at whether we measured what we intended to measure. In our case, we used expert ratings to judge the relevance of identified risks, tradeoffs, and sensitivity points. These ratings are subjective and can vary depending on the experts' background or interpretation. This led to some differences in their assessments. While we looked at inter-rater agreement, using more expert raters or clearer rating guidelines could help reduce this variation. Another issue is the

difference in how risks were identified: manual analysts might have missed some points due to time pressure or cognitive load, while the RAG-enhanced tool generated results systematically. This difference may have given the RAG approach an advantage by producing more complete outputs.

Conclusion validity is about whether the results actually support the conclusions we made. One limitation is that we did not perform formal statistical tests (like t-tests [LRY+05] or ANOVA [SW89]) to check if the observed differences were significant. Although the differences were large and consistent, statistical analysis would have made the findings more robust. Another issue is that the study focused only on web-based architectures (monolithic and microservices). This narrow scope may not reflect how the tool performs in other domains. Future evaluations should include a wider range of architectures to test how well the tool works in different contexts.

Despite these threats, several steps were taken to improve the validity of the study. The use of multiple expert raters, who independently evaluated the relevance of all identified risks, tradeoffs, and sensitivity points, helped reduce individual bias and increased confidence in the results. Additionally, the study explicitly acknowledged areas for improvement, such as redundancy reduction and expanded evaluation scope. These considerations offer a clear path for enhancing both the tool and the evaluation process in future research, contributing to greater robustness and generalizability of the findings.

8 Conclusion

This thesis explored how the Architecture Tradeoff Analysis Method (ATAM) can be enhanced through the use of retrieval-augmented generation (RAG) techniques. The main goal was to design, implement, and evaluate a prototype that semi-automates the analysis phase of ATAM using pre-trained LLMs and RAG techniques, meaning that the tool generates architectural risks, sensitivity points, and tradeoffs from provided architectural information. In this chapter, we summarize the key findings of this research, discuss the benefits and limitations of the proposed approach, and outline lessons learned during the study. We also suggest future research directions to improve the tool's effectiveness and usability in real-world software architecture evaluation practices.

8.1 Summary

This thesis introduces a retrieval-augmented generation (RAG)-based approach to enhance the Architecture Tradeoff Analysis Method (ATAM), a structured technique used to evaluate the impact of architectural decisions on software quality attributes. The primary goal of this work was to evaluate the time efficiency of the ATAM analysis phase through automation while preserving, or ideally improving, the quality and relevance of the analytical outcomes. To achieve this, a prototype was designed that LLMs and RAG techniques to identify architectural risks, sensitivity points, and tradeoffs from given architectural inputs.

The approach was evaluated through an experiment in which the outputs of the RAG enhanced tool were compared to those produced via a manual ATAM analysis performed by participants with software engineering knowledge. The evaluation covered both the quantitative performance (in terms of time and coverage) and the qualitative relevance of the generated outputs. The results demonstrated that the RAG-enhanced approach significantly reduced analysis time by approximately 90 percent under the given hardware and processing constraints. Specifically, the average analysis duration dropped from approximately 13 minutes in the manual process to around 1.3 minutes when using the RAG-based tool.

In addition to time efficiency, the RAG based approach also outperformed manual analysis in terms of breadth of coverage. The tool was able to systematically identify a greater number of risks, sensitivity points, and tradeoffs, offering broader insight into architectural implications. Manual analyses were often limited by time and cognitive load, while the RAG-enhanced method provided consistent and exhaustive results across all decision-scenario pairs within the experimental scope.

Crucially, the relevance and quality of the tool's outputs were assessed by two experienced software architecture experts. Their evaluations confirmed that the RAG-generated results were at least comparable to, and often more relevant than, those produced by the manual method. This was especially evident for sensitivity points and tradeoffs, where the RAG outputs achieved higher

average and median relevance scores. Experts noted that the RAG generated outputs were often more clearly phrased, more self-contained, and easier to interpret, especially when compared to the terse and sometimes ambiguous phrasing used in manual results. However, the RAG-generated results occasionally lacked precision, with some sensitivity points being too general or repetitive.

These findings indicate that AI-assisted methods, and retrieval-augmented generation, in particular, have a huge potential to be integrated into architectural evaluation workflows. Not only can such tools significantly reduce the time and effort required to conduct ATAM-style analyses, but they can also support consistency, clarity, and coverage in architectural reasoning. Overall, this thesis demonstrates that the use of generative AI technologies has strong potential to augment traditional architecture evaluation practices, enabling architects to make more informed decisions in less time.

8.2 Benefits

Software architects can use the tool to speed up and automate parts of qualitative architecture analysis. Especially for new or inexperienced architects, the tool can provide guidance and support in identifying architectural risks, sensitivity points, and tradeoffs in an ATAM-style evaluation. Additionally, the structured output enhances consistency across evaluations, providing a common language and format for general framework analysis like ATAM. Software developers and engineers could benefit from the automation of part of the ATAM analysis, which assists in early design decisions and helps teams anticipate potential risks and improve system design.

8.3 Limitations

Despite its benefits, the study has several limitations that should be considered when interpreting the results. The study was conducted using a single model problem, specifically an e-commerce system. While the results are promising, they may not generalize to other domains, such as real-time systems or embedded architectures. The manual ATAM analysis was performed by students rather than experienced software architects, which may have influenced the quality of the manual results. Although expert evaluations were conducted to assess the tool's output, a direct comparison with professional ATAM assessments was not performed. The prototype tool was also limited by the quality of the pre-trained LLM and the RAG techniques used. In addition to that, the implementation of the concept probably could have impacted the results, as the tool was developed in a short time frame and may not have been optimized for performance. Finally, while mean relevance scores and median comparisons suggest that RAG enhances result quality, no formal statistical tests, such as t-tests or ANOVA, were conducted to confirm significance, suggesting that a more rigorous statistical analysis would strengthen the conclusions.

8.4 Lessons Learned

Throughout the research and experimentation process, several key insights emerged. One insight from the implementation and testing is that prompt engineering played a crucial role in determining the quality of AI-generated results. Refinements in query formulation chunking and document retrieval significantly impacted the clarity and coherence of results. Additionally, further expert evaluations and an improved selection of expert raters are necessary due to the subjective nature of assessing architectural relevance. Also, involving multiple expert raters helped reveal differences in interpretation and scoring, emphasizing the need for robust validation. The study also demonstrated the importance of empirical validation, even for controlled experiments. Testing the tool on real-world case studies would help gain more insights about the possibility of using LLMs for architecture analysis, as small-scale evaluations can reveal major differences, but broader testing across diverse software architectures is needed for practical adoption. Future improvements should focus on refining the RAG pipeline, especially in terms of query formulation, result post-processing, LLM fine-tuning, and expert validation.

8.5 Future Work

Several areas offer opportunities for future research and development. The tool should be tested on a wider variety of software architectures, including cloud-based architectures, embedded systems, and safety-critical systems, to assess its generalizability. Improving output quality and filtering by enhancing post-processing techniques to remove redundant results and increase precision in sensitivity point identification is another important area for refinement. Focusing on the implementation, the tool could be optimized for performance, especially in terms of query formulation and database retrieval/content. Referring to the generator model, fine-tuning the LLM or choosing more current models with a focus on reasoning like *deepseek-r1* [DGY+25] could improve the quality of generated results. Future research should also conduct hypothesis testing to confirm whether observed differences between manual and RAG enhanced ATAM are statistically significant. Further integration with architecture modeling tools, such as UML-based or model-driven design tools, could enable direct extraction of architectural concerns from system models. Another promising direction is the development of a hybrid AI-assisted ATAM approach that combines automated AI-driven analysis with human feedback loops, allowing experts to refine or validate generated insights. Last but not least, the RAG enhanced prototype can also be integrated into the vision proposed by Eisenreich et al. [ESW24], complementing their goal of semi-automated architecture candidate generation and evaluation, by providing the possibility to conduct a semi-automated qualitative analysis of the generated architecture candidates.

Bibliography

[ABGM09] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya. "ArcheOpterix: An extendable tool for architecture optimization of AADL models". In: *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009, May 16, 2009, Vancouver, Canada*. IEEE Computer Society, 2009, pp. 61–71. DOI: 10.1109/MOMPES.2009.5069138. URL: https://doi.org/10.1109/MOMPES.2009.5069138 (cit. on pp. 14, 15).

[AG04] M. Ali Babar, I. Gorton. "Comparison of Scenario-Based Software Architecture Evaluation Methods". In: Jan. 2004, pp. 600–607. ISBN: 0-7695-2245-9. DOI: 10.1109/APSEC.2004.38 (cit. on p. 1).

[AGI13] M. Almorsy, J. Grundy, A. S. Ibrahim. "Automated software architecture security risk analysis using formalized signatures". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 662–671. DOI: 10.1109/ICSE.2013.6606612 (cit. on pp. 15, 16).

[ATO25] R. Andrade, J. Torres, I. Ortiz-Garcés. "Enhancing Security in Software Design Patterns and Antipatterns: A Framework for LLM-Based Detection". In: *Electronics* 14.3 (2025). ISSN: 2079-9292. DOI: 10.3390/electronics14030586. URL: https://www.mdpi.com/2079-9292/14/3/586 (cit. on p. 1).

[AWA24] M. Asifuzzaman Jishan, M. Wasif Allvi, M. Ataullah Khan Rifat. "Analyzing User Prompt Quality: Insights From Data". In: *2024 International Conference on Decision Aid Sciences and Applications (DASA)*. 2024, pp. 1–5. DOI: 10.1109/DASA63652.2024.10836452 (cit. on p. 10).

[BBK+24] N. A. Birur, T. Baswa, D. Kumar, J. Loya, S. Agarwal, P. Harshangi. *VERA: Validation and Enhancement for Retrieval Augmented systems*. 2024. arXiv: 2409.15364 [cs.CL]. URL: https://arxiv.org/abs/2409.15364 (cit. on p. 10).

[BCK03] L. Bass, P. Clements, R. Kazman. "Software Architecture In Practice". In: Jan. 2003. ISBN: 978-0321154958 (cit. on p. 3).

[BFK19] A. Busch, D. Fuchß, A. Koziolek. "PerOpteryx: Automated Improvement of Software Architectures". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 162–165. DOI: 10.1109/ICSA-C.2019.00036 (cit. on pp. 14, 15).

[BGMM21] E. M. Bender, T. Gebru, A. McMillan-Major, M. Mitchell. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. 2021, pp. 610–623. DOI: 10.1145/3442188.3445922. URL: https://dl.acm.org/doi/10.1145/3442188.3445922 (cit. on p. 1).

[BJ00] L. Bass, B. E. John. “Achieving usability through software architectural styles”. In: *CHI ’00 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’00. The Hague, The Netherlands: Association for Computing Machinery, 2000, pp. 171–172. ISBN: 1581132484. DOI: 10.1145/633292.633387. URL: https://doi.org/10.1145/633292.633387 (cit. on p. 1).

[BKR09] S. Becker, H. Koziolek, R. H. Reussner. “The Palladio component model for model-driven performance prediction”. In: *J. Syst. Softw.* 82.1 (2009), pp. 3–22. DOI: 10.1016/J.JSS.2008.03.066. URL: https://doi.org/10.1016/j.jss.2008.03.066 (cit. on pp. 1, 14).

[BLBV04] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet. “Architecture-level modifiability analysis (ALMA)”. In: *Journal of Systems and Software* 69.1-2 (2004), pp. 129–147 (cit. on pp. 16, 17).

[BMR+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei. “Language Models are Few-Shot Learners”. In: *arXiv preprint arXiv:2005.14165* (2020). URL: https://arxiv.org/abs/2005.14165 (cit. on p. 1).

[BMS24] S. M. Bsharat, A. Myrzakhan, Z. Shen. *Principled Instructions Are All You Need for Questioning LLaMA-1/2, GPT-3.5/4*. 2024. arXiv: 2312.16171 [cs.CL]. URL: https://arxiv.org/abs/2312.16171 (cit. on pp. 11, 12).

[BOP22] G. Blinowski, A. Ojdowska, A. Przybylek. “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”. In: *IEEE Access* 10 (Jan. 2022), pp. 1–1. DOI: 10.1109/ACCESS.2022.3152803 (cit. on p. 51).

[CB11] T. Clark, B. S. Barn. “Event driven architecture modelling and simulation”. In: *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. IEEE. 2011, pp. 43–54 (cit. on p. 22).

[CG12] J. Cabot, M. Gogolla. “Object Constraint Language (OCL): A Definitive Guide”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by M. Bernardo, V. Cortellessa, A. Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_3. URL: https://doi.org/10.1007/978-3-642-30982-3_3 (cit. on p. 16).

[CKK01] P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Accessed: 2025-Mar-13. Oct. 2001. URL: https://insights.sei.cmu.edu/library/evaluating-software-architectures-methods-and-case-studies/ (cit. on pp. 4, 6).

[CND+22] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus,

D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, N. Fiedel. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL]. URL: https://arxiv.org/abs/2204.02311 (cit. on p. 25).

[CZLZ24] B. Chen, Z. Zhang, N. Langrené, S. Zhu. *Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review*. 2024. arXiv: 2310.14735 [cs.CL]. URL: https://arxiv.org/abs/2310.14735 (cit. on p. 46).

[DGY+25] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948 (cit. on pp. 56, 119).

[DJP+24] A. Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783 (cit. on pp. 1, 56).

[DKS13] T. Denning, T. Kohno, A. Shostack. "Control-Alt-Hack™: a card game for computer security outreach and education (abstract only)". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, p. 729. ISBN: 9781450318686. DOI: 10.1145/2445196.2445408. URL: https://doi.org/10.1145/2445196.2445408 (cit. on p. 16).

[DKV00] A. Deursen, P. Klint, J. Visser. "Domain-Specific Languages". In: *ACM SIGPLAN Notices* 35 (June 2000), pp. 26–36. DOI: 10.1145/352029.352035 (cit. on p. 23).

[DPAM02] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017 (cit. on p. 14).

[ESW24] T. Eisenreich, S. Speth, S. Wagner. "From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures". In: *Proceedings of the 1st International Workshop on Designing Software*. Designing '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 52–55. ISBN: 9798400705632. DOI: 10.1145/3643660.3643942. URL: https://doi.org/10.1145/3643660.3643942 (cit. on pp. 21, 23, 119).

[FAW+08] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi. "Enhanced modeling and solution of layered queueing networks". In: *IEEE Transactions on Software Engineering* 35.2 (2008), pp. 148–161 (cit. on pp. 14, 15).

[FLK+24] X.-Y. Fu, M. T. R. Laskar, E. Khasanova, C. Chen, S. B. TN. *Tiny Titans: Can Smaller Large Language Models Punch Above Their Weight in the Real World for Meeting Summarization?* 2024. arXiv: 2402.00841 [cs.CL]. URL: https://arxiv.org/abs/2402.00841 (cit. on p. 44).

[FMW+05] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, M. Mroz. "Layered queueing network solver and simulator user manual". In: *Dept. of Systems and Computer Engineering, Carleton University (December 2005)* (2005), pp. 15–69 (cit. on p. 15).

[Fri03] E. J. Friedman-Hill. "Jess in action : rule-based systems in Java". In: 2003. URL: https://api.semanticscholar.org/CorpusID:58519195 (cit. on p. 15).

[FZA+24] Y. Fang, J. Zhan, Q. Ai, J. Mao, W. Su, J. Chen, Y. Liu. "Scaling laws for dense retrieval". In: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2024, pp. 1339–1349 (cit. on p. 8).

[GB00] J. van Gurp, J. Bosch. "Automating software architecture assessment". In: *Proceedings of Nordic Workshop on Programming and Software Development Environment Research*. 2000 (cit. on p. 16).

[GSB18] D. Gunawan, C. Sembiring, M. A. Budiman. "The implementation of cosine similarity to calculate text relevance between two documents". In: *Journal of physics: conference series*. Vol. 978. IOP Publishing. 2018, p. 012120 (cit. on pp. 8, 9, 46, 69).

[GXG+24] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, H. Wang. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997 (cit. on p. 10).

[GYZ+25] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning". In: *arXiv preprint arXiv:2501.12948* (2025) (cit. on p. 1).

[Här23] F. Härer. *Conceptual Model Interpreter for Large Language Models*. 2023. arXiv: 2311.07605 [cs.SE]. URL: https://arxiv.org/abs/2311.07605 (cit. on p. 34).

[HDW+23] S. Huang, L. Dong, W. Wang, Y. Hao, S. Singhal, S. Ma, T. Lv, L. Cui, O. K. Mohammed, B. Patra, Q. Liu, K. Aggarwal, Z. Chi, J. Bjorck, V. Chaudhary, S. Som, X. Song, F. Wei. *Language Is Not All You Need: Aligning Perception with Language Models*. 2023. arXiv: 2302.14045 [cs.CL]. URL: https://arxiv.org/abs/2302.14045 (cit. on p. 34).

[HPP+25] J. Hwang, J. Park, H. Park, S. Park, J. Ok. *Retrieval-Augmented Generation with Estimation of Source Reliability*. 2025. arXiv: 2410.22954 [cs.LG]. URL: https://arxiv.org/abs/2410.22954 (cit. on p. 26).

[HW02] K. Henningsson, C. Wohlin. "Understanding the relations between software quality attributes-a survey approach". In: *Proceedings 12th International Conference for Software Quality*. 2002 (cit. on p. 23).

[JSM+23] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825 (cit. on pp. 45, 56).

[KBAW94] R. Kazman, L. Bass, G. Abowd, M. Webb. "SAAM: a method for analyzing the properties of software architectures". In: *Proceedings of 16th International Conference on Software Engineering*. 1994, pp. 81–90. DOI: 10.1109/ICSE.1994.296768 (cit. on pp. 1, 16, 17).

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere. "The architecture tradeoff analysis method". In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*. 1998, pp. 68–78. DOI: 10.1109/ICECCS.1998.706657 (cit. on pp. 1, 3–8, 17, 21, 23, 27, 33).

[Kru95] P. Kruchten. “The 4+1 View Model of architecture”. In: *IEEE Software* 12.6 (1995), pp. 42–50. DOI: 10.1109/52.469759 (cit. on pp. 23, 34).

[LBBH25] E. Lavrinovics, R. Biswas, J. Bjerva, K. Hose. “Knowledge Graphs, Large Language Models, and Hallucinations: An NLP Perspective”. In: *Journal of Web Semantics* 85 (2025), p. 100844. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2024.100844. URL: https://www.sciencedirect.com/science/article/pii/S1570826824000301 (cit. on p. 10).

[LBHB99] L. Lundberg, J. Bosch, D. Häggander, P.-O. Bengtsson. “Quality attributes in software architecture design”. In: *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*. Citeseer. 1999, pp. 353–362 (cit. on p. 1).

[Li23] Y. Li. “A Practical Survey on Zero-shot Prompt Design for In-context Learning”. In: *arXiv preprint arXiv:2309.13205* (2023) (cit. on p. 10).

[LLH+23] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL]. URL: https://arxiv.org/abs/2307.03172 (cit. on pp. 26, 41).

[LLP+20] P. Lewis, P. Lewis, E. Perez, E. Perez, E. Perez, A. Piktus, A. Piktus, A. Piktus, F. Petroni, F. Petroni, V. Karpukhin, V. Karpukhin, N. Goyal, N. Goyal, H. Küttler, H. Küttler, M. Lewis, M. Lewis, M. Lewis, M. Lewis, M. Lewis, W.-t. Yih, W.-t. Yih, T. Rocktäschel, T. Rocktäschel, S. Riedel, S. Riedel, D. Kiela, D. Kiela. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *arXiv: Computation and Language* (2020). DOI: null (cit. on pp. 8, 9).

[LPP+20] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474 (cit. on pp. 1, 17, 25, 53).

[LRY+05] G. B. Limentani, M. C. Ringo, F. Ye, M. L. Bergquist, E. O. McSorley. *Beyond the t-test: statistical equivalence testing*. 2005 (cit. on p. 115).

[LZL+23] M. Li, Y. Zhang, Z. Li, J. Chen, L. Chen, N. Cheng, J. Wang, T. Zhou, J. Xiao. “From quantity to quality: Boosting llm performance with self-guided data selection for instruction tuning”. In: *arXiv preprint arXiv:2308.12032* (2023) (cit. on p. 10).

[MSM+24] A. Masoudifard, M. M. Sorond, M. Madadi, M. Sabokrou, E. Habibi. *Leveraging Graph-RAG and Prompt Engineering to Enhance LLM-Based Automated Requirement Traceability and Compliance Checks*. 2024. arXiv: 2412.08593 [cs.SE]. URL: https://arxiv.org/abs/2412.08593 (cit. on p. 10).

[NAH+01] T. L. Nielsen, J. Abildskov, P. M. Harper, I. Papaeconomou, R. Gani. “The CAPEC database”. In: *Journal of Chemical & Engineering Data* 46.5 (2001), pp. 1041–1044 (cit. on p. 16).

[NBS+24] F. Neha, D. Bhati, D. K. Shukla, A. Guercio, B. Ward. *Exploring AI Text Generation, Retrieval-Augmented Generation, and Detection Technologies: a Comprehensive Overview*. 2024. arXiv: 2412.03933 [cs.AI]. URL: https://arxiv.org/abs/2412.03933 (cit. on p. 44).

[NF96] M. Neil, N. Fenton. “Predicting software quality using Bayesian belief networks”. In: Jan. 1996, pp. 217–230 (cit. on pp. 15, 16).

[NMDM25] Z. Nussbaum, J. X. Morris, B. Duderstadt, A. Mulyar. *Nomic Embed: Training a Reproducible Long Context Text Embedder*. 2025. arXiv: 2402.01613 [cs.CL]. URL: https://arxiv.org/abs/2402.01613 (cit. on pp. 56, 63).

[Pal23] Y. Paliwal. "Understanding Retrieval-Augmented Generation (RAG) in Generative AI: An Overview". In: *Medium* (2023). https://medium.com/@yashpaliwal42/understanding-retrieval-augmented-generation-rag-in-generative-ai-an-overview-914ef820d0f0 (cit. on p. 8).

[PATK25] J. Park, K. Atarashi, K. Takeuchi, H. Kashima. *Emulating Retrieval Augmented Generation via Prompt Engineering for Enhanced Long Context Comprehension in LLMs*. 2025. arXiv: 2502.12462 [cs.CL]. URL: https://arxiv.org/abs/2502.12462 (cit. on p. 26).

[PSX+24] L. Pan, M. Saxon, W. Xu, D. Nathani, X. Wang, W. Y. Wang. "Automatically Correcting Large Language Models: Surveying the Landscape of Diverse Automated Correction Strategies". In: *Transactions of the Association for Computational Linguistics* 12 (May 2024), pp. 484–506. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00660. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00660/2369509/tacl_a_00660.pdf. URL: https://doi.org/10.1162/tacl%5C_a%5C_00660 (cit. on p. 8).

[PTZ+24] S. Pawar, S. M. T. I. Tonmoy, S. M. M. Zaman, V. Jain, A. Chadha, A. Das. *The What, Why, and How of Context Length Extension Techniques in Large Language Models – A Detailed Survey*. 2024. arXiv: 2401.07872 [cs.CL]. URL: https://arxiv.org/abs/2401.07872 (cit. on p. 11).

[RH24] D. Rouabhia, I. Hadjadj. *Enhancing Class Diagram Dynamics: A Natural Language Approach with ChatGPT*. 2024. arXiv: 2406.11002 [cs.SE]. URL: https://arxiv.org/abs/2406.11002 (cit. on pp. 18, 34).

[RHJ21] Q. Rouland, B. Hamid, J. Jaskolka. "Specification, detection, and treatment of STRIDE threats for software components: Modeling, formal methods, and tool support". In: *Journal of Systems Architecture* 117 (Aug. 2021), p. 102073. DOI: 10.1016/j.sysarc.2021.102073 (cit. on p. 16).

[Rob23] J. Robinson. "Likert Scale". In: *Encyclopedia of Quality of Life and Well-Being Research*. Ed. by F. Maggino. Cham: Springer International Publishing, 2023, pp. 3917–3918. ISBN: 978-3-031-17299-1. DOI: 10.1007/978-3-031-17299-1_1654. URL: https://doi.org/10.1007/978-3-031-17299-1_1654 (cit. on p. 90).

[RS21] G. Rau, Y.-S. Shih. "Evaluation of Cohen's kappa and other measures of inter-rater agreement for genre analysis and other nominal data". In: *Journal of english for academic purposes* 53 (2021), p. 101026 (cit. on p. 106).

[Rum16] B. Rumpe. *Modeling with UML*. Vol. 98. Springer, 2016 (cit. on p. 34).

[RZ09] S. Robertson, H. Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond". In: *Foundations and Trends in Information Retrieval* 3 (Jan. 2009), pp. 333–389. DOI: 10.1561/1500000019 (cit. on pp. 8, 9).

[Sha24] W. Shafik. "Introduction to ChatGPT". In: *Advanced applications of generative AI and natural language processing models*. IGI Global, 2024, pp. 1–25 (cit. on pp. 1, 17, 18).

[SKB24] L. Seymour, B. Kutukcu, S. Baidya. *Large Language Models on Small Resource-Constrained Systems: Performance Characterization, Analysis and Trade-offs*. 2024. arXiv: 2412.15352 [cs.LG]. URL: https://arxiv.org/abs/2412.15352 (cit. on p. 25).

[SM09] J. Savolainen, V. Myllarniemi. "Layered architecture revisited—Comparison of research and practice". In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE. 2009, pp. 317–320 (cit. on p. 22).

[SMB24] S. Speth, N. Meissner, S. Becker. "ChatGPT's Aptitude in Utilizing UML Diagrams for Software Engineering Exercise Generation". In: July 2024, pp. 1–5. DOI: 10.1109/CSEET62301.2024.10663027 (cit. on p. 18).

[SW89] L. St»hle, S. Wold. "Analysis of variance (ANOVA)". In: *Chemometrics and Intelligent Laboratory Systems* 6.4 (1989), pp. 259–272. ISSN: 0169-7439. DOI: https://doi.org/10.1016/0169-7439(89)80095-4. URL: https://www.sciencedirect.com/science/article/pii/0169743989800954 (cit. on p. 115).

[Swa23] A. Swain. "Build Domain-Specific LLMs Using Retrieval Augmented Generation". In: *Medium* (2023). https://medium.com/@avijitswain11/build-domain-specific-llms-using-retrieval-augmented-generation-b3b26bb112c0 (cit. on p. 9).

[TG19] N. Torvekar, P. Game. "Microservices and Its Applications An Overview". In: *International Journal of Computer Sciences and Engineering* 7 (Apr. 2019), pp. 803–809. DOI: 10.26438/ijcse/v7i4.803809 (cit. on p. 22).

[TOW17] J. J. Y. Tan, K. N. Otto, K. L. Wood. "Relative impact of early versus late design decisions in systems development". In: *Design Science* 3 (2017), e12. DOI: 10.1017/dsj.2017.13 (cit. on p. 1).

[WFH+23] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. C. Schmidt. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. arXiv: 2302.11382 [cs.SE]. URL: https://arxiv.org/abs/2302.11382 (cit. on pp. 10–12, 69).

[WWLL24] C. Wang, B. Wang, P. Liang, J. Liang. *Assessing UML Models by ChatGPT: Implications for Education*. 2024. arXiv: 2412.17200 [cs.SE]. URL: https://arxiv.org/abs/2412.17200 (cit. on pp. 17, 34).

[XAA24] Y. Xie, K. Aggarwal, A. Ahmad. "Efficient Continual Pre-training for Building Domain Specific Large Language Models". In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by L.-W. Ku, A. Martins, V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10184–10201. DOI: 10.18653/v1/2024.findings-acl.606. URL: https://aclanthology.org/2024.findings-acl.606/ (cit. on p. 25).

[XCW+25] B. Xu, Y. Chen, Z. Wen, W. Liu, B. He. *Evaluating Small Language Models for News Summarization: Implications and Factors Influencing Performance*. 2025. arXiv: 2502.00641 [cs.CL]. URL: https://arxiv.org/abs/2502.00641 (cit. on pp. 45, 56).

[Xu08] J. Xu. "Rule-based automatic software performance diagnosis and improvement". In: *Proceedings of the 7th International Workshop on Software and Performance*. WOSP '08. Princeton, NJ, USA: Association for Computing Machinery, 2008, pp. 1–12. ISBN: 9781595938732. DOI: 10.1145/1383559.1383561. URL: https://doi.org/10.1145/1383559.1383561 (cit. on p. 15).

[YF11] X. Yuan, E. B. Fernandez. *Patterns for Business-to-consumer E-Commerce Applications*. 2011. arXiv: 1108.3342 [cs.SE]. URL: https://arxiv.org/abs/1108.3342 (cit. on p. 76).

[YO10] C.-h. Yu, B. Ohlund. *Threats to validity of research design*. 2010 (cit. on p. 114).

[YWL+24] S. Yang, T. Wu, S. Liu, D. Nguyen, S. Jang, A. Abuadbba. *ThreatModeling-LLM: Automating Threat Modeling using Large Language Models for Banking System*. 2024. arXiv: 2411.17058 [cs.CR]. URL: https://arxiv.org/abs/2411.17058 (cit. on p. 1).

[ZDD+24] Y. Zhao, L. Du, X. Ding, K. Xiong, Z. Sun, J. Shi, T. Liu, B. Qin. *Deciphering the Impact of Pretraining Data on Large Language Models through Machine Unlearning*. 2024. arXiv: 2402.11537 [cs.CL]. URL: https://arxiv.org/abs/2402.11537 (cit. on p. 25).

[ZSG+20] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, B. Dolan. "Dialogpt: Large-scale generative pre-training for conversational response generation". In: *arXiv preprint arXiv:1911.00536* (2020). URL: https://arxiv.org/abs/1911.00536 (cit. on p. 1).

[ZYW+24] S. Zhao, Y. Yang, Z. Wang, Z. He, L. K. Qiu, L. Qiu. *Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely*. 2024. arXiv: 2409.14924 [cs.CL]. URL: https://arxiv.org/abs/2409.14924 (cit. on p. 9).

All links were last followed on March 27, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature